SYSTEM DESIGN INTERVIEW

GROKS KING

“Great companies are built on great products.”

– Elon Musk

SYSTEM DESIGN

INTERVIEW

BY

Groks King

ABOUT THE AUTHOR

GROKS KING

BRITISH ACADEMIC &
BASED IN LONDON,
ENGLAND .

WE’RE IN THE CHANGING LIVES BUSINESS.

© Copyright 2021 by (United Arts Publishing, England.) - All rights reserved.

This document is geared towards providing exact and reliable information in regards to the topic and issue covered. The publication is sold with the idea that the publisher is not required to render accounting, officially permitted, or otherwise, qualified services. If advice is necessary, legal or professional, a practiced individual in the profession should be ordered.

- From a Declaration of Principles which was accepted and approved equally by a Committee of the American Bar Association and a Committee of Publishers and Associations.

In no way is it legal to reproduce, duplicate, or transmit any part of this document in either electronic means or in printed format. Recording of this publication is strictly prohibited and any storage of this document is not allowed unless with written permission from the publisher. All rights reserved.

The information provided herein is stated to be truthful and consistent, in that any liability, in terms of inattention or otherwise, by any usage or abuse of any policies, processes, or directions contained within is the solitary and utter responsibility of the recipient reader. Under no circumstances will any legal responsibility or blame be held against the publisher for any reparation, damages, or monetary loss due to the information herein, either directly or indirectly.

Respective authors own all copyrights not held by the publisher.

The information herein is offered for informational purposes solely, and is universal as so. The presentation of the information is without contract or any type of guarantee assurance.

The trademarks that are used are without any consent, and the publication of the trademark is without permission or backing by the trademark owner. All trademarks and brands within this book are for clarifying purposes only and are the owned by the owners themselves, not affiliated with this document.

WANT FREE GOODIES?

EMAIL ME AT:

mindsetmastership@gmail.com

FOLLOW US ON INSTAGRAM!
@MINDSETMASTERSHIP

MASTERSHIP BOOKS
UK | USA | CANADA | IRELAND | AUSTRALIA
INDIA | NEW ZEALAND | SOUTH AFRICA | CHINA
MASTERSHIP BOOKS IS PART OF THE UNITED ARTS PUBLISHING HOUSE GROUP OF
COMPANIES BASED IN LONDON, ENGLAND, UK.

FIRST PUBLISHED BY MASTERSHIP BOOKS (LONDON, UK), 2021
ISBN: 9781915002037

TEXT COPYRIGHT © UNITED ARTS PUBLISHING
ALL RIGHTS RESERVED. WITHOUT LIMITING THE RIGHTS UNDER COPYRIGHT RESERVED ABOVE, NO PART OF THIS PUBLICATION MAY BE REPRODUCED, STORED IN OR INTRODUCED INTO A RETRIVAL SYSTEM, OR TRANSMITTED, IN ANY FORM OR BY ANY MEANS (ELECTRONIC, MECHANICAL, PHOTOCOPYING, RECORDING OR OTHERWISE), WITHOUT THE PRIOR WRITTEN PERMISSION OF BOTH THE COPYRIGHT OWNER AND THE ABOVE PUBLISHER OF THIS BOOK.

COVER DESIGN BY RICH © UNITED ARTS PUBLISHING (UK)
TEXT AND INTERNAL DESIGN BY RICH © UNITED ARTS PUBLISHING (UK)
IMAGE CREDITS RESERVED.
COLOUR SEPARATION BY SPITTING IMAGE DESIGN STUDIO
PRINTED AND BOUND IN GREAT BRITAIN
NATIONAL PUBLICATIONS ASSOCIATION OF BRITAIN
LONDON, ENGLAND, UNITED KINGDOM.
PAPER DESIGN UAP
ISBN: 9781915002037
(PAPERBACK)
A723.5
TITLE: **SYSTEM DESIGN**
DESIGN, BOUND & PRINTED**:**
LONDON, ENGLAND, GREAT BRITAIN.

CONTENTS

INTRODUCTION TO SYSTEM DESIGN INTERVIEW

CAPACITY ESTIMATION

FRAMEWORK FOR SYSTEM DESIGN INTERVIEW

ID GENERATOR SYSTEM DESIGN

WEB CRAWLER SYSTEM DESIGN

DISTRIBUTED KEY-VALUE STORE SYSTEM DESIGN

NOTIFICATION SERVICE DESIGN

GOOGLE DRIVE DESIGN

AUTOCOMPLETE SYSTEM DESIGN

API RATE-LIMITING SYSTEM DESIGN

FACEBOOK NEWSFEED SYSTEM DESIGN

CHAT MESSENGER SYSTEM DESIGN

CONSISTENT HASHING IN SYSTEM DESIGN

YOUTUBE SYSTEM DESIGN

URL SHORTENING

CONCLUSION

0

INTRODUCTION TO SYSTEM DESIGN INTERVIEW

System Design

This is the process of defining the aspects of a system, such as the architecture, modules, and components and the many interfaces between those components and the data that flows through it. It's designed to meet a company's or organization's particular demands and requirements by constructing a well-functioning system. It is the systematic approach to the design of a system is referred to as systems design. It can be done from the bottom up or from the top down. Still, either way, the process is systematic. It considers all similar variables of the system that needs to be implemented, from the architecture to the required hardware and software, all the way down to the data and how it moves and evolves throughout its journey through the system. The terms "system design," "systems analysis," "systems engineering," and "systems architecture" are all used interchangeably. Engineers were striving to tackle complicated control and communications difficulties before World War II when the systems design method arose. They needed to formalize their work into a formal discipline with suitable techniques, especially in emerging fields like information theory, operations research, and general computer science.

System Design Interview

A system design interview is held to provide candidates such as programmers, designers, developers, and software engineers the chance to demonstrate their competence in the area by using their knowledge to address a real-world problem that a firm may be experiencing. Typically, the system design interview takes place later in the interview process. It's a test to assess how effectively you work in a group and solve problems using open-ended questions to come up with the best potential answers. A system design interview examines your problem-solving and system-designing processes to assist clients. It's your chance to convince the hiring manager and likely team that you're a helpful asset by tangibly demonstrating your talents and knowledge.

There is no set procedure for conducting a system design interview. Moreover, large systems have so many fundamentally unclear aspects that it would be difficult to find a solution without first clarifying at least a few of them. The purpose of ambiguous system design questions is to allow you to showcase your qualifications. Before responding, you might ask questions to limit the scope, provide guidance, and clarify any expectations. In your system design interview, you may be asked some of these questions:

- How to build a social media network like Facebook, Twitter?
- How to design a URL shortening service like TinyURL?
- How to design a search engine?
- How to design a WebCrawler?
- How to design shared services?

Such a question may appear to any applicant who has no prior knowledge of building systems to be highly unfair. Furthermore, there is rarely a single proper response to such questions. How you respond to the question reveals your professional expertise and background experience. That is the criterion through which the interviewer will assess you. Because the questions are purposefully vague, rushing right into developing the answer without comprehending them will result in failure. Spend a few minutes interrogating the interviewer to get a complete understanding of the system's capabilities. Never make assumptions about anything that aren't mentioned openly.

What was the process of designing these? Another thing to remember is that the interviewer expects the candidate's analytical abilities and problem-solving skills to be equivalent to their experience. If you have a couple of years of software development skills, you can expect to know a certain amount of information. You should refrain from asking simple inquiries that would be reasonable if you were a recent graduate. You should plan enough ahead of time for this. Most interview questions are based on real-life items, difficulties, and obstacles, so carefully go through genuine projects and procedures before the interview. In the interview, the conversation process itself is essential than the eventual answer to the problem. And it is the candidate who should drive the discussion, which should include both the broad and detailed aspects of the subject, as a result, including the interviewer in the problem-solving process by talking with them step by step as you go.

At first glance, design questions may appear to be difficult and frightening. However, regardless of the problem's level of complexity, a top-down and modularization strategy can significantly aid in its resolution. As a result, you should split the problem down into modules and work on each one separately. Then, by reducing the issue to the level of a known algorithm, each component may be addressed as a subproblem. This technique will not only make the design much more straightforward for you and the interviewer, but it will also make the interviewer's judgment a lot easy. You could run across various obstacles while working on the solution. This is quite normal. When dealing with obstacles, your system may need a load balancer with several computers behind it to handle user requests, or the data may be so large that you need to spread your database across multiple servers. It's also conceivable that the interviewer wishes to steer the conversation in a specific direction. If this is the case, you should proceed in that path and dig deep, disregarding everything else. If you are stuck, you can ask for a clue to help you get moving again. Please remember that each option is a trade-off; so, altering one item might make another worse. What matters here is your ability to discuss these trade-offs and assess their influence on the system while keeping all restrictions and use scenarios in mind. After you've finished your high-level design and double-checked that the interviewer approves, you can move on to making it more comprehensive. Typically, this entails scaling your system.

Database (DB)

A database is in charge of storing and retrieving data for an application. Because it is a program that handles access to physical data storage, a database is sometimes known as a Database Management System (DBMS). A database's primary functions are to store data, update and delete data, return data according to a query and database administration. And, to provide these data services, a database must be dependable, efficient, and accurate. Databases may be divided into two categories at a high level: relational and non-relational.

Relational Database

A relational database is one in which the relationships between the items recorded in the database are tightly enforced. These connections are usually made feasible by requiring the database to represent each entity as a structured table, with zero rows representing records, entries, and one or more columns representing attributes and fields. We may verify that each entry contains the appropriate data by imposing such a structure on an object. It improves consistency and the capacity to form close connections between the entities. The majority of relational databases offer SQL - Structured Query Language - is a database querying language. This programming language was created mainly for interacting with the contents of a structured (relational) database. The two ideas are inextricably linked, to the point that a relational database is frequently referred to as a "SQL database" and occasionally pronounced as a "sequel" database. In general, SQL (relational) databases are regarded to handle more sophisticated queries combining many fields, filters, and conditions than non-relational databases. The database itself processes these queries and returns results that match. The PostgreSQL database, sometimes known as "Postgres," is a popular and well-liked instance of a relational database.

Transactions

A transaction is a single unit of work that consists of a sequence of database activities. A transaction's actions are either successful or unsuccessful. A collection of characteristics that characterize a decent relational database's transactions is known as ACID transactions. ACID is the acronym for "Atomic, Consistent, Isolation, Durable."

Atomicity: When a single transaction consists of many operations, atomicity requires the database to ensure that if any process fails, the entire transaction fails as well. It's a case of "all or nothing." If the transaction succeeds, you'll know that all of the sub-operations succeeded as well, and if one operation fails, you'll know that all of the operations that happened with it failed as well.

Consistent: Each transaction in a database must be legitimate according to the database's specified rules, and when the database changes state, some information must be updated. This change must be valid and not corrupt the data. Each transaction changes the database's condition from one that is valid to another that is valid. The following is an example of consistency. Every "read" operation gets the latest "write" operation results.

Isolation: It implies you may execute many transactions on a database simultaneously. However, the database will end up in a state that seems like each operation was run sequentially, like a backlog of operations.

Durable: The term "durability" refers to the assurance that once data is saved in a database; it will stay there indefinitely. It will be permanent, meaning it will be held on the disk rather than in memory.

Non-Relational Databases

Non-relational databases are designed to meet unique scalability, schema flexibility, or specialized query functionality requirements. There are many different types of non-relational databases. Non-Relational Databases are sometimes referred to as NoSQL databases. Non-relational databases, on the other hand, frequently include SQL or SQL-like query capabilities to make developers' life easier. In relational and non-relational systems, however, the fundamental execution of these queries will be substantially different. Because they're addressing other use cases with varied priorities between availability and consistency, non-relational databases are classified as either AP or CP databases. The eventual consistency concept is used in AP non-relational databases to ensure that consistency occurs over time, even if it isn't guaranteed precisely when a transaction completes.

Graph Database

Nodes and edges are used to model data in graph databases. Because they are good at expressing data with many relationships, they are the most comparable non-relational databases to a relational data model. Because data isn't stored in tables, the significant advantage of a graph database is that queries don't require joins to follow relationship edges. As a result, queries that traverse numerous edges of a graph, such as social network analytics, are ideally suited to them.

Key-Value Store

A key-value store is identical to a document store, except that the data contained in the value is transparent. Because the key-value store has no idea what's in the value, it only supports read, overwrite, and delete actions. There are no schemas, joins, or indexes; it's just a massive hash table. They are simple to scale due to their low overhead. Caching implementations benefit greatly from key-value storage. This type of database is known as an Object Store or Blob Store when the values must be significant. The data may be serialized and optimized for large file sizes in this scenario. Videos, pictures, music, disk images, and log binaries are examples of use cases.

Search Engine Database

A search engine database provides a specialized function of full-text search across very large volumes of unstructured text data. The data might come from a variety of places, and database users could desire a fuzzy search, which means the results might not perfectly fit the search term. Searching websites (such as Google) is a well-known example, but it may also be helpful in other situations, such as searching and debugging enormous amounts of system logs.

Time Series Database

A time-series database is designed for data entries that must be sorted chronologically. The most common application is for storing real-time data streams from systems monitors. Because time series databases need a lot of writing, they typically include services for sorting streams as they arrive to ensure that they are appended in the correct order. These datasets may be divided into periods.

Scaling

As database volumes continue to rise, the ability to deploy databases in distributed clusters is more critical than ever. Because of the functionality they give, different databases scale better or poorly. Scaling can be divided into two categories:

- Vertical Scaling
- Horizontal Scaling

Vertical Scaling: Vertical scaling, also known as scaling up, refers to the act of increasing the processing capability (CPU, RAM) of your servers. Vertical scaling is simple, but the overall memory capacity is significantly smaller.

Horizontal Scaling: It is also known as scale-out, allows you to expand your pool of resources by adding more servers. Horizontal scaling has far more total computing and storage capacity and can be scaled dynamically without causing downtime. Due to the limits of vertical scaling, horizontal scaling is preferable for large-scale applications.

Database Indexing

Indexing is a method of quickly getting to a record with matching values instead of going through each row. Indexes are a type of data structure added to a database to search the database for specific fields quickly. So, if the census bureau has 120 million entries with names and ages, and you frequently need to get lists of individuals in a particular age group, you'd index the database using the age property. Indexing is a fundamental feature of relational databases. However, it is also frequently available in non-relational databases. The advantages of indexing are thus theoretically accessible for both sorts of databases, which is extremely useful in reducing lookup times.

Schemas

A schema's job is to determine the form of a data structure and what data types may go there. A schema can describe database-level structures like tables and indexes and data-level limitations like field types in databases. Schemas can be strongly enforced throughout the whole database, weakly enforced on a subset of the database, or completely absent. A single schema can be used for the entire database, or various schemas can be used for distinct entries. When a schema is carefully followed, it is reasonable to expect that all system queries will produce data consistent with the schema assumptions.

Replication and Sharding

Duplicating or making copies of your database is referred to as replication. Replication ensures database redundancy if one fails. However, since the replicas are supposed to have the same data, it begs the challenge of synchronising data among them. Write and update actions to a database that can be replicated synchronously (at the same time as the primary database) or asynchronously (later). The period between synchronizing the main and replica databases is entirely dependent on your requirements. If you genuinely want the state between the two databases to be consistent, you'll require fast replication. You should also make sure that if the write to the replica fails, the write to the main database fails as well. Data sharding divides a large database into smaller databases. Depending on the structure of your data, you may determine how you wish to shard it. It may be as easy as saving every 5 million rows in a separate shard, or it could be more complex, depending on your data, needs, and regions serviced.

Cache

A cache is a hardware or software that stores data that can be retrieved more quickly than data from other sources. Caches are commonly used to keep track of frequently requested answers. It may also be used to keep track of the results of lengthy computations. Caching is the process of storing data at a location other than the original data source so that it may be accessed more quickly. Caching, like load balancers, may be utilized in a variety of areas across the system. Caching is used to prevent having to repeat the exact difficult computation. It is used to lower the time complexity of algorithms. For example, we utilize the memory approach to minimize the time complexity of dynamic programming. Caching as a notion is similarly comparable in terms of system design ideas. Caching is a technique for speeding up a system. We need to employ caching to reduce a system's latency. Using caching to decrease network queries is another reason to do so. Caches often save the most recently accessible data since recently requested material is likely to be asked repeatedly. As a result, caching should be used to reduce data retrieval operations from the database in such circumstances.

However, there are other methods for deciding which data should be kept in a cache. A Cache Hit occurs when requested data is discovered in the cache. When requested data is not located in the cache, it has a detrimental impact on the system. It's known, as a Cache Miss. It's a metric for bad design. To enhance performance, we need to increase the number of hits while decreasing the miss rate. If the source of data is updated but the cache is not, data might become stale. If stale data isn't a concern in a system, caching may dramatically increase speed. Assume we're creating a system to track how many people have seen a video on YouTube. It doesn't matter too much if various users see different watch count values. As a result, staleness isn't an issue in these situations.

Distributed Cache

A caching layer in a distributed system can be implemented on each computer in the application service cluster or a cluster separate from the application service. Each application computer in the service cluster has a private cache in its memory. Because each computer has its cache, the cached data will get out of sync depending on what requests have been sent and when. The major benefit of in-memory distributed caches is speed because the cache is in memory. It will run much quicker than a shared caching layer that relies on network requests. For private cache implementations, Redis and Memcached are two popular options. In an isolated layer, there is a shared cache. In reality, the program may be unaware that a cache exists. It enables the cache and application layers to scale independently and the usage of the same cache by many distinct types of application services.

The application layer must detect the shared cache's availability and transition to the primary data storage if the shared cache becomes unavailable. The same resilience considerations apply to a shared cache built on its cluster as they do to any distributed system, including failovers, recovery, partitioning, rebalancing, and concurrency. Just as various CPU architectures have varied caching, private and shared caches can be utilized simultaneously. If the shared cache layer fails, the private cache can be used as a fallback to keep the benefits of caching.

When should you utilize caching?

- Caching comes in handy when the data you require is difficult to obtain, whether due to sluggish hardware, having to go across the network, or sophisticated calculation.

- Caching is useful in systems when there are many requests for static or slow-changing data because the cache will always be up to date.
- Caching can also minimize the amount of data stored in main data storage, lowering service costs and allowing for faster response times.

Cache Replacement Policies

The cache replacement policy is a critical component of a caching layer's success. The replacement policy (also known as eviction policy) determines what memory to release when the cache is full. A good replacement strategy will use the concept of locality to optimize for cache hits, ensuring that the cached data is as relevant as feasible to the application. Because replacement policies are tailored to specific use situations, there are many methods and implementations to select from. Cache Replacement Fundamental Policies are:

- Least Recently Used (LRU): The oldest entry in the cache will be released using an LRU replacement strategy. LRU is an excellent default replacement policy since it works well and is very simple to comprehend. To implement LRU, the cache uses aging bits to keep track of recency, which must be changed on every entry every time data is requested. Although LRU makes efficient judgments regarding what data to discard, the computational expense of keeping track of aging bits forces approximations such as the clock replacement strategy.
- Least Frequently Used (LFU): The entry in the cache utilized the least frequently will be released under an LFU replacement strategy. A basic access count in the entry information is used to

measure frequency. LFU replacement rules are handy in situations where data is accessed rarely and is unlikely to be reused. For example, an encyclopedia-like site might include popular entries (such as those concerning celebrities) and obscure items.

- Expiration Policies: There is typically an explicit expiry or retention policy in distributed systems to ensure that there is always room in the cache for fresh data. The Time To Live option sets the time after which a resource is released if it hasn't been used (TTL). One method to improve a cache layer is to find the correct TTL. Explicit removal policies are sometimes event-driven, such as releasing an entry when it is written to.

1

CAPACITY ESTIMATION

After your interviewer presents you with the problem, spend a few moments asking clarifying questions and ascertain what they are looking for specifically. The worst-case scenario is that you begin in the entirely incorrect path due to your failure to ask a few questions. You only have a short time during the interview, so you want to focus on the most critical points. The traffic volume will determine the complexity of the system it must manage, so be sure to collect this data. You do not have to over-engineer anything if the traffic is minimal, but you also don't want to end up with an app that cannot scale due to improper design. Consider the number of users the app will have, the mean quantity of data per request, the length of time data must be kept, and the system's reliability and availability requirements. Capacity Estimation is a very informative activity. It can assist you in determining the viability of a design and provide better alternatives. It may be quite beneficial to approach it as part of the design process in any significant change defined as how the change influences relative scale.

Using the information you obtained in the first step, you may start making some preliminary estimations and generalizations for storage and bandwidth requirements. This procedure will need simple math, such as multiplying the number of users by the average request size and the estimated number of requests per user per day.

Numbers to Note

To assess design options, you must first have a solid understanding of how long routine procedures will take.

L1 cache reference = 0.5 ns

Branch mispredict = 5 ns

L2 cache reference = 7 ns

Mutex lock/unlock = 100 ns

Main memory reference = 100 ns

Compress 1K bytes with Zippy = 10,000 ns

Send 2K bytes over 1 Gbps network = 20,000 ns

Read 1 MB sequentially from memory = 250,000 ns

Round trip within same datacenter = 500,000 ns

Disk seek = 10,000,000 ns

Read 1 MB sequentially from network = 10,000,000 ns

Read 1 MB sequentially from disk = 30,000,000 ns

Send packet CA->Netherlands->CA = 150,000,000 ns

Important Information to Note

- Take note of the substantial variations in performance between the various alternatives.
- Because data centers are spread apart, sending data between them takes a considerable time.
- Memory is quick, while disks take a long time.
- An inexpensive compression technique can save a lot of network traffic (by a factor of two).
- The cost of writing is 40 times that of reading.
- Data that is shared globally is costly. Distributed systems have this as a fundamental drawback. Because transactions become serialized and sluggish due to lock contention in shared highly written objects, performance suffers.

- Writes an architect for scaling.
- Write conflict should be kept to a minimum.
- Optimize on a large scale. Make your writing as parallel as possible.

Considerations

Calculations on the back of the envelope are meant to be imprecise and approximate.

Don't get too caught up in the details of your computations. If necessary, round numbers. Make assumptions that are simple to compute, such as 200 KB or 500 million users. Don't search for genuine figures and come up with strange or overly precise values like 187 KB or 324 million users. This will make computations more complex, and interviewers may not expect you to be overly exact. After you've written down your assumptions, always check with the interviewer to see if they want to change them to fit what they're thinking. This demonstrates that you conduct cross-verification and are prepared to alter the scale based on the actual requirement.

Calculations to keep in mind:

1 KB data (1000 bytes) * 1 million user (1,000,000) = 1,000,000,000 = 1 GB of storage (For example: 1 KB data (1000 bytes) * 1 million user (1,000,000) = 1,000,000,000 = 1 GB of storage)

x megabytes * y million users = xy terabytes

Assumptions

- **Storage Assumptions**

Assumption for data storage in general.

A single char = two bytes.

8 bytes (long or double).

Average images = 200 KB; good photos = 2 MB; videos average 2 MB in posts and other places; standard videos for streaming = 50 MB each minute of video

Example:

Social media: assuming 140 characters each tweet or post, 140*2 bytes = 280 bytes per post/tweet

The average URL length is less than 100 characters. More URLs that require a tiny URL will often be longer than 150 characters, perhaps 200 characters. Then each URL is assumed to be 200*2 = 400 bytes, with the database id, price, and other fields being double or long, for a total of 8 bytes per field.

- **Traffic Assumptions**

Large social media application: 1 billion total users, 500 million daily active users on social media (Facebook, Instagram, Twitter) or chat application (WhatsApp, Facebook Messenger).

500 million overall users, 200 million daily active users on a typical social media application

One billion overall users, 800 million daily active users on video streaming apps (YouTube, Netflix, Hulu).
Large – 1 billion total users, 500 million daily active users Normal – 500 million total users, 100 million daily active users Cloud or file storage apps (Google Drive, Dropbox, Microsoft OneDrive)

- **Time Assumptions**

The following are some time-related assumptions that have been estimated to make calculations without a calculator easier.

Seconds in a day = 24 hours 60 minutes 60 seconds = 86,400 seconds = 100,000 seconds (approximate)
365 days * 5 years = 1825 days = 2000 days (approximate) (Approximate)

Calculations

Occasionally, we must choose between two different architectures. We can start by asking if a single server is adequate. How many servers would we need if that wasn't the case? We can get an approximate estimate from a simple calculation. The calculation, in general, informs us whether the design can meet the functional requirements, such as the number of users supported, response latency, and resource needs. Rather than putting down a whole number, use tiny numbers with an abbreviation for magnitude (or, if necessary, exponents). Instead of 10,000, use 10,000. If you're given a huge and excessively exact figure, such as 7432, convert it to 7K right away. In any case, you're estimating. Multiplication and division are simplified when numbers are presented in this format. M is K*K. G/M equals K. 4K*7M=28G. Round both numbers to a modest multiple of a power of 2 or 10 to deal with bigger numbers.

27*14 = 30*10 = 300

6500/250 = 6400/256 = 100 * 2^6/2^8= 100 / 2^2 = 25

Social Media Calculations

Assumption

Assume the character limit for a post or tweet is 140 characters. For simplicity of computations, size = 140 char * 2 byte char = 280 bytes = Approximate to 300 bytes

Assume a total of 1 billion users, with 250 million daily active users. So 250 million people tweet once a day or once a week.

Assume that ten million users each upload a photo of 100 KB in size.

Total Storage per Day

300 bytes * 250,000,000 = 75,000,000,000 = 75 GB of text data each day

100 KB * 10,000,000 = 1,000 GB = 1 TB photo storage

10 year worth storage

The number of days in a ten-year period is about 4000 days

Text data of 75 GB per day * 2000 days = 150,000 = 150 TB

1 TB per day pictures multiplied by 4000 days equals 4,000 TB.

Bandwidth

In days, one second equals 100,000 seconds.

75GB per day / 100,000 seconds = 75,000,000,000 / 100,000 = 750,000 = 0.75 megabytes per second

1 TB per day / 100,000 seconds = 1,000,000,000,000 / 100,000 = 100,000,000 bytes per second = 100 MB per second

Cache

If each user has 20 posts/tweets, the cache will be 300 bytes * 500,000,000 daily users * 20 = 150 GB * 20 = 3000 GB = 3 TB.

If one system or server can store 150 GB of cache, caching will require 20 machines or servers.

Database Size Calculation

This is a single table example. You may make similar fast calculations for all tables based on your design to obtain a general estimate of database storage.

Example of User Table

User Id is an 8-byte double value.

User-name = 200 bytes (10 char * 2 bytes)

Email-id = 100 bytes (50 char * 2 bytes)

So each row is 8 + 200 + 100 = 308 bytes (about 300 bytes).

1 billion users * 300 bytes = 1,000,000,000 * 300 = 300,000,000,000 = 300 GB

2

FRAMEWORK FOR SYSTEM DESIGN INTERVIEW

Systems Design is one of the most sophisticated and significant topics for individuals looking to further their careers in software. It's crucial to understand the context of a systems design interview to comprehend the questions' nature. While interviews for the early phases of your software career will most likely focus on your coding skills and programming talents, as you advance in your career, you'll be expected to understand the ins and outs of creating sophisticated, dynamic, and scalable systems. This is one of the most critical aspects of the systems design interview. The vagueness and open-ended nature of the question presented is the most challenging aspect of dealing with systems design interviews. For many applicants, the absence of organization might be daunting.

Requirements

Functional, non-functional, and out-of-scope requirements are the three types of requirements. They'll be defined as part of the "Design Twitter Feed" assignment.

Functional Requirements

What capabilities does the system or program have to provide the user? For example, on Twitter, a user may follow another user, tweet, like a tweet, retweet another user's tweet, and share a tweet; however, they just focus on the fundamental functions of Twitter and do not dive into the more sophisticated capabilities. Consider 3–5 features that would add the most value to the company.

- Users should be able to browse indefinitely through a list of tweets.
- A tweet should be able to be liked.
- Users should be able to view comments on a tweet when they open it.

Non-Functional Requirements

Although not immediately apparent to the user, it plays an essential part in the overall design. They provide the following functions:

- Offline assistance is available.
- Notifications in real-time.
- Maximum bandwidth and CPU/Battery use.

The following are the essential ideas to consider for every distributed system:

- High availability: The majority of the systems must be available at all times.
- Consistency: High-availability systems will eventually be consistent. Because there can't be any data inconsistencies in any financial system, consistency takes precedence over availability. User data is not lost due to a lack of reliability.
- Latency is the time it takes for user activity, such as loading a web page or like a post, to be completed.

Out of Scope

Features that will be left out of the job yet are critical in a real-world project. Login/Authentication, Tweeting, Followers/Retweets, and Analytics are all available.

High-Level Design

When your interviewer is happy with the explanation phase and your system requirements selection, you should ask them if they want to view a high-level design.

Server- Side Components

Back-end: this class represents the whole server-side infrastructure. Your interviewer is unlikely to be interested in addressing it.

Push Provider: this class represents The Mobile Push Provider infrastructure. Delivers push payloads to clients after receiving them from the backend.

Content Delivery Network (CDN): Delivering static information to clients is your responsibility.

Client- Side Components

API Service: Client-server communication is separated from the rest of the system.

Persistence: There is only one source of truth. Data received by your system is first persisted on disk and then transmitted to other components.

Repository: A component that acts as a bridge between API Service and Persistence.

Tweet Feed Flow: Represents a group of components to produce an endless scrollable list of tweets.

Tweet Details Flow: Represents a set of components for showing the information of a single tweet.

DI Graph: Graph of dependency injection.

Image Loader: It's your job to load and cache static images. A third-party library is usually used to represent this.

Coordinator: Flow logic between the Tweet Feed and Tweet Details components is organized. Assists in the decoupling of system components from one another.

App Module: A system's executable component that "glues" parts together.

Providing the Signal

The interviewer may be on the lookout for the following cues:

- The applicant can explain the broad vision without going into too much detail about implementation.
- The candidate can recognize the system's major components and how they interact with one another.
- The applicant considers app modularity and can think in terms of the complete team rather than just themselves, which is especially essential for senior candidates.

Detailed Design

Following high-level design, your interviewer may direct the topic to a specific system component. Assume the source was Tweet Feed Flow. Topics you might wish to discuss include:

Architecture Patterns: MVP, MVVM, MVI, and so on. MVC is no longer seen to be a good option. Compared to other less well-known homegrown techniques, it's better to use a famous pattern since it's simpler to onboard new personnel.

Pagination: It is necessary for endless scroll functionality.

Dependency Injection: The ability to construct an isolated and tested module

Image Loading: Image loading performance (low-resolution vs. full-resolution), scrolling performance, and so on.

Components

Feed API service: Abbreviations Client for the Twitter Feed API allows you to request paginated data from the backend. DI-graph was used to inject the substance.

Feed Persistence: Abstract paginated data storage cached DI-graph was used to inject the substance.

Remote Mediator: Causes the next/previous page of data to be retrieved. The freshly acquired paged response is redirected to a persistence layer.

Feed Repository: Uses Remote Mediator to combine remote and cached answers into a Pager object.

Pager: Data is fetched from the Remote Mediator and exposed to the UI as an observable stream of paged data.

Tweet Like and Tweet Details: Delegate the "Like" and "Show Details" actions. DI-graph was used to inject the substance.

Image Loader: Abstracts the image loading library's image loading. DI-graph was used to inject the substance.

Providing the Signal

The interviewer may be on the lookout for the following signals:

- The applicant is well-versed in the majority of MVx patterns.
- The candidate establishes a distinct separation of business logic and user interface.
- The applicant is well-versed in dependency injection techniques.

- The applicant can create self-contained, independent modules.

Bottlenecks

To manage user requests, your system may require a load balancer and several computers. Perhaps the data is so large that you need to spread it over many computers.

- What are some of the negative consequences of doing so?
- Is the database running slowly and in need of some in-memory caching?

Given the restrictions of the problem, your high-level design will almost certainly contain one or more bottlenecks. This is entirely OK. You are not expected to build a system from the ground up that can manage all of the world's load right now. It only has to be scalable so that you can develop it with standard tools and approaches. Start thinking about the bottlenecks in your high-level design now that you have it. Perhaps your system requires a load balancer and a large number of computers to handle user requests. Maybe the data is so large that you need to spread it over many computers. What are some of the negative consequences of doing so? Is the database running slowly and in need of some in-memory caching? These are just a few samples of questions you could be asked to finish your solution. The interviewer may try to steer the conversation in a specific direction. Then you might not need to fix all of the bottlenecks but instead, focus on one in particular. In any scenario, you must be able to detect and address a system's flaws. Remember that every solution is usually a trade-off of some sort. Something will deteriorate if you change it. Given the restrictions and use cases stated, the critical issue is to discuss and quantify these trade-offs in terms of the system's effectiveness. After you've identified the main bottlenecks, you may go on to the next stage of resolving them.

Trade-Off

Almost every decision will include a compromise. Articulating them in real-time as you propose solutions demonstrates that you realize that complex systems frequently necessitate compromises and allows you to show your understanding of the benefits and drawbacks of various methods. Because there is no one-size-fits-all solution, having this conversation will give your interviewer the idea that you are pragmatic and utilize the best tool for the task.

Consider the following questions :

- What type of database would you choose and why?
- What options do you have for caching? Which one would you pick and why?
- What infrastructure frameworks can we utilize in your ecosystem of choice?

4

ID GENERATOR SYSTEM DESIGN

Introduction

Many distributed services now require unique global identifiers. Social security numbers, Aadhar numbers, personal account numbers, bank account numbers, tweet IDS, picture IDS, and file IDs are examples of unique identifiers in a service. This list may be expanded to include a wide range of organizations, and almost all services utilize some sort of unique identifier in their systems. This widespread use necessitates a consideration of several methods for obtaining unique global IDs within a system. If you're creating a social network product like Twitter, you'll need to assign a user ID to each user in your database. This ID will be unique and will be used to identify each person in the system. You may just keep incrementing the ID from 1, 2, 3,... N in specific systems. We may need to generate the ID as a random string in other systems.

Requirements

ID generators must meet a few requirements :

- They can't be any length. Let's assume we limit ourselves to 64 bits.
- The date is used to increase the ID. This provides a lot of versatility to the system; for example, you may arrange people by ID, which is the same as sorting by registration date.

High-Level Design

Distributed Sequence Number Generation

How do you get a unique ID in a distributed setting while keeping scalability in mind? In addition, the following features are required:

- Probabilistic methods are ruled out since the ID must be assumed to be globally unique.
- The client will utilize the issued ID for an indefinite length of time, which implies that once assigned, the ID will be lost forever and will not be reused for future assignments.
- Because the ID is short, say 64 bits, compared to the machine's unique identifier, the idea of creating a unique ID using a combination of MAC address + Process id + Thread id + Timestamp is ruled out.

Architectural goals are:

- The solution must be scalable as the number of requests increases.
- The solution must be able to withstand component failure.

Using a generic distributed computing approach, a pool of ID generators dubbed workers will be located on many interconnected, inexpensive commodity computers. Depending on the application, the client requesting a unique ID may sit on the same system as the worker or a different machine. In the latter instance, a load balancer will be placed between the client and the worker to ensure that the client's burden is divided equally. One simple, if naïve, solution is to have the worker, i.e. the ID generator, send a request to a centralized book-keeping server that keeps track of the counter. The central bookkeeper may be both a performance bottleneck and a single point of failure. The speed bottleneck can be alleviated by requesting a "number range" from the centralized bookkeeper rather than the "ID" itself. In this scenario, ID assignment will be done locally by the worker inside this number range, and the bookkeeper will only be notified once the entire range has been exhausted. When the bookkeeper receives a number range request, it will save the allotted number range to disk before returning to the client. As a result, if the bookkeeper crashes, it will know where to begin allocating the number range following a reboot. Mirrored disks should be utilized to avoid the disk becoming a SPOF. The bookkeeper's SPOF problem can be solved by having two bookkeepers (primary and backup). The primary bookkeeper must use shared disks or counterchange replication to synchronize its counter state with the standby. The standby will keep an eye on the primary bookkeeper's health and take over if it fails.

Each worker will construct a "DHT Node" with a randomly-generated 64-bit node id and join a "DHT ring" utilizing DHT's "key-based-routing" paradigm. This scheme implicitly allocates the number range between the node id and its immediate neighbor's node id. We can now use some valuable features of the DHT architecture, such as the ability to employ a large number of user resources for workers with O (logN) routing hobs. Furthermore, DHT nodes include duplicated data from their neighbors, ensuring that if one DHT node fails, its neighbor will quickly take over its number range. What happens if a DHT node's implicitly assigned number range is depleted? When this happens, the worker will create a new DHT node, which will join the ring at a later time and receive a new number range.

UUID

UUIDs are globally unique 128-bit hexadecimal numbers. It's doubtful that the identical UUID will be produced again. The issue with UUIDs is that they are pretty large and complex to index. As your dataset grows, so does the index size, and query speed suffers as a result. Another issue with UUIDs is how they affect the user's experience. Our users will eventually require unique IDs. Consider a scenario in which a client contacts Customer Service and is asked to supply the identification. It's not fun to have to spell out a UUID in its entirety.

Pros

- It is entirely independent. It works with the current timestamp, process ID, and MAC address and scales well for sharded databases.
- There are very few chances of collisions, and the same UUID is generated twice.
- The IDs remain time-sortable if the timestamp is the first component of the ID.

Cons

- To ensure adequate uniqueness assurances, UUIDs require more capacity (128 bits).
- It does not index properly due to its large size. As our dataset grows, so does the index size, and query speed suffers.
- Some UUID types have no natural order and are random.
- Some UUID package types have long been leaky because they reveal timestamp and mac address information. Although v5 UUIDs are produced by hashing and discarding certain bits, we have no hope of retrieving any information from them.

MongoDB Object IDS

MongoDB Object Ids are 12-byte (96-bit) hexadecimal integers that begin with a random value and consist of a 4-byte epoch timestamp in seconds, a 3-byte machine identification, a 2-byte process id, and a 3-byte counter. This is a smaller UUID than the previous 128-bit version. However, the size is more than we would typically find in a single MySQL auto-increment column (a 64-bit digit value).

Pros

- Each application thread creates IDs individually, reducing ID creation failure spots and contention. The IDs remain time-sortable if the first component of the ID is a timestamp.
- To establish adequate uniqueness assurances, additional storage space (96 bits or more) is usually required.
- Some UUID types have no natural order and are random.

Detailed Design

Ticket Servers

This is one of the most well-known techniques, in which you just keep a table with only the most recent produced ID in it, and whenever a node requests an ID, you simply do a ‘select for update' on this table, increase the value, and use the selected value as the next ID. The essence of this method is that it is robust and dispersed. The ID creation process can be isolated from the data storage process. However, because all nodes rely on this table for the next ID, there is a danger of Single Point of Failure, and if this service goes down, your app may cease working correctly. For further robustness, MySQL shards are constructed as master replicant pairs. To avoid critical collisions, we need to be able to ensure uniqueness inside a shard. This method generates unique incrementing IDS using a centralized database server. In basic terms, a database that retains the most recent produced ID and increases the values provided back as a new ID every time a node requests one.

Pros

- It performs admirably in a sharded database and at scale.
- Short length, good indexing, and does not degrade query speed while dealing with massive datasets.

Cons

- Because all nodes rely on this table for the next Id, there is a single point of failure.
- Ticketing servers can become a write bottleneck at scale because they may not be adequate when the number of writes per second is enormous, causing the ticketing server to overload and decrease performance.
- Extra computers are required for ticketing servers.
- There is a single point of failure if we use a single database, and we can't ensure that ids will be sortable over time if we use many databases.

Twitter Snowflakes

This is a network service devoted to creating 64-bit unique IDs at a large scale. Twitter made it to create unique identifiers for tweets, direct messages, lists etc. These unique identifiers are 64-bit unsigned numbers that are time-based rather than sequential. It is made up of the following components:

We get 69 years with a millisecond precision epoch timestamp with a custom epoch, i.e. 41 bits gives us 69 years. We can have up to 1024 machines with a configured machine ID of 10 bits. The sequence number is 12 bits, and each computer has a local counter that rolls over every 4096 seconds. The extra bit will be used in the future. For the time being, this bit is always set to 0. They are time sortable since the timestamp is the first component. This is the finest option I could discover that satisfies all of my needs. This allows our microservices to generate IDs on their own.

Pros

- Snowflake IDs are 64 bits long, which is half the length of a UUID.
- Time may be used as the first component and still be sorted.
- A distributed system that can withstand the loss of nodes.

Cons

- A zookeeper is required by this design to maintain the mapping of Nodes and Machine Ids.
- It also necessitates several Snowflake servers, which adds to the complexity and number of moving components.

5

WEB CRAWLER SYSTEM DESIGN

Introduction

A web crawler, often known as a spider or Spiderbot, is an internet bot that explores web pages for the sole purpose of indexing. Crawling is generally performed by multiple machines in a distributed web crawler. Google's web crawler, which crawls all web pages on the internet, is one of the most well known distributed web crawlers.

Requirements

Functional Requirements

- Crawl only HTML pages
- Crawl only HTTP pages
- The crawler must be courteous.
- Seed URLs should be accepted.
- The pages that were crawled should be preserved.

Non-Functional Requirements

- The crawler should not crash while doing its duties.
- The latency of the service should be kept to a minimum.

High-Level Design

Design Components and Considerations

We go through the fundamental design elements of any crawler and the key design concepts that must be considered for distributed and decentralized crawlers. A typical web crawler is made up of three main parts:

Downloader: This component receives a list of URLs and sends HTTP requests to retrieve the requested web pages.

URL Extractor: This component is in charge of extracting URLs from a web page that has been downloaded. The URL is subsequently added to the list of URLs to be crawled.

Duplicate Tester: This component is in charge of looking for duplicate URLs and content.

It also keeps the following data structures for the crawl:

Crawl-Jobs: It's a list of URLs that will be crawled.

Seen-URL: It's a list of URLs that have been crawled before.

Seen-Content: It's a list of pages that have been crawled and their fingerprints (hash signature/checksum).

Seed URLs

Web crawlers use seed URLs as a beginning point for their crawls. An obvious approach to choose seed URLs for crawling all web pages from a university's website, for example, is to utilize the university's domain name. We need to be inventive when choosing seed URLs if we want to crawl the entire web. A decent seed URL can be a starting point for a crawler to visit as many links as feasible. The main idea is to break up the URL space into smaller chunks. Because various nations may have different popular websites, the first recommended strategy is based on location. Another option is to select seed URLs based on subjects; for example, we may split

URL space into commerce, sports, and healthcare categories. The choice of a seed URL is an open-ended question.

Any Web crawler's fundamental technique is to accept a list of seed URLs as input and perform the following steps repeatedly.

- Choose a URL from the list of unvisited URLs.
- Determine the host-IP name's address.
- To download the appropriate document, establish a connection to the host.
- Look for new URLs by parsing the text of the page.
- Add the new URLs to the list of URLs that haven't been visited yet.
- Process the downloaded document, for example, by storing or indexing its contents, and so on.

Difficulties Implementing Efficient Web Crawler

Web crawling is exceptionally challenging due to two key features of the Internet:

- Large Volume of Web Pages: Because a web crawler can only download a portion of the online pages at any given moment due to the significant number of web pages, the web crawler must be intelligent enough to prioritize downloads.
- Rate of Change on Web Pages: Another issue in today's dynamic environment is that websites on the internet are constantly changing. As a result, when the crawler downloads a site's last page, the page may have changed, or a new page may have been added.

At the very least, a crawler must include the following components:

URL Frontier: To track the URLs to download and prioritize which ones should be crawled first.

HTTP Fetcher: To obtain a web page from a server.

Extractor: To extract hyperlinks from HTML files.

Duplicate Eliminator: To ensure that the identical material isn't accidentally extracted again.

Data Store: To keep track of retrieved pages, URLs, and other information.

Detailed Design

Detecting Update and Duplicate

Assume that our crawler is operating on a single server. All of the crawling is handled by numerous worker threads, each of which executes all of the processes required to download and analyze a document in a loop.

The initial step in this loop is to remove an absolute URL from the download shared URL boundary. An absolute URL starts with a scheme (for example, "HTTP") that specifies the network protocol to employ while downloading it. We can build these protocols in a modular fashion for flexibility so that our crawler can easily handle other protocols in the future. The worker contacts the relevant protocol module to download the content based on the URL scheme. The document is saved to a Document Input Stream (DIS) when it has been downloaded. By storing documents in DIS, other modules will be able to read them numerous times. The worker thread does the dedupe test after writing the document to the DIS to examine if this document (associated with a different URL) has been viewed previously. If this is the case, the document is not further processed, and the worker thread deletes the next URL from the frontier.

Our crawler must then process the downloaded document. Each document can have its MIME type, such as HTML page, Image, Video, etc. We can build these MIME schemes in a modular fashion so that if our crawler needs to handle other kinds in the future, we can easily add them. The worker calls the process method of each processing module associated with the MIME type of the downloaded document based on that MIME type. In addition, our HTML processing program will extract all of the page's links. To decide if a link should be downloaded, it is transformed to an absolute URL and checked against a user-supplied URL filter. Whether the URL passes the filter, the worker runs the URL-seen test to see if it's been seen previously, that is, if it's in the URL frontier or if it's been downloaded. The URL is added to the border if it is new.

URL Frontier

The data structure that holds all of the URLs that need to be downloaded is the URL frontier. We can crawl by traversing the Web in a breadth-first manner, beginning with the sites in the seed set. Using a FIFO queue, such traversals are simple to construct. We may divide our URL boundary among several servers because we'll be crawling a large number of URLs. Assume many worker threads are executing the crawling duties on each server. Let's also suppose that our hash function associates each URL with a server that will crawl it. When building a distributed URL border, keep the following politeness criteria in mind:

- By downloading a large number of pages from a server, our crawler should not overburden it.

- A webserver should not be connected to numerous computers.

Our crawler can have a set of separate FIFO sub-queues on each server to achieve this politeness restriction. Each worker thread will have its sub-queue from which URLs for crawling will be removed. When a new URL has to be added, the URL's canonical hostname determines which FIFO sub-queue it goes into. Each hostname may be mapped to a thread number using our hash function. These two arguments indicate that just one worker thread will download content from a particular Web server and that the Web server will not be overloaded because of the FIFO queue. The number of URLs would be in the hundreds of millions. As a result, we'll need to save our URLs to disk. We may design our queues so that enquiring and de-queuing buffers are distinct. The enquire buffer will be emptied to disk once it is full, but the dequeue buffer will retain a cache of URLs that need to be visited and read from the disk to fill the buffer regularly.

Fetcher Module

A fetcher module's job is to use the proper network protocol, such as HTTP, to download the document associated with a specified URL. Our crawler's HTTP protocol module can retain a fixed-sized cache mapping host-names to their robot's exclusion criteria to avoid downloading this file on every request.

Document Input Stream (DIS)

The architecture of our crawler allows numerous processing modules to process the same page. To avoid downloading a document innumerable times, we use a Document Input Stream abstraction to cache the document locally. A DIS is a type of input stream that saves the complete contents of a document that has been downloaded from the internet. It also has options for re-reading the document. Small documents (64 KB or less) can be cached fully in memory by the DIS, whereas more extensive documents can be temporarily written to a backup file. Each worker thread has its DIS, which it reuses from one document to the next. The worker transfers the URL extracted from the border to the appropriate protocol module, which initializes the DIS from a network connection to include the document's contents. The worker then sends the DIS to all processing units that need it.

Document Dedupe Test

Many documents on the Internet can be found at several URLs. There are also numerous instances where documents are replicated over several servers. Any Web crawler will download the same content many times as a result of both of these effects. We run a dedupe test on each document to eliminate duplication and prevent it from being processed twice. We may compute a 64-bit checksum of each processed document and store it in a database to perform this test. We can verify the checksum of each new document against all previously calculated checksums to determine if it has been seen before. These checksums may be calculated using MD5 or SHA. If the entire point of our checksum store is to do dedupe, we only need to retain a single set of checksums for all previously processed documents. To accommodate 15 billion unique web pages, we'd need:

15B * 8 bytes = 120GB

Although this can fit within the RAM of a modern-day server, if we don't have enough, we can retain a smaller LRU-based cache on each server, with everything backed up by persistent storage. The checksum must be present in the cache for the dedupe test to succeed. If not, it must determine if the checksum is stored in the back storage. If the checksum is detected, the document will be ignored. It will be added to the cache and back storage if not.

URL Filters

The URL filtering method allows you to customize the list of URLs that are downloaded. This is used to add websites to a blacklist so that our crawler would ignore them. The worker thread examines the user-supplied URL filter before adding each URL to the frontier. Filters can be used to limit URLs by domain, prefix, or protocol type.

Domain Name Resolution

A Web crawler must utilize the Domain Name Service (DNS) to translate the Web server's hostname into an IP address before accessing it. Given the number of URLs, we'll be working with, DNS name resolution will be a major barrier for our crawlers. By constructing a local DNS server, we may begin caching DNS results to avoid repeated queries.

URL Dedupe Test

Any Web crawler will come across many links to the same content while extracting links. To prevent downloading and processing a document numerous times, each extracted link must pass a URL dedupe test before being included in the URL frontier. We may save all of the URLs encountered by our crawler in canonical form in a database to conduct the URL dedupe test. We keep a fixed-sized checksum instead of the textual representation of each URL in the URL collection to conserve space. We may retain an in-memory cache of popular URLs on each host shared by all threads to decrease the number of operations on the database store. The purpose of this cache is because specific URLs include a lot of links. Thus caching the most popular ones in memory will result in a high hit rate.

Bloom Filters

Bloom filters are a probabilistic data structure that can produce false positives when used for set membership checking. A significant bit vector represents the set by computing the element's hash functions and setting the appropriate bits. If the bits at all ‘n' of the element's hash locations are set, the element is considered in the set. As a result, a document may be mistakenly identified as part of the collection, but false negatives are not conceivable. The downside of employing a bloom filter for the URL seen test is that each false positive causes the URL to be removed from the frontier, preventing the document from being downloaded. Making the bit vector bigger reduces the probability of a false positive.

Check Pointing

It takes weeks to crawl the entire Internet. Our crawler may write frequent snapshots of its state to the disk to protect against failures. A crawl that has been halted or aborted can simply be resumed from the most recent checkpoint.

BFS VS DFS

The most common search method is breadth-first search (BFS). However, Depth First Search (DFS) is also used in some instances, such as when your crawler has already established a connection with a website and wants to save time by DFSing all URLs on that page.

Path Ascending Crawling

Crawling in a path-ascending manner can help you find some isolated resources or resources for which no inbound connection would have been detected in a typical scan of a Web site. A crawler would rise to every path in each URL is intended to crawl in this scheme. If provided a seed URL of http://foo.com/a/b/page.html, for example, it will try to crawl /a/b/, /a/, and /a/b/.

Fault Tolerance

For distribution among crawling servers, we should utilize consistent hashing. Consistent hashing will aid in the replacement of a dead host and the distribution of load across crawling servers. All of our crawling servers will checkpoint and save their FIFO queues to disks regularly. We can replace a server if it goes down. Meanwhile, constant hashing should help to distribute the load among the servers.

Data Partitioning

Our crawler will deal with three different types of data :

- Web addresses to check out.
- Dedupe URL checksums.
- Dedupe checksums should be documented.

We can keep this data on the same server because we're distributing URLs depending on hostnames. As a result, each host will keep track of the URLs that must be visited, as well as the checksums of all previously visited URLs and the checksums of all downloaded contents. We may presume that URLs will be redistributed from overloaded hosts because we'll be utilizing consistent hashing. Periodically, each host will conduct check pointing and dump a snapshot of all the data on a distant server. This ensures that if a server fails, another server may take its data from the previous snapshot and replace it.

Crawler Traps

Crawler traps, spam sites, and disguised material abound. A crawler trap is a URL or a group of URLs that causes a crawler to continue crawling indefinitely. Unintentional crawler traps exist. A cycle can be created via a symbolic link within a file system, for example. Other crawler traps are placed on purpose. People have created traps that dynamically generate an endless Web of documents, for example. The motives behind such traps differ. Anti-spam traps are meant to catch crawlers employed by spammers seeking email addresses, while traps catch search engine crawlers on other sites to improve their search rankings.

Bottlenecks

By including backup servers in the architecture, you may enhance redundancy. Due to data sharding, we'd also have several distributed databases. As a result, we'll require servers to aggregate data from various shards. The application servers will be linked to these aggregator servers. For traffic distribution, load balancers must be included in the architecture. For scalability purposes, we'll include a content distribution network (CDN) in the architecture. A content delivery network (CDN) is a geographically dispersed set of servers that collaborate to deliver Internet material quickly. A CDN provides the rapid delivery of materials such as HTML pages, JavaScript files, stylesheets, pictures, and videos required for loading Internet content. CDN services are becoming increasingly popular, and they now handle the bulk of online traffic, including traffic from big sites like Facebook, Netflix, and Amazon.

Trade-Off

Because the Lambda page collects the tax continually, it will ultimately be the page with the most credit and will be crawled. We just take the credits from the Lambda page and spread them evenly over all of the pages in our database after crawling it. Because bot catchers only provide credit to internal links and seldom acquire credit from outside sources, credits (from taxes) will continue to seep to the Lambda page. The Lambda page will evenly distribute those credits to all of the database pages, and the bot trap page will lose more and more credits with each cycle until it has so few credits that it is practically never crawled again. This is unlikely to happen with good pages, as they frequently receive credit from hyperlinks on other sites.

6

DISTRIBUTED KEY-VALUE STORE SYSTEM DESIGN

Introduction

A key-value store is a powerful method that can be found in nearly any system on the planet. It may be as basic as a hash table while also serving as a distributed storage system. For example, Cassandra's core system is a key-value storage system, and organizations like Apple and Facebook extensively utilize Cassandra.

Requirements

Set a value for a key, and if the value already exists, update it.

- Get the value that the key specifies.
- Remove the key value from the equation.
- It is highly scalable because it should be able to scale up instances in real-time as demand grows.
- On each call, it should return a consistent and proper response.
- Durable: During network partition failures, no data should be lost.
- Accessibility. It's a CP, according to the CAP theorem, which means consistency takes precedence above availability.

High-Level Design

Basic Key-Value Storage

The simplest straightforward solution is to store key-value pairs in a hash table, which is how most modern systems function. A hash table is a type of data structure that allows you to read and write a key-value combination in real-time and is incredibly simple to use. This is something that most languages have built-in support for. However, there is a disadvantage. When using a hash table, you must generally store everything in memory, which is not always practical when the data set is large. Two options are commonly used:

- Compress your information. This should be the first item to consider, and there is usually a lot you can condense. You can, for example, save a reference rather than the actual data. Instead of float64, you may use float32. Furthermore, various data formats such as bit arrays (integer) or vectors might be helpful.
- Putting data on a disk. You can save a portion of the data to disk if it's difficult to fit everything in memory. You may conceive of the system as a caching system to further optimize it. The often accessed data is maintained in memory, while the remainder is stored on a disk.

Because a single computer cannot store all of the data, the basic concept is to divide the data among many machines according to some criteria, with a coordinator machine directing clients to the unit with the required resource. The topic is how to divide data among several machines and, more critically, a suitable data partitioning technique.

Sharding

Assume that all of the keys are URLs and that there are 26 computers. One method is to divide all keys (URLs) into these 26 machines depending on the first character of the URL (after "www"). Let's disregard URLs that include ASCII characters. A decent sharding algorithm should be capable of evenly distributing traffic across all computers. In other words, each computer should ideally get the same number of requests. The design above appears to be ineffective. To begin with, the storage is not spread evenly. There are probably a lot more URLs that begin with "a" than with "z." Second, specific URLs, such as Facebook and Google, are significantly more popular. You should make sure that keys are given randomly to balance traffic. Another option is to utilize the hash of the URL, which is generally considerably faster. To create a decent sharding method, you must first understand the application and predict the system's bottleneck.

Data Partitioning

Data must be partitioned among a cluster of servers, with no one server holding the entire data set. Even when the data fits on a single disk, seek time dominates disk access for tiny values. Therefore partitioning improves cache performance by separating the "hot" set of data into smaller pieces that may (ideally) fit entirely in memory on the server that holds that partition. This means that the cluster's servers aren't interchangeable, and requests must be directed to a server with the desired data rather than to any accessible server at random. Similarly, servers fail, get overloaded, or are taken offline for maintenance regularly. If there are S servers and each server fails with a probability of p in a given day, the chance of losing at least one server in a given day is 1 - (1 - p)s. Given this, we obviously cannot keep data on a single server, else the risk of data loss will be inversely related to cluster size.

The most straightforward method would be to cut the data into S partitions (one per server) and store copies of a given key K on R servers. Taking a = K mod S and storing the value on servers a, a+1,..., a+r is one approach to associate the R servers with key K. So you may choose a suitable replication factor R for any probability p to attain an acceptable low chance of data loss.

This method has the wonderful characteristic of allowing anybody to determine the location of a value just by knowing its key, allowing us to conduct peer-to-peer lookups without contacting a central metadata server with a mapping of all keys to servers. The technique above has a disadvantage when a server is added or withdrawn from the cluster, for example, because we have acquired new hardware or a server is temporarily offline. In this instance, d may change, causing all data to be sent across servers. Even if d does not change, the load will not be distributed equally from a single removed/failed server throughout the cluster.

Replica

One important statistic to consider when evaluating a distributed system is system availability. What happens if one of our computers fails for whatever reason, such as a hardware failure or a software bug? How does this affect our key-value storage system? We won't deliver the right response if someone requests resources from this computer, it appears. When creating a side project, you might not think about this issue. However, if you have a large number of servers servicing millions of people, this happens frequently, and you can't afford to restart the server every time manually. This is why, in today's distributed systems, availability is critical. So, how would you approach this problem? Of course, test cases may help you create more resilient code. Your software, on the other hand, will always contain defects.

Furthermore, hardware concerns are considerably more challenging to safeguard. Replica is the most frequent solution. We can drastically decrease system downtime by establishing up computers with redundant resources. When a single computer has a 10% risk of crashing every month, having a single backup machine reduces the likelihood to 1% when both are down.

Replica Vs. Sharding

First and foremost, we must understand the objective of these two approaches. Because a single system can only hold so much data, sharding divides data over many machines. Replica is a method of preventing system downtime. With that in mind, replica won't assist if a single computer can't store all the data.

Distributed Key-Value Store

A distributed key-value store expands on the benefits and uses cases outlined above by delivering them at scale. Because more servers with greater memory now house the data. A distributed key-value store is designed to function on many computers working together, allowing you to work with more extensive data sets. You can improve processing performance by spreading the storage over many servers. You may enhance the fault tolerance of your distributed key-value store by using replication. For larger-scale deployments, Hazelcast is an example of a system that provides a distributed key-value store. Hazelcast's "IMap" data type is a memory-based key-value store comparable to Java's "Map" type. Unlike Java Maps, Hazelcast IMaps are spread in memory over a cluster of machines' aggregate RAM, allowing you to store considerably more data than is feasible on a single computer. This will enable you to do in-memory lookups while maintaining other essential features like high availability and security.

Detailed Design

Consistency

We may strengthen the system by introducing copies. However, there is a problem with consistency. Let's suppose we have replica A2 for machine A1. How can you know A1 and A2 have the same information? When adding a new entry, for example, we must update both machines. However, one of them may fail the write operation. As a result, A1 and A2 may accumulate a significant amount of conflicting data over time, an important issue. There are a few options available here. The first strategy is to retain a local copy in the coordinator. When a resource is changed, the coordinator keeps a copy of the new version. If the update fails, the coordinator can retry the procedure. Another option is to use a commit log. If you've ever used Git, you're probably familiar with the idea of a commit log. Essentially, each node computer will retain a commit log for each activity, which tracks all changes. So, if we wish to update an entry on machine A, we'll have to go through the commit log first. Then, in sequence, a separate software will process all commit logs (in a queue). We can simply recover if an operation fails since we can check the commit log. Finally, I'd want to discuss how to settle the disagreement in a read. If the desired resource is found on machines A1, A2, and A3, the coordinator can request it from all three computers. If the data is different, the system will automatically resolve the disagreement. It's important to remember that none of these strategies is mutually exclusive. Depending on the application, you may want to utilize more than one.

Versioning in a Distributed System

A basic versioning system is optimistic locking. Each piece of data is assigned a unique counter or "clock" value, and changes are only permitted provided the update specifies the correct clock value. This works fine in a centralized database, but it fails in a distributed system where computers come and go, and replication might take long. For our application, a single variable will not have enough write history to discard older versions. The last two writes in this model render the initial value meaningless (since they happen after the original). However, no rule tells the server that either the email or the name change can be discarded. So we need a versioning system that can identify overwrites and discard the previous version and detect conflicts and allow the client to resolve them. A so-called vector-clock version is one solution. A vector clock keeps track of each writing server's counter, allowing us to determine when two versions conflict and when one succeeds or precedes the other. A list of servers makes up a vector clock: [1:45,2:3,5:55]

For that amount of writes, the version indicates that the server was the "master." If for all I v1i > v2i, a version v1 succeeds a version v2. If neither v1 > v2 nor v1 v2, v1 and v2 co-exist and are at odds. Here's a basic example of two versions that are at odds:

[1:2,2:1]

[1:1,2:2]

So, whereas simple, optimistic locking systems provide a complete order, our versioning technique defines a partial order over values.

Consistency Vs. Versioning

Consistency becomes a challenge when numerous simultaneous writes are dispersed over different servers (and maybe various data centers). Distributed transactions are the conventional answer to this problem. Still, they are both sluggish (multiple round trips) and unstable (since they require all servers to be online to complete a transaction). Any method that needs to communicate with more than 50% of servers to guarantee consistency becomes particularly troublesome if the application is deployed across several data centers. The latency for cross-data-center operations would be pretty high. Tolerate the potential of inconsistency and fix discrepancies at read time is an alternative option. That is the strategy used in this case. When changing data, most applications follow a read-modify-update pattern. We may load the user object, add the email address, and then write the updated values back to the database if a user adds an email address to their account.

Database transactions are a solution to this problem, but they aren't a viable choice when the transaction spans many page loads that may or may not complete and which might finish at any time. If all reads of a particular key return the same value in the absence of updates, the value for that key is consistent. Data is produced consistently and is not altered in a read-only environment. When we combine writes with replication, we have issues: we now have to update various variables on different computers while maintaining consistency. This is exceedingly difficult in server failures, and it is probably impossible in the presence of network partitions (a partition occurs when, for example, A and B can contact each other, but A and B can't reach C and D). There are numerous techniques for achieving consistency, each with its own set of guarantees and tradeoffs.

Two-Phase Commit: This is a locking procedure involving two rounds of machine coordination. It's completely constant, but it's also incredibly sluggish and intolerant of failure.

Paxos-style consensus: This is a mechanism for reaching a more failure-tolerant agreement on a value.

Read-repair: The first two methods prevent inconsistency from becoming persistent. This method entails writing all incompatible versions, then identifying the disagreement and fixing the issues at read-time. This requires little coordination and is entirely fault-tolerant, although additional application logic to resolve disputes may be required.

We utilize read-repair and versioning. This offers the highest efficiency and the best availability assurances (only W writes network roundtrips are required for N replicas where W can be configured to be less than N). In most cases, 2PC necessitates 2N blocking roundtrips. Paxos variants vary a lot, but they're all equivalent to 2PC. Another method for achieving consistency is to use Hinted Handoff. If the destination nodes are down during writes, we keep a "hint" of the changed data on one of the living nodes in this procedure. The "hints" are then pushed to these down nodes as they come back up, ensuring that the data is constant.

Trade-Off

The compromises involved in developing distributed database systems are numerous, and neither CAP nor PACELC can fully describe all of them. However, incorporating the consistency/latency tradeoff into current DDBS design considerations is significant enough to justify bringing the tradeoff closer to the forefront of architectural debates. There has been work on mathematically defining the tradeoff between latency and strong consistency models, in addition to the Weak CAP Principle and PACELC. Their consistency models aren't the same as ours, and they didn't consider the tradeoff between consistency and availability. The systems that are most relevant to our work fall into two categories. The first group of systems is concerned with metrics for determining whether data is fresh or stale. We do not compare our work to this class of systems since our objective is not to provide yet another consistency model or measure. For quorum-based storage, he suggested a probabilistic consistency model (PBS). However, he ignored latency, soft partitions, and the CAP theorem. -atomicity, a time-based staleness metric, was suggested. The gold standard for detecting atomicity breaches (staleness) over many read and write operations is -atomicity.

7

NOTIFICATION SERVICE DESIGN

Introduction

The act of bringing something to the user's attention is known as notification. An app can use a notification to alert you or send you a message that you can read without launching the app. An email alert is a simple form of a notice. When you get an email, a flash message appears on your smartphone screen. You want to launch the app immediately from the home screen. You may also swipe the notice across to dismiss it. The notification's primary purpose, however, is to notify the arrival of an email. To check your emails in typical situations, you must first open the email. The notice allows you to obtain a quick overview of the situation without opening the email client. Users have varied feelings about notifications. They frequently find it beneficial. They are commonly irritated by it. Notifications, on the other hand, have a function. They help alert users about app problems, introduce new features and upgrades, and inform them of pals' new messages and emails. They aid in connection with users who have abandoned apps and increase engagement from a marketing standpoint. So, how do you make your notification more purposeful and valuable?

Requirements

- Assume we're creating a social network similar to Facebook. To add asynchronous behavior to our program, we'd utilize a message queue.
- A user will have a large number of followers and friends. One person can have a lot of friends and be friends with a lot of

people.

- We must store a post created by a user in the database. As a result, we'll need a User table and a Post table. Because a single user can create many posts, the user and posts tables will have a one-to-many connection.
- We must display the user's post on his friends' home page while saving the post in the database. This new post necessitates sending a notification to our friends.

High-Level Design

Types of Notification

- **Users Generated Notifications**

This is the most prevalent and exciting form of an alert. The most basic example of this sort of notification is mobile texting. It is addressed to a specific user. Other basic instances of these notifications are social media postings, likes, and comments.

- **Context Generated Notifications**

This is another popular form of notification in which the application generates a notice depending on the users' permission. The finest examples are location-based alerts. This category also includes a lot of sports and meeting updates.

- **System Generated Notifications**

These are app-generated alerts based on the app's requirements. A security alert demanding a password reset is an example of such a message.

- **Push Notifications**

In reality, all types of alerts may be classified as push notifications for the sole reason that the system pushes through them. There are two sorts of push notifications. The first demands urgent action, whereas the second is passive notice.

Frontend

A Reverse Proxy is the first component to receive a request when it arrives on the host. It is a lightweight server that performs the following functions:

SSL termination: It occurs when HTTPS requests are decrypted and forwarded on in an unencrypted form. At the same time, the proxy is in charge of encrypting replies before returning them to clients.

Compression (gzip): It occurs when answers are compressed before being returned to clients. As a result, the amount of bandwidth required for data transport is reduced. This functionality isn't helpful for notification services because, in our instance, answers are small and primarily serve as confirmation that a request was performed correctly. We must also produce metrics. This is merely a set of key-value data that can be aggregated and used to track service health and compile statistics. We'll need all of this information for system monitoring, for example, the number of requests, errors, and call latency. We may also need to write data that may be used for auditing, such as a log of who and when made calls to a specific API in the system. It's crucial to realize that the Frontend service is in charge of writing log data. However, other components, commonly referred to as agents, are in charge of log data processing. Agents handle data aggregation and log transmission to other systems for post-processing and storage. This division of duties is what allows Frontend services to be simpler, quicker, and more reliable.

Temporary Storage

Messages are sent to the Temporary Storage service after being processed by the Frontend service. Why is it called temporary storage rather than just storage? Because communications are only meant to be stored here for a limited time. Unless the subject is designed to transmit messages with a delay, the faster we can deliver messages to subscribers, the better. What can we anticipate from the service of Temporary Storage?

- It must, first and foremost, be quick, highly accessible, and scalable.
- Second, it must ensure data persistence, ensuring that messages survive a subscriber's unavailability. And it can be re-delivered at a later time.

There are several design alternatives to choose from, and this is a fantastic chance for you to demonstrate your knowledge's breadth and depth to the interviewer. Let's see how many paths the interview may take from here. You can begin by discussing databases with the interviewer, weighing the advantages and disadvantages of SQL vs. NoSQL databases, evaluating various NoSQL database types, and providing particular names to the interviewer. When considering SQL or NoSQL for message storage, we may state that we do not want ACID transactions, do not need to perform sophisticated dynamic queries, and do not intend to utilize this storage for analytics or data warehousing. We need a database that can readily scale for both writes and reads instead. It must be highly available and tolerant of network failures. After considering all of this, it is apparent that NoSQL is the better option for our use case. If we need to pick a specific NoSQL database type, we should indicate that messages are restricted in size (say, no more than 1 MB). Thus we don't require a document store.

Furthermore, there is no discernible link between the texts. As a result, we can also rule out graph type. As a result, we're left with either column or key-value databases. We can think of a few well-known names for these two database formats. Apache Cassandra and Amazon DynamoDB, for example. In-memory storage is the next option we may consider. We should use in-memory storage that allows for persistence so that messages can survive for several days before being deleted. Also, several excellent in-memory storage options, such as Redis, should be mentioned. Message queues are another option to explore. Distributed message queues meet all of our requirements and may be examined in further depth.

Detailed Design

Metadata Service

Metadata service is the next component in the notification system. A web service for saving information in the database about subjects and subscribers: it's a cache that's distributed. When our notification service grows to the point where we have millions of subjects, all of this data cannot be stored on a single host's RAM. Instead, information about subjects is shared throughout cluster hosts. Cluster denotes a consistent hashing ring. Each Frontend host generates a hash, such as an MD5 hash, using a key, a combination of the topic name and the topic owner identity. The Frontend host selects a Metadata service host based on the hash value. We establish a component that is responsible for coordination in the first option. This component is aware of all Metadata service hosts since those hosts send heartbeats to it regularly.

Each Frontend host queries the Configuration service to determine which Metadata service hosts have data for a given hash value. When we scale up and add new Metadata service hosts, the Configuration service notices the changes and remaps the hash key ranges. We don't utilize a coordinator in the second option. Instead, we ensure that every Frontend host has access to data on all Metadata service hosts. When new Metadata service hosts are introduced or if a Metadata host dies due to a hardware issue, each Frontend host is alerted. Different techniques exist to assist Frontend hosts in locating Metadata service hosts. We won't go into detail about this issue here; we'll just describe the Gossip protocol. The propagation of epidemics is the basis for this procedure. This sort of protocol is usually implemented using random "peer selection," in which one computer selects another machine at random and exchanges data at a set frequency.

Sender

We can now begin delivering this message to subscribers when properly published and placed in Temporary Storage. You'll see that the concepts that the Sender service is based on are easily transferable to various distributed systems, not just the Notification service. Consider the following concepts if you're designing a solution that requires data collection, processing, and delivering findings in a fan-out fashion, which means messages are delivered to many destinations simultaneously. The first thing Sender does is retrieve messages. This is accomplished by using a pool of threads, each of which attempts to read data from the Temporary Storage. A naïve method would always be to launch a certain number of message retrieval threads. The difficulty with this method is that it may leave certain threads idle since there aren't enough messages to fetch.

Alternatively, all threads may become filled fast, and the only option to scale message retrieval is to add more Sender hosts. Keeping track of idle threads and dynamically adjusting the number of message retrieval threads is a preferable technique. No new threads should be generated if there are too many idle threads. More threads in the pool can begin reading messages if all threads are busy. This not only allows the Sender service to scale better, but it also prevents the Temporary Storage service from being assaulted by the Sender service. This is especially beneficial when Temporary Storage service suffers from performance issues, and Sender service can assist.

Temporary Storage service to recover faster by lowering the pace of message readings. How can we put this auto-scaling method into action? To the rescue, semaphores. A semaphore, in theory, keeps track of a set of permissions. Thread must get a permit from the semaphore before obtaining the following message. A permit is returned to the semaphore after the thread has read the message, enabling another thread from the pool to begin reading messages. We can dynamically change a few of these permissions based on the current and intended message read pace. After retrieving the message, we must use the Metadata service to acquire subscriber information.

Trade-Off

Every component may be scaled horizontally. When more capable hosts can execute more delivery jobs, the Sender service has a lot of potential for vertical scaling. Each component is distributed across many data centers, so there is no single point of failure. The front-end service is quick. We placed it in charge of some low-cost, small-scale operations. Several additional tasks were assigned to agents that operate asynchronously and have no influence on request processing. A distributed cache is what the metadata service is. It's quick since we keep active topic data in memory. We spoke over a few different Temporary Storage design alternatives and suggested several quick 3-rd-party solutions. And our Sender service divides message distribution into smaller jobs so that each one may be completed efficiently. Whatever Temporary Storage option we pick, data will be kept in a redundant way, which means that several copies of a message will be saved across many computers, ideally across multiple data centers. We also retry messages to ensure that they are sent at least once to each subscriber.

8

GOOGLE DRIVE DESIGN

Introduction

Google Drive is a cloud-based file storage service provided by Google. It offers an online storage space and synchronization platform that allows users to save data on remote servers. Google Drive will synchronize users' files across different devices, share them with other users as desired, and keep them on these servers. Dropbox, OneDrive, and Google Photos are comparable apps that support huge file storage and exchange for millions of users.

This chapter will go over the architecture of basic file storage and exchange service on Google Drive.

Requirements

It's critical to spell out the design's needs. The actual design of the application might include a variety of features and complexity that are outside the scope of a system design interview. Before constructing the system, define the needs to a few critical parts.

Functional Requirements

- Users may log onto the platform to upload and download from any of their devices.
- Users can exchange files with one another.
- The platform can synchronize files across multiple devices automatically.

Non-Functional Requirements

- Large files of up to 1GB apiece can be stored on the platform.
- The system must be scalable to a large number of users
- The service must be able to cope with a large number of reads and writes. In this scenario, the read-to-write ratio is similar.
- The smallest amount of network traffic feasible should be used for file synchronization.
- The file transfer should have the shortest possible latency.

High-Level Design

File processing Workflow

User A saves a chunk of data to the cloud. User A uses the Metadata server to modify metadata and save modifications to Metadata database. The user is notified, and notifications are delivered to the client's various devices. Metadata modifications are sent to multiple devices, which then download modified pieces from online storage.

Design of a file workflow in the Google Drive system:

Scalability

We have to divide the metadata DB to accommodate 1 million clients and billions of files. We may divide data into partitions to spread read-write requests across servers.

Metadata Partitioning:

- File pieces can be stored in partitions depending on the File Path's initial letter. For instance, we retain all files that start with the letter ‘A' in a partition and others that start with ‘B' in another partition, etc. The term for this method is range-based partitioning. We may merge less commonly occurring letters such as ‘Z' or ‘Y' into a single partition. The fundamental issue is that certain letters are interchangeable in the event of a first letter. For instance, if we put all files beginning with the letter "A" into a database partition and there are too many files beginning with "A," we will be unable to accommodate them all into a single database partition. In such instances, this strategy will be ineffective.

- The hash of the file's field can also be used to split. Our hash function will produce a random server address, which we will use to save the file. However, we may have to ask all servers to get a proposed list and combine it to reach the desired result. As a result, reaction time latency may grow.

This method may still result in overcrowded partitions that may be handled by employing Constant Hashing.

Caching:

Caching is a standard performance method. This is useful for reducing latency. Before reaching the DB, the server could check the cache server to determine if the query list is in the cache. We can't keep all of the data in the cache since it's too expensive.

When the cache is filled, and a file chunk needs to be replaced by a newer chunk. This method can make use of the Least Recently Used (LRU) algorithm. The least recently utilized chunk gets deleted from the cache first in this method.

Security:

The data protection of users is critical in a file exchange system. We may keep every file's rights in the metadata Db to deal with this, allowing us to specify what files are accessible or changeable by which client.

Client-Side:

All files which clients store on a cloud storage are transferred to the user application. The application will store, download, or alter files to cloud storage. A user can make changes to metadata, such as renaming a file, editing it, and so on.

Uploading Files in Small Pieces

The final two non-functional requirements, such as minimal bandwidth and lowest latency, are critical, which is why Google Drive and comparable systems prefer to upload data in pieces instead of one huge file.

If a heavy file, like 10 MB, were to be uploaded to the system as a single file, it would result in excessive latencies and traffic use. If the upload fails, the full file will have to be uploaded again, which will take more time, bandwidth, and cost. In addition, if you change the file, the system will re-upload and save the full 10 MB file. Every update in this method entails uploading and saving a new 10 MB file on the cloud services.

This method has two disadvantages:

- Every upload consumes 10MB of network capacity.
- Every time the file is changed, you will need 10 MB of cloud storage.

You can replace the original file on the cloud storage and preserve all of this space. Nevertheless, because a file storage system generally keeps records of the update activity, we'll need a system that can save all the changes while taking up as little cloud space as possible.

Chunking

Rather than uploading entire files, the service came up with a technique to divide files into chunks and manage these portions separately to enhance performance. Every file is broken down into little chunks of predefined sizes, such as 2 MB. The metadata collection can save information for every of these file pieces, such as name and sequence. The original file may be reconstructed by using the chunks while downloading or synchronizing.

If a customer uploads a 10MB file to the cloud services, the file storage service will split it into five 2MB portions before uploading it. If a client chooses to change a section of the file, just the 2MB chunk containing the modified section will be uploaded again, not the whole file.

Rather than sending the full 10 10 MB file again, you could just be uploading a small piece (the changed portion of the file), i.e., 2 MB, to the cloud system if you do a tiny change to a 10 MB file. The model preserves network bandwidth, uploading time, and data storage by using this method.

Detailed Design

There are several aspects to the overall architecture of the Google Drive system.

Architecture

- **Client**

The client is the Google Drive app on the smartphone or tablet. The customer keeps a close eye on the workspace files. On the client's computers, these workspace directories, often known as sync folders, are present. It is equally the client's responsibility to synchronize such local folders with the online storage. The client program on your computer will be used to upload, download, and modify the files in the cloud storage. The user will send a request to the block server to retrieve or store files to cloud storage, also known as the storage server.

The client is in charge of a variety of duties. Let's break down the desktop client into four sub dimensions to make it simpler to comprehend and improve effectiveness: Chunker, Watcher, Indexer, and Client's Metadata DB.

- **Chunker**

The importance of chunking files has already been discussed. The chunker is present in the client and is responsible for breaking the file into individual bits. The chunker is also in charge of recognizing which chunks are changed by the client and uploading just those pieces to the cloud storage, thereby conserving traffic. Chunker will equally reconstruct the file by putting the pieces back together in the proper sequence.

- **Watcher**

The watcher will keep an eye on the workspace files for any modifications and notify the indexer of any modifications executed on the documents by the user (such as addition, update, or removal). If many users can modify the workspace file, the watcher also monitors any modifications to the files received over the synchronization system by other users.

- **Indexer**

The indexer gets modifications to the workspace file through the watcher and modifications to the user's local metadata store with updated chunks for various files. After the changed chunks are posted to the online storage, the indexer will use a message queuing service to deliver messages to the synchronization system, which will update the metadata store and inform all users of the changes.

- **Metadata Database**

Each user keeps its local database in addition to the main distant metadata database. This database contains all of the details for that user's workspace directories, including files, chunks, and revisions. User's Metadata Database enables users to modify offline files, which are then implemented in the distant metadata database when the user is back online.

Message Queuing Service (MQS)

For remote access between the indexer and the synchronization system, we employ a robust message queuing service. The message queuing service ought to manage a large volume of read and write demands because all file modifications are routed through it. Aside from robustness, the message queuing service ought to store a large volume of simultaneous messages in a highly robust queue with excellent reliability.

There are two sorts of queues that our message queuing service will manage:

- **Request Queue**

A global request queue is maintained by the message queuing service, which accepts requests from all users. The indexer on the user pushes the request via the request queue whenever it gets any modifications to the files or directories. The synchronization system will forward the request to the request queue, which will modify the metadata DB.

- **Response Queue**

Contrary to request queues, the message queuing service will keep a response queue for every user. The synchronization service receives modifications and broadcasts them to the subscribing users' response queues. Subscribed users will get messages from the response queue with instructions on how to change their workspace directories. The purpose of keeping unique response queues for users is that they are removed from the queue after messages are sent. To ensure that each update is provided to all subscribers before being deleted from the queue, we should have separate response queues for every subscribing user.

Synchronization Service

The user uses the message queue to connect with the synchronization system for two reasons:

- To upload the most recent modification from the local workspace to the online storage. The synchronization system accepts the user's request from the message queuing system's request queue and changes the remote metadata DB with the user's most recent modification.
- To get the most recent updates from the online storage. It integrates the user's metadata DB with the remote metadata DB's changes for every subscribed user. The synchronization service broadcasts these changes to registered users via the message queuing system's response queues. The indexer downloads the most recent chunks from the online storage and modifies its file system. When a user stays offline for some time, it will poll Google Drive for the most recent updates when it reconnects.

In a nutshell, the synchronization service handles metadata and integrates files for clients.

Metadata Storage

We may utilize a cloud storage provider like Amazon S3 for storing file. The cloud storage system will essentially store the files. The system may download or upload files as required from this online storage. In other words, it creates a system that allows clients to upload and download things from the cloud and exchange them with others. It doesn't have the resources to create cloud storage.

Even though the Google Drive-like system does not provide file storage, it will keep records of every file chunk, file revisions, clients, and workspaces. A metadata DB may be used to hold all of this metadata data. To keep metadata, you may use a relational DB like MySQL or a NoSQL DB like MongoDB.

We have to maintain data consistency because multiple users may make modifications to the file simultaneously. In this scenario, MySQL could be a better option because it already provides ACID characteristics. If you want to use NoSQL for file synchronization, you'll have to include ACID properties in your design.

Block Service

The subservice that communicates with the online storage for processing files is the Block service. Anytime the user needs to upload or download a file piece from the online storage, it will use the block service.

When many users can modify the same file, the user will need the block service to acquire the revised file piece from the online storage. The indexer immediately gets a broadcast update from the messaging queue that a modification has been made.

Cloud Storage

As a cloud storage system, we may use Amazon S3. All of the file pieces will be sent to Amazon S3 storage centers and kept there. The users connect directly with the block service to transfer or get file pieces from the online storage.

9

AUTOCOMPLETE SYSTEM DESIGN

Introduction

Auto-complete or word completion is an application in which the rest of a user's term is predicted. Users can generally hit the tab key on graphical user interfaces for a recommendation or down arrow key to accept one or more. Autocomplete participates in numerous search functions on many platforms such as Facebook, Instagram, Google Search, etc. If you are asked to design one of the platforms, the interviewer may discuss the autocomplete function and other platform functionality. If you are requested to design Facebook, a certain time might be invested in developing a Facebook search box with automatic functions, especially for Facebook usage, including friendly users, mutual connections and suggestions in the heading. The words recommended originating from a preset vocabulary, e.g., Wikipedia or English dictionary, are all separate terms. The vocabulary can contain several terms: Wikipedia has millions of unique words. The vocabulary is significantly greater if multi-word sentences and names must be recognized. The system's core is an API that takes prefixes and looks for vocabulary terms starting with a particular prefix. Typically, there are returned just k numbers of potential completions. Many designs to implement such a system are available.

Requirements

Functional Requirements

- Obtain a list of recommendations based on the user entry, usually a prefix.
- Suggestions are sorted by weighing the frequency and recurrence of a specific sentence/question

The two primary APIs are:

Top-phrases (Prefix): Retrieves the list of top sentences for a specific prefix.

Collect-phrase (Phrase): Transmits the sought sentence to the system. The assembler then utilizes this sentence to construct a data structure that translates a prefix to a list of top phrases.

Non-functional Requirements

- Highly accessible.
- Performance: Reaction time for the top sentences should be faster than the user's pace (< 200ms).
- Scalable: The system should meet a high number of requests while retaining its performance.
- Lasting: Previously searched sentences (for a specific period) should be available, even if a hardware failure or crash exists.

High-Level Design

Data Partition

We'll partition our index to fulfill our better efficiency and lower latencies needs, even though it can easily fit one server. How can we divide our data effectively so that it may be distributed across several servers?

- **Range-based Partition**

What if we divided our sentences into groups depending on their initial letter? As a result, we save all words beginning with the letter "A" in one partition, those starting with the letter "B" in another, and so on. We can even merge a few characters that aren't used very often into a single database division. This partitioning scheme should be created statically to store and predictably search words permanently.

- **Partition Based on Maximum Capacity of the Server**

Let's assume we want to partition our trie based on the servers' maximum memory capacity. We can continue to store data on a server as long as it has memory. We break our partition there to allocate that range to this server and go on to the next server to continue this procedure whenever a sub-tree cannot fit into a server. If our first trie server can store all terms from ‘A' to ‘AABC,' our next server will store terms from AABD onwards. If our second server has enough space to store ‘BXA,' the following services will begin with ‘BXB,' and so on. To rapidly retrieve this partitioning technique, we can retain a hash table:

A-AABC A-AABC A-AABC A-AABC A-AABC A

AABD-BXA is the second server.

BXB-CDA is the third server.

If the user types the letter ‘A,' we must query servers 1 and 2 to obtain the top suggestions. We still have to query servers 1 and 2 when the user types ‘AA,' but when the user types ‘AAA,' we just have to query server 1. In front of our trie servers, we can put a load balancer to store this mapping and reroute traffic. Also, if we're querying several servers, we'll either have to combine the results on the server side to get the overall top results, or we'll have to rely on our clients to do it. If we want to accomplish this on the server-side, we'll need to add a new layer of servers between the load balancers and the trie servers, which we'll name aggregators. The top results from various trie servers will be returned to the client by these servers.

- **Partition Based on Hash of the Term**

Each phrase will be sent via a hash function, which will create a server number, which will be used to store the word. As a result, our word distribution will be random, reducing hotspots. We must query all servers and then aggregate the results to obtain typeahead recommendations for a keyword. For fault tolerance and load distribution, we must utilize consistent hashing.

Cache

We must understand that caching the most often searched phrases will be highly beneficial to our business. A tiny number of searches will be responsible for the majority of the traffic. Install a separate cache server in front of the trie servers to store the most commonly searched phrases and type ahead recommendations. Before contacting the trie servers, application servers should check these cache servers to verify whether they hold the necessary searched keywords. We may also create a simple Machine Learning (ML) model to predict interaction on each recommendation based on simple counting, customization, or trending data and then cache these phrases.

Replication and Load Balancer

For load balancing and failure tolerance, we need to have replicas for our trie servers. A load balancer that maintains track of our data partitioning strategy and redirects traffic depending on prefixes is also required.

Fault Tolerance

What happens if one of the trie servers goes down? We can have a master-slave setup, which means that if the master dies, the slave can take over after failover. Any server that restarts can recreate the trie using the most recent snapshot.

Type Ahead Client

To improve the user's experience, we can execute the following optimizations on the client:

- If the user hasn't pushed any keys in 50 milliseconds, the client should try reaching the server.
- The client can terminate in-progress queries if the user is continually typing.
- The client can first wait till the user inserts a few characters.
- Clients can pre-fetch data from the server to reduce the number of queries they have to make in the future.
- Clients can save the recent history of ideas locally because recent history is frequently repeated.

One of the most crucial things is to establish a connection with the server as soon as possible. The client can connect with the server as soon as the user visits the search engine page. As a result, when the user inputs the first character, the client does not waste time setting up the connection. The server might push a portion of its cache to CDNs and Internet Service Providers (ISPs) for efficiency.

Personalization

Users will receive type-ahead suggestions based on previous searches, location, language, and other factors. We can keep each user's personal history individually on the server and cache it on the client. Before transmitting the final set to the user, the server might include these customized keywords. Personalized searches should always take precedence over other types of searches.

Detailed Design

Trie Data Structure

Trie is a data recovery structure. It minimizes search complexity and optimizes and increases performance. In O(M) time, Trie may search for the key. It needs data storage, though. The storage might be an in-memory cache, a database or a file. Assume that S is a k string set. S = {s1, s2, ..., sk}, in other words. Model S as a rooted tree T so that each path from T root to one of its nodes corresponds to at least one string of S prefix. The best approach to demonstrate this building is to take a series of examples. Assume that S = {het, hi, hi, car, cat} and p is a blank string. Take into account that each edge of a child's internal node has a letter from our alphabet indicating the extension of the string. In addition, each node matching to an S string is colored. One observation is that if it is a prefix of another S string, then a node that fits it is an internal T node or a leaf. It is sometimes helpful to have all nodes, which correspond to S strings as leaves, and a character that is guaranteed to be not Ã 1 is extremely prevalent in each S string. In our example, mark $ as this particular character.

Because there is no longer a string in S that is a prefix, another string from S, all nodes in T that correspond to strings from S are now leaves. To make trieT, just start with one node as the root, which represents the empty string. Start at the root of T and consider the first character h of e if you wish to insert a string e into T. You consume the character h and descend to this child if an edge is marked with h from the current node to any of its children. If there is no such edge or kid at some time, you must construct them, consume h, and repeat the procedure until the entire e is completed. You may now just search in T for the string e. Iterate over the characters in e, starting at the root, and follow related edges to these characters in T. If there is no transition to children or if all of the letters of e are consumed. Still, the node where the process ends does not correspond to any string from S, then e does not belong to S. Otherwise, you'll finish the operation in a node that corresponds to e, making e a member of S.

Suffix Tree Algorithm

A suffix-tree is a compressed trie that contains all of a string's suffixes. Pattern matching, identifying unique substrings in a given string, finding the largest palindrome, and other string-related issues may be solved via suffix trees. A suffix tree T is a better data structure for pattern matching issues than a trie data structure defined over a collection of substrings of a string s. In essence, it's a trie that may have a longer route without branching. The preferable technique is to combine these lengthy pathways into a single path, which has the advantage of considerably decreasing the size of the trie. Consider the suffix tree T for the string s = abakan as an example. The suffix tree for the word abakan is as follows: abakan, bakan, akan, kan, an, n. Because it includes so much information about the string itself, suffix trees may solve a variety of complex issues. For instance, finding P in T and returning the size of a subtree corresponding to its node will tell you how many times a pattern P appears in s. Finding the number of different substrings of s is another well-known application, and it's simple to solve with a suffix tree. A prefix's completion is found by first following the path defined by the prefix's letters. This will lead to a node inside a node within a node within a node within a no.

The path of taking the left edge from the root and the lone edge from the child node, for example, corresponds to the prefix in the depicted prefix tree. The completions can then be created by traversing all leaf nodes that are reachable from the inner node. It is incredibly quick to search through a prefix tree. The number of comparison steps required to locate a prefix is equal to the prefix's letter count. This means that the search time is unaffected by the amount of vocabulary. As a result, even huge vocabularies can benefit from prefix trees. Prefix trees significantly increase the performance of over-ordered lists. Because each comparison may prune a substantially greater portion of the search field, the improvement is realized.

Minimal DFA

Common prefixes are effectively handled using prefix trees, but other shared word components are still kept separately in each branch. Suffixes like -ing and -ion, for example, are prevalent in the English language. Fortunately, there is a method for more effectively saving shared word portions. A prefix tree is part of a larger category of data structures known as acyclic deterministic finite automata (DFA). Some algorithms can be used to convert a DFA into a DFA with fewer nodes. The size of the data structure is reduced when a prefix tree DFA is minimized. Even when the vocabulary is enormous, a minimum DFA fits in the memory. The secret to lightning-fast autocomplete is avoiding costly disk requests. The Myhill-Nerode theorem offers us a theoretical description of the minimum DFA in string equivalence classes. When two states are indistinguishable, it indicates that they both run to final or non-final states for all strings. Not all of the strings are tested. The goal is to compute the indistinguishability equivalence classes progressively.

If a string of length k can differentiate p and q, they are k-distinguishable.

The inductive characteristic of this relationship is simple to grasp:

If p and q are (k-1) distinguishable, or if (p,) and (q,) are (k-1) distinguishable for some symbol, they are k-distinguishable.

We can't subdivide the partitions anymore after a certain point. At this time, the process should be terminated because no additional steps can create new subdivisions. We have the indistinguishability equivalence classes, which constitute the states of the minimum DFA after we finish. By selecting an arbitrary state in the source class, performing the transition, and then taking the whole, the transition from one equivalence class to another may be achieved.

Bottlenecks

Servers may become imbalanced due to this if we wish to store all words that begin with the letter "E" in a database partition. However, we subsequently discover that we have too many words that begin with the letter "E" to fit into a single database partition. We can see that the problem described above will occur with any statically stated scheme. It is impossible to determine if each of our partitions will statically fit on a single server. Even partitioning based on maximum capacity might result in hotspots. If there are many requests for words that begin with the letter ‘cap,' the server that holds it will be overburdened compared to others.

Trade-Off

The preceding explains how to create an autocomplete system. Furthermore, these data structures may be improved in a variety of ways. There are frequently more potential completions for a given prefix than can be displayed on the user interface. While the autocomplete data structures are intriguing, they are not required to be implemented. Open-source libraries are available that provide functionality. Elasticsearch, Solr, and other

Lucene-based search engines, for example, provide a fast and reliable autocomplete mechanism.

10

API RATE-LIMITING SYSTEM DESIGN

Introduction

At its most basic level, a rate limiter restricts the number of events that an entity, user, device, IP address, or other devices can execute in a given time interval. A rate limiter, in general, restricts the number of requests a sender may send in a given period. Once the cap is reached, it stops queries. By limiting the number of times any user may contact the API, rate-limiting protects your APIs against unintentional or malicious usage. Without rate limitation, any user may make as many requests as possible, resulting in "spikes" of demands that starve other users. When rate limitation is enabled, it can only make a certain number of queries per second. The procedure may be automated with the aid of a rate-limiting algorithm. What is the purpose of API rate limiting? Protect services against abusive behavior, brute-force password attempts, and security threats while lowering expenses and increasing income. Rate limiting is critical for public APIs where you want to ensure that everyone has a pleasant experience, even if some users take more than their fair share. Rate limiting is especially important for computationally expensive endpoints, especially when serviced by auto-scaling or pay-by-the-computation services like AWS Lambda. You may also want to rate restrict APIs that provide sensitive data since this will limit the amount of information revealed if an attacker obtains access through an unforeseen circumstance.

In other words, rate limiting is the process of restricting some activities by a certain threshold. The system will return errors if such threshold values are exceeded. You might argue that rate-limiting restricts the number of operations that can be completed in a given length of time. Rate limiting can be done in stages. A network request, for example, can be limited to one request per second, four requests per minute, and ten requests every five minutes. To prevent a system from being pulled down by hackers, rate limiting is required. To comprehend the significance of rate-limiting, we must first comprehend the DOS assault. A denial of service attack, often known as a DOS attack, occurs when hackers attempt to shut down a system by flooding it with many requests in a short amount of time.

Because a server has a limit on the number of requests it can handle in a given amount of time, as a result, the system cannot cope with the influx of demands; it is unable to cope with them. This may be avoided by using rate limiting, which protects the system from being inundated with needless requests. The system throws an error when a threshold value is exceeded. For example, rate restriction may be required in Leetcode or any website where we may run our code so that customers do not unnecessarily spam the code execution service. We may set rate limits for the system in a variety of ways, such as by user id. For example, we may limit a user's request activity to a server to 5 per minute. Rate-limiting can also be done depending on IP address, region, and other factors. We may also set a rate limitation for the entire system. For example, the system might not be able to handle more than 20K queries per minute. If the total number of user requests in a minute exceeds that threshold, the system will return problems.

Requirements

The following is a list of the system's requirements :

- Limit the number of API queries a single entity may make in a given period.

- Exceptional fault tolerance. If there are any issues with the rate limiter, for example, if a cache server goes down, the system as a whole is not affected.
- Because the APIs are accessed via a cluster, the rate restriction should be considered across all servers.
- Highly accessible.
- Not creating significant latencies.

High-Level Design

When a new client request comes, the Server initially consults the Rate Limiter to determine if it should be provided or denied. The request will be sent to the internal API servers if it is not denied. The Rate Limiter's job is to decide which customer requests will be fulfilled and which will be denied. API Rate Limiter is a distinct node that interacts with the webserver. Rate Limiter will advise the webserver whether or not the request should be forwarded to the API server. If the request limit exceeds the threshold, the server will discard the request.

Algorithms for Rate Limiting

Rate limitation algorithms come in a variety of flavors, each with its own set of pros and cons.

- **Leaky Bucket**

Leaky Bucket is a basic, straightforward way to rate limiting using a queue, which you can think of as a bucket storing the requests. When a request is submitted, it is added to the end of the queue. The first item in the queue is processed at regular intervals, or first in, first-out (FIFO). Additional requests are rejected if the queue is full or leaked.

Pros

- Smooths outbursts of requests and processes them at a rate that is close to average.
- Simple to deploy on a single server or load balancer.
- Given the restricted queue size, memory is efficient for each user.

Cons

- A spike in traffic might cause the queue to fill up with old requests, preventing newer requests from being completed.
- There is no assurance that requests will be completed within a certain period.
- If load balancing is employed, distributed setup problems include the need to apply a policy to coordinate and enforce the limit amongst servers.

- **Token Bucket**

Assume a bucket has a few tokens. When a request comes in, a token from the bucket must be taken to be processed. The request will be denied if no token is available in the bucket, and the requester will have to try again later. Per time unit, the token bucket is also replenished. By allocating a bucket with a predetermined number of tokens to each user, we may limit requests per user per time unit. When a user uses up all of his tokens in a given amount of time, we know he's reached his limit and will refuse his requests until his bucket is replenished.

Pros

- Simple and straightforward to implement.
- Efficient Memory

Cons

- The race condition counter may cause problems in a distributed system owing to race conditions.
- **Fixed Window**

Fixed-window limits, such as 3,000 requests per hour or ten requests per day, are simple to express, but they are vulnerable to spikes as the available quota resets at the window's boundaries. Consider a 3,000-request-per-hour limitation, which still allows for a surge of all 3,000 requests in the first minute of the hour, potentially overwhelming the server.

Pros

- Simple to put into practice.
- Because all that is done is save the counts, there is a smaller memory footprint.
- With Redis-like technology, you can utilize the built-in concurrency.

Cons

- Because it allows requests for both the current and future windows to be handled in a short period, a single burst of traffic at the window's boundary can result in twice the pace of requests being processed.
- If many consumers wait for a reset window, such as at the top of the hour, your API may be stampeded at the exact moment.
- **Sliding Window Log**

A queue of timestamps reflecting the times at which all historical calls happened within the time range of the most recent window is maintained for each user. When a new request is made, any timestamps older than the window time are checked and removed since they are no longer relevant.

Instead of doing this step for each request, it might be performed regularly, such as every ‘n' minutes or when the queue reaches a particular length. The user's queue is updated with the new timestamp. The request is permitted to proceed if the number of elements in the queue does not exceed the allowable count; otherwise, an exception is triggered.

Pros

- It's perfect.

Cons

- Memory use is high. To support many users or long window times, all request timestamps must be kept for a window time, which necessitates a lot of memory.
- For eliminating the older timestamps, there is a lot of temporal complexity.
- **Sliding Window Counter**

This hybrid technique combines the fixed window algorithm's cheap processing cost with the sliding log's better boundary conditions. We keep a counter for each fixed window, much like the fixed window algorithm. We account for a weighted value of the preceding window's request rate to smooth out traffic surges depending on the current date. We maintain track of each user's request counts by utilizing several fixed time windows, such as 1/60th the size of our rate limit's time frame. For example, when we receive a new request, we may retain a count for each second and calculate the sum of all counters in the previous minute to determine the throttle limit. Our memory footprint would be reduced as a result of this.

Pros

- Because just the counts are kept, there is no huge memory footprint.

Cons

- It only works for lookback window timings that aren't very tight, especially for small unit times.

Detailed Design

Rate Limiting in Distributed System

When we execute our application on several clusters of multiple nodes, we might run into two significant issues:

Synchronization Policies

If you wish to use a cluster of several nodes to impose a global rate restriction, you'll need to create a policy to do so. When sending queries to various nodes, a consumer might exceed a global rate limit if each node tracked its rate limit. The more nodes a person has, the more probable they will go over the global limit. Setting up sticky sessions in your load balancer to deliver each customer to only one node is the easiest method to enforce the restriction. A lack of fault tolerance and scaling issues when nodes are overwhelmed are among the drawbacks. A centralized data store like Redis or Cassandra, which provides more flexible load-balancing rules, is a superior alternative. The counts for each window and customer will be collected in a centralized data storage. This method has two major flaws: higher latency while sending queries to the data storage and race situations.

Race Conditions

The possibility for race situations in high concurrency request patterns is one of the most serious issues with a centralized data store. When you utilize a naive "get-then-set" technique, obtain the current rate limit counter, increment it, and then put it back to the data store, you'll run into this problem. The difficulty with this approach is that in the time it takes to complete an entire cycle of read-increment-store, more requests might arrive, attempting to save the increment counter with an incorrect (lower) counter value. This allows a customer to submit many requests in a short period, bypassing rate restriction constraints. One solution is to place a "lock" around the problematic key, preventing other processes from accessing or writing to the counter. A lock can soon become a severe performance barrier when employing distant servers like Redis as the underlying data store and does not scale effectively. A better method is to think in terms of "set-then-get," depending on atomic operators to implement locks in a highly efficient manner, enabling you to swiftly increment and verify counter values without the atomic operations getting in the way.

Optimizing for Performance

Another downside of utilizing a centralized data storage for checking the rate limit counters is the increased latency. Unfortunately, even querying a fast data store like Redis would add milliseconds of delay to each request. To make these rate limit decisions with the least amount of delay, do checks locally in memory. Relax the rate check conditions and utilize an eventually consistent model to do local checks. Each node, for example, can initiate a data synchronization cycle with the main data storage. Each node sends a counter increment to the datastore for each consumer and window regularly. These push atomically update the values. The node can then update its in-memory version by retrieving the changed data. This cycle of converging, diverging, and reconverging across cluster nodes ultimately becomes consistent.

Bottlenecks

The methods we described will have certain issues when applied to distributed systems. Within a minute, a user is permitted to make three requests. In this instance, the user has already made two queries and has made two more requests in two seconds. And the user's requests were routed via two distinct load balancers before being sent through two different rate limiter services. The Rate Limiter service obtains the value below the threshold and authorizes the queries since the DB already has the count number 2. As a result, we are now allowing four requests from the same user in a minute, which exceeds the rate limiter's maximum. This is the problem of inconsistency. We could utilize sticky session load balancing to ensure that a user's request always goes to the same rate limiter service. However, the issue is that it is not a fault-tolerant design. Because the system cannot serve user requests if the Rate Limiter service is unavailable, locks can overcome this problem. When a service utilizes database counter data, the database is locked. As a result, other services will be unable to utilize it until the counter data is refreshed. However, this technique has its own set of issues. It will add additional delay to the locking process. As a result, we must choose between performance and availability trade-offs.

Trade-Off

A rate limiter restricts the number of events that a particular entity (user, IP, etc.) may execute in a given time frame. A user can only submit five queries per minute, for example. To spread the user's data, we can shard the data depending on the 'UserID.' For fault tolerance and replication. We should utilize Consistent Hashing. For each user or IP, we may have various rate limiters for different APIs. A rate limiter's job is to limit the number of requests sent to or received from a system. Rate limitation is most commonly used to avoid DoS attacks by limiting the number of incoming requests from the user. It can be enforced depending on IP address, user account, and other factors. Normal DoS attacks may be avoided by rate-limiting, which we should be aware of. However, it may not be adequate in the case of a widespread DoS attack.

11

FACEBOOK NEWSFEED SYSTEM DESIGN

Introduction

Every social media site, including Facebook, Twitter, Instagram, Quora, and Medium, has a news feed mechanism. A news feed is a collection of postings containing text, images, or videos created by other entities in the system and customized for your consumption. While other entities create fresh postings, it is continuously updated. The continually changing list of stories in the centre of Facebook's site is known as Newsfeed. Status updates, photographs, videos, links, app activity, and likes from individuals, pages, and groups a user follows on Facebook are all included. In other words, it's a collection of photographs, videos, places, status updates, and other activities that make up a comprehensive scrollable version of a person's and his friends' life narrative.

Requirements

Functional Requirements

- A user's newsfeed will depend on the posts from the individuals, pages, and groups they follow.
- A user can follow a huge number of pages/groups and have a large number of friends.
- Images, movies, and text may all be found in feeds.
- All active users should be able to see new posts as they appear on our service's newsfeed.

Non-Functional Requirements

- Our system should create every user's newsfeed in real-time, with a maximum delay of 2 seconds.
- If a new newsfeed request comes in, it shouldn't take more than 5 seconds for a post to appear in a user's feed.

High-Level Design

Feed Generation

The postings or feed items of people and entities pages and groups that a user follows create the newsfeed. So, anytime our system receives a request to produce a feed for a specific user, let's say, Mark, we'll do the following:

- Get the IDs of all the users and entities Mark is following.
- Retrieve the most recent, most popular, and most relevant posts for those IDs; these are the posts that could appear in Mark's newsfeed.
- This is Mark's current feed, so rank these posts based on their importance to her.
- Save this feed in the cache, and then render the top 20 posts on Mark's feed.

When Mark reaches the end of her current feed on the front end, she can fetch the following 20 posts from the server, and so on. We produced the feed only once and cached it here. What about fresh postings from Mark's friends and followers?

- We should rate and add those new entries to Mark's feed if she is online.
- We may repeat the above processes every five minutes, for example, to rate and add newer content to her feed.
- Mark can then be alerted when additional things in her feed become available for her to acquire.

Feed Publishing

Mark must request and pull feed articles from the server every time he loads her newsfeed page. She can pull new data from the server when she reaches the end of her current stream. For newer goods, the server may either alert Mark and have her pull them or the server can push them. We'll go through these alternatives in further detail later.

Web Server

This maintains the user's connection. This link will be used to send and receive data between the user and the server.

Application Server

To carry out procedures for saving new postings in database servers. To retrieve and push the newsfeed to the end-user, we'd additionally need some application servers.

Metadata Database and Cache

To keep track of the metadata associated with Users, Pages, and Groups.

Post Database and Cache

Database and cache for posts: To keep track of the information associated with postings and their contents.

Video and Photo Storage

To save all of the material contained in the postings, use blob storage.

Newsfeed Generation Service

To aggregate and rank all relevant content for a user's newsfeed, generated and stored in the cache. This service would likewise get real-time updates, and these updated feed items would be added to any user's timeline.

Feed Notification Service

To alert the user that additional items have been added to their newsfeed.

Fan-Out Method

- **Fan-Out Write**

Offline generation is another term for it. We can have dedicated servers that are constantly producing and storing users' newsfeeds in memory. We may just provide the news feed from the pre-generated, cached location whenever a user wants it. Adjust a user's feed's usage pattern memory? Create and save in memory a newsfeed for people who don't log in too often. LRU-based caching is a simple method for fan-out writing. Learning the login pattern of users is a more thoughtful approach to fan-out writing. What time is it? Which days of the week are you talking about? When you send a new post, the system receives a written request. The written request is spread out to all of your followers to update their newsfeed using fan-out write.

Pro

- The read operation has a minimal cost.

Con

- For a user with millions of followers, the writing process is too expensive.

- **Fan-Out Read**

Sometimes referred to as Naive Implementation". Problems with this fan-out read include:

- Users with a large number of friends/followers will notice a significant slowdown since we must sift, merge, and rank a large number of postings.

- When a user loads their page, we produce the timeline. This may be sluggish and have a lot of delays.
- Each status update will result in feed updates for all followers in the case of live updates. As a result, our Newsfeed Generation Service may experience severe backlogs.

We may pre-generate the chronology and save it in memory to increase efficiency. When you request a news feed, the system receives a read request. Fan-out read sends a read request to all of your followers, asking them to read their content.

Pros

- When you request a news feed, the system receives a read request.
- Fanout read sends a read request to all of your followers, asking them to read their content.

Cons

- For a person with a large number of followers, the read process is quite expensive.
- Users will not be able to see fresh data until they pull.
- If we pull to get the most recent postings regularly, it's challenging to find the correct pull cadence, and most pull requests will return an empty answer, wasting resources.

Hybrid Approach

We can combine fan-out-on-write and fan-out-on-load techniques. We can, for example, cease pushing posts from users with a large number of followers like celebrities and just send data to people with a few hundred (or thousand) followers. We may allow the followers of celebrities to pull the updates. Because the push process may be costly for users with many friends or followers, we can save a significant amount of resources by deactivating fan-out for them.

Tedious Hybrid Approach

We can limit the fan-out to only their online friends in this method whenever a user publishes a post. A mix of a push to inform and pull to serve end-users is also a fantastic method to get the best of both worlds. A model that is only pushed or pull is less adaptable.

Feed Ranking

The most basic approach to rank posts in a newsfeed was by when they were published. However, today's ranking algorithms do a lot more to guarantee that essential articles get prioritized.

- Select essential "signals" that indicate the importance of a post first, and then work out how to combine them to arrive at a final ranking score.
- More precisely, we may choose criteria that are important to the relevance of every feed item, such as the number of likes, comments, shares, time of the update, whether the article contains images/videos, and so on, and then use these features to generate a score.
- This is usually sufficient for a basic ranking system.
- By continually analyzing if we are making progress in user stickiness, retention, ad income, and so on, a better ranking system may considerably enhance itself.

Data Partitioning

Sharding Posts and Metadata

- We need to divide our data over several machines to read and write it effectively because we have a large number of new postings every day and our read load is also quite high.
- We may use an architecture similar to the one described in Designing Twitter for sharding our DBs that store posts and their information.

Sharding Feed Data

- For feed data that is kept in the RAM, we can split it on a UserID basis.
- We can attempt storing all of a user's data on a single server.
- Pass the UserID to the hash function to map the user to a cache server where the user's feed objects will be stored.
- Also, because we don't intend to keep more than 500 FeedItmeIDs per user, we won't come into a situation where a person's feed data won't fit on a single server.
- We would always have to contact only one server to obtain a user's feed.
- Consistent Hashing is required for future development and replication.

Bottlenecks

- Very sluggish for people with many friends, as we must sort/merge/classify an enormous amount of postings.
- When a user loads his page, we create the timeline. This would be pretty sluggish and latency high.

- Every status update leads to feed updates for all followers for live updates. This might lead to large backlogs in our newsfeed service.
- For live updates, pushing (or reporting) fresh content for users can lead to excessive loading, particularly for persons or pages with many followers. We can pre-generate the chronology and store it in memory to increase performance.

Trade-Off

- Push customer complexity whenever possible
- Operational performance is just as essential
- Separating cache and permanent storage provides independent scaling

12

CHAT MESSENGER SYSTEM DESIGN

Introduction

Nowadays, we all use some form of personal chat messengers, such as WhatsApp or Signal. This application is used to send messages to people or groups. We can communicate by text messaging or social media (image, video, document, etc.). Facebook Messenger is a software program that enables users to communicate via text-based instant chat. Messenger users may communicate with their Facebook friends via text message or on their website.

Requirements

Functional Requirements

- Transmit text message (One to one).
- Acknowledgement of Sent, Delivered, and Read receipts.
- The last time an individual was seen.
- Send a message to the media.
- Management of profiles.

Non Functional Requirements

- Users should be able to communicate in real-time with the least amount of delay possible.
- Our system should be extremely consistent; users should view their conversation history across all devices.
- While high availability for Messenger is ideal, we may tolerate lesser availability for the sake of consistency.

- Group Chats: Messenger should enable many users to communicate in a group setting.
- Push notifications: When users are offline, Messenger should be able to inform them of new messages.

High-Level Design

Network Protocol

AN INTERMEDIARY IS REQUIRED whenever A sends a message to B since A does not know B's address. Of course, as an intermediary, the server would be involved. When A wishes to send a message to B, A must first send it to the server, then transmit it to B. However, is the server capable of initiating the request? HTTP is a request-response protocol. This implies that anytime the server receives a request, it is the only one capable of responding to the client. Thus, in our case, where the client is distinct from the sender of the message (A), the server cannot deliver the message straight to B. We can resolve this issue by utilizing WebSockets. It establishes a full-duplex connection over a single TCP port. When a user connects to the internet, a TCP connection is established with the server. This is a private tunnel via which the client and server may communicate securely, so it is referred to as a full-duplex connection.

Send Text Message

When user A sends a message to user B, if user A is not connected to the internet, the mobile client stores the message in a local SQL database, such as SQLite. When a user connects to the Internet, the client sends a pending message to a gateway such as G1, and the gateway establishes a duplex connection with the client. G1 requests MessageService to transmit a message. MessageService does a database query to determine whether or not B is currently connected to any gateway. If it cannot locate B in the database, it will store the message on the server, and when B becomes available, the service will pass the message to B via the gateway (G2) and remove the message from the server if it was already saved.

Detailed Workflow

User-A communicates with User-B via the chat server. The server acknowledges receipt of the message and forwards it to User-A. The server logs the message and delivers it to User-B. User-B acknowledges receipt of the message by sending it to the server. The server informs User-A that the message was successfully sent to User-B.

Acknowledgement of Sent, Delivered and Read Receipts

MessageService sent the message to B, but B did not open it. As a result, B's client will return an Ack to the MessageService indicating that the message has been delivered. MessageService will send user A an Ack of Sent message. Similarly, when user B reads a message, the client sends an Ack to the MessageService, and the service sends an Ack of read receipts to A.

Last Seen

We can determine a user's last seen in a variety of methods. When a user performs an action in the client, such as sending a text or a media message, the MessageService invokes the LastSeenService to update the user's timestamp. At times, the user is connected to the gateway, but the client is closed, and the message is continually received in the background, similar to utilizing notifications. Ack returned the message to the service. Because the user has not yet launched the program, they are system-initiated messages rather than user-initiated ones, and therefore LastSeen should not be changed. This manner, we can maintain the user's last seen information current in the database. However, there is a caveat here. Users can leave the program open and do nothing; in this case, LastSeen should be updated. Thus, the client can provide the LastSeen timestamp regularly, say every 5 or 10 seconds, and the LastSeenService will update the user's timestamp in a database. The downside of this method is that it regularly sends updates to the service and consumes network capacity. To store users' LastSeen, we may utilize Redis. This allows us to retrieve a user's last seen when needed rapidly.

Send Media Messages

Here, media communications may take the form of a picture, a video, or a paper. When user C delivers a media message, a service named MediaService is invoked. This service will store the media file to an external storage location or a content delivery network (CDN). Additionally, it generates a unique hash of the message and calls the MessageService. MessageService will transmit the message (which contains the hash) to user B's client, then download the media file using the hash. Why do we require hash here? The solution is to identify the message so that an acknowledgement may be sent to the sender and get the media file's storage location until it is downloaded to the user's device.

Load Balancers

Because a single instance of a service cannot handle the load, several instances of each service are required. The load balancer may be installed in front of each service to divide traffic across many instances of the same service.

Detailed Design

Messages Handling

To transmit messages, a user must first establish a connection to the server and then publish messages for the benefit of other users. The user has two choices for obtaining a message from the server:

- Pull Model: Users can regularly check the server to see whether they have any new messages. The server must maintain track of messages that have not yet been delivered. The server returns all outstanding messages when the receiving user connects to the server to request a new message. Regular server checks are required to minimize delay, which often returns an empty response if there are no outstanding messages. This will result in significant resource waste and does not appear to be an efficient approach.
- Push Model: Users can maintain an open connection to the server and rely on the server to inform them when new messages arrive. When a server gets a message, the message can be promptly sent to the designated user. In this manner, the server is not required to maintain a list of outstanding messages. We get the lowest possible latency because messages are delivered immediately via the established connection.

Maintaining Open Connection with the server

Through the use of HTTP Long Polling, clients request information from the server to understand that the server may not answer immediately. If the server does not have any fresh data for the client when the poll is received, the server keeps the request open and waits for response information to become available. When the server receives fresh information, it instantly delivers the answer to the client, thereby closing the current request. Once the client receives the server answer, it can instantly send another server request for future updates. This results in significant reductions in latencies, throughputs, and performance. The lengthy polling request may time out or encounter a server disconnect; in this case, the client must initiate a fresh request.

Storing and Receiving Messages from DB

Each time the chat server receives a new message, it must save it in the database. There are two ways to accomplish this.

- Allow for a large number of minor updates.
- Obtain a large number of records in a short amount of time.

To store the message, send an asynchronous request to the database. While designing our database, we must have the following in mind:

- How to operate effectively with a database connection pool.
- How to retry unsuccessful requests?
- Where should requests that fail despite several retries be logged?
- When difficulties are addressed, how do I retry these recorded requests?
- A wide-column database system like HBase can easily meet both of our criteria.

What is HBase, and how does it work?

HBase is a column-oriented key-value NoSQL database that may store numerous values for a single key in many columns. HBase is based on the Google BigTable database and works on top of the Hadoop Distributed File System (HDFS). HBase combines data to store new data in a memory buffer, which it then dumps to disk once the buffer is full. This kind of storage facilitates the storage of large amounts of tiny data and the retrieval of rows by key or the scanning of ranges of rows. HBase is an efficient database for storing variable-size data, which our business also requires.

Managing Users Status

We need to maintain track of each user's online/offline state and inform all affected users when that status changes. We can easily determine a user's current state since we keep a connection object on the server for all active users. With 500 million active users at any given moment, it will take many resources to broadcast each status change to all relevant active users. Around this, we can make the following optimization:

- When a user first opens the app, it may see the current state of all of their friends.
- We can send a failure to the sender and change the status on the client whenever a user sends a message to another user who has gone offline.
- When a user logs in, the server can broadcast that information with a few seconds delay to check if the person logs out quickly.
- Clients can get the status of those users visible in the user's viewport from the server. This should not be done frequently because the server broadcasts the users' online status, and we can live with the stale offline state of users for a time.
- We can get the current status whenever the client begins a new conversation with another user.

Data Partitioning

We'll need to spread the data over several database servers because we'll be storing a lot of it (3.6PB for five years).

UserID-based partitioning:

Let's pretend we split based on the UserID's hash to keep all of a user's messages in the same database.

Total Shards: 3.6PB/4TB = 900 shards for five years, assuming 1 DB shard is 4TB.

Let's say we preserve 1K shards for the sake of simplicity. We'll use "hash(UserID) percent 1000" to get the shard number and then store/retrieve the data from there. This segmentation method will also make retrieving conversation history for any user very quick. We can start with fewer database servers first, with several shards sitting on a single physical server. We can simply store numerous partitions on a single server since we may have multiple database instances on it. The hash function must understand this logical partitioning method to map numerous logical partitions on a single physical server. We may start with many logical partitions mapped to fewer physical servers since we will be storing a complete history of messages. As our storage need grows, we can add additional physical servers to spread our logical partitions.

Message-ID Partitioning

We should not utilize this method since retrieving a range of messages from a chat would be exceedingly sluggish if we store different user messages on distinct database shards.

Cache

We can cache a subset of recent messages like the last 15 in a subset of recent conversations displayed in the user's viewport like the last 5. Because we opted to store all of the user's messages on a single shard, the cache for each user should be located on the same computer.

Fault Tolerance and Replication

Our chat servers are keeping track of the users' connections. Failover TCP connections to other servers are quite difficult. If a connection is lost, it may be easier to have clients rejoin automatically. We can't have only one copy of a user's data since we won't retrieve it if the server that has it crashes or goes down permanently. To do this, we must either keep several copies of the data on separate servers or distribute and duplicate it using techniques such as Reed-Solomon encoding.

Chat Server

The chat server is a server or, more commonly, a cluster of servers that house all of the software, frameworks, and databases required for the chat app to function. This server, or group of servers, is in charge of receiving messages securely, identifying the proper recipient, queuing the message, and then delivering it to the recipient's chat client. A REST API, a WebSocket server, an AWS instance for media storage, and other resources can be used by the chat server.

Group Chat

Separate group-chat objects can be created in our system and kept on the chat servers. GroupChatID is used to identify a group-chat object, keeping track of who is in the conversation. Our load balancer can send each group chat message to the server that handles that group chat based on GroupChatID. The server that handles that group chat can loop over all users in the discussion to locate the server that handles each user's connection to deliver the message. We may keep all of the group conversations in a separate table partitioned by GroupChatID in databases.

Bottlenecks

The chat servers and temporary storage solutions are the primary bottlenecks in the system that are more prone to breakdowns.

Chat Server Failure: Failure of the chat servers in the system will result in the users' connections being lost. There are two approaches to dealing with chat server failure. Transferring those TCP connections to another server is one option; however, such fail-overs are not simple to execute. The second option, which is comparably easier, is to have the user clients immediately begin the connection in the event of a connection loss. The database must be updated with information from the server to which the user is connected, regardless of our technique.

Transient Storage Failure: Another component prone to failure is transient storage, resulting in the loss of messages en route to offline users. We can create duplicates of each user's temporary storage to prevent the loss of messages delivered to them while offline. When a user returns to the internet, both the original and replica instances of their temporary storage are searched and combined.

Trade-Off

The majority of these trade-offs are centered on databases. Despite the plethora of database management systems, software professionals are always surprised to find a solution that enables real-time changes. Many applications benefit from relational databases, but they can't push data to clients - at least not millions of them. Non-relational alternatives aren't much better in terms of real-time performance. RethinkDB came the closest to being a general-purpose real-time storage solution so far, but it failed to acquire momentum and eventually fell out of favor. Managed real-time storage systems do exist, but they frequently necessitate application design compromises. They become costly at scale because general real-time synchronisation methods and data structures are difficult to optimize. Technically, a chat system with a database that does not allow real-time updates may still be built.

13

CONSISTENT HASHING IN SYSTEM DESIGN

Introduction

Hashing, often known as hash code or simply hash, is the process of translating one piece of data — generally an arbitrary size object — to another piece of data of defined size, typically an integer. A hash function is a function that is used to translate items to a hash code. By offering a distribution strategy that is not directly dependent on the number of servers, consistent hashing overcomes the problem of rehashing. Consistent Hashing is a distributed hashing technique that assigns each server or item in a distributed hash table a place on an abstract circle, or hash ring, regardless of the number of servers or objects in the table. This permits servers and objects to grow without compromising the system's overall performance. If the output range of our hash function is zero to 2**32 or INT MAX, this range is mapped into the hash ring, and values are wrapped around. The same hash function is used to hash all keys and servers placed on the circle's edge. To determine which server to ask for or store a particular key, we must first identify the key on the circle and then proceed clockwise until we reach a server. Consistent hashing allows us to spread data across several nodes/servers with a little rearrangement. The way we give keys to the servers is where the magic of consistent hashing lies. The hash function in in-consistent hashing operates regardless of the number of nodes/servers. Assume that a virtual ring has been established and that keys and servers have been dispersed within the ring.

Position = hash(key) % (2^32) is the hash function. The number of places (or ring length) is 2^32 and is a fully random value; you may choose whatever huge number you like.

REQUIREMENTS

- A constant hash function regardless of system view and maps keys relatively equally on all computers is a fundamental criterion for consistent hashing implementation.

Hash Table and Hash Function

We can use a hash table to remedy this. We utilize a fixed-size array of N to map the hash codes of all keys in a hash table. To determine the array index, do a modulo operation on the hash of the key.

Hash = index (key) modulo N, where N is the array's size.

Because there will be multiple keys that map to the same index, each index is given a list or bucket to hold all items that map to the same index. We hash the key, identify the index, then check the bucket at that index to add a new item. Add the object to the bucket if it isn't already there. Hash the key to getting the index, then check for the key in the bucket at that index to locate an item by key.

Consider the following scenario: we need to retain a list of all the members of a club and search for any individual member. We might handle it by storing the list in an array or linked list and searching by iterating over the items until we locate the one we want, for example, by searching by name. In the worst-case scenario, this would entail looking through all members to see if the one we're looking for is last, or not there at all, or half of them on average. In complexity theory words, the search would have complexity O(n), and it would be relatively quick for a shortlist, but it would become more sluggish as the number of members increased. What could be done to make that better? Assume that each of these club members had a unique member ID, a sequential number representing their membership rank. Assuming that searching by ID is acceptable, we might put all members in an array with indexes that match their IDs (for example, a member with ID=10 in the array would be at index 10). This would allow us to access each member without the need for a search immediately. That would be extremely efficient, in fact, as efficient as it is conceivable to be, equating to the simplest feasible complexity, O(1), commonly known as constant time. However, our club member ID scenario is a little artificial. What if identification numbers were large, non-sequential, or random? Or, what if searching by ID wasn't an option and we wanted to instead search by name (or some other field)? Maintaining our quick, direct access (or something similar) and handling arbitrary datasets and less restricted search criteria would be beneficial. Hash functions come to the rescue in this situation. An appropriate hash function may be used to translate any piece of data to an integer, which will serve a similar purpose to our club member ID, but with a few key distinctions.

Because a decent hash function often has a broad output range, such as the whole range of a 32-bit or 64-bit integer, creating an array for all possible indices would be either difficult or impossible, as well as a huge waste of memory. To get around this, we may use a decently large array, say twice the amount of elements we plan to store, and modulo the hash to obtain the array index. As a result, the index would be index = hash mod N, where N is the array's size. The hash function is usually applied to a key rather than the entire object when working with complicated objects. Each object in our club member example might have multiple fields (such as name, age, address, email, and phone), but we could choose one of them to act as the key, such as the email so that the hash function would only be applied to the email. In reality, the key does not have to be a component of the object; it is common to store key/value pairs, where the key is often a short text, and the value can be any data. In these situations, the hash table or map is utilized as a dictionary, and several high-level languages employ this approach to build objects or associative arrays.

Distributed Hashing

Assume that the number of workers continues to rise, and it becomes increasingly impossible to keep all employee data in a hash table that can fit on a single computer. In that case, we'll try to disperse the hash table over several servers to avoid a single server's memory restriction. Objects (and their keys) are spread across several servers. In-memory caches like Memcached, Redis, and others frequently use this configuration. How will we know which server will store a key if there are several servers? The easiest method is to multiply the number of servers by the hash modulo.

Server = hash(key) modulo N, for example, where N is the number of servers.

To store a key, first, hash it to obtain the hash code, then modulo the number of servers to obtain the server where the key should be stored. In some cases, splitting a hash table into multiple portions hosted by various servers may be required or beneficial. One of the major objectives is to get beyond the memory constraints of utilizing a single computer, enabling arbitrarily huge hash tables. The objects and their keys are dispersed over multiple servers in this situation, thus the term. Implementing in-memory caches, such as Memcached, is a common use case for this. These configurations consist of a collection of caching servers that store many key/value pairs and are used to offer quick access to data that was previously saved (or calculated) elsewhere. An application can be designed first to fetch data from cache servers to reduce the load on a database server while improving performance. Only if it's not there—a situation known as a cache miss—resort to the database, running the relevant query and caching the results with an appropriate key so that it can be found the next time it's needed. How does distribution happen now? What factors are considered while deciding which keys to store on which servers? The easiest method is to multiply the number of servers by the hash modulo. The server is equal to hash(key) mod N, where N is the pool size. The client computes the hash, applies a modulo N operation, and uses the resultant index to contact the relevant server to save or retrieve a key (probably by using a lookup table of IP addresses). It's worth noting that the hash function used for key distribution must be the same for all clients, but it doesn't have to be the same as the one used by the cache servers internally.

Rehashing Problem

Not only for the keys from S3 but virtually all keys, the server location has moved. In caching servers, this will burden the origin since keys will be missing from the cache, and all of them will have to be rehashed. This is referred to as the rehashing problem. This distribution system is straightforward, easy to understand, and effective. That is, until the server count changes. What occurs if one of the servers malfunctions or goes down? Of course, keys must be reassigned to account for the absent server. If additional servers are introduced to the pool, the keys must be reallocated to accommodate the new servers. This is true for any distribution method, but the issue with our basic modulo distribution is that when the number of servers changes, most hashes modulo N will change, requiring most keys to be transferred to a new server. As a result, even if a single server is added or withdrawn, all keys will very certainly need to be rehashed into a new server.

The Solution: Constant Hashing

By offering a distribution strategy that is not directly dependent on the number of servers, consistent hashing overcomes the problem of rehashing. Consistent Hashing is a distributed hashing technique that assigns each server or item in a distributed hash table a place on an abstract circle, or hash ring, regardless of the number of servers or objects in the table. This permits servers and objects to grow without compromising the system's overall performance.

If the output range of our hash function is zero to 2**32 or INT MAX, this range is mapped into the hash ring, and values are wrapped around. The same hash function is used to hash all keys and servers placed on the circle's edge. To determine which server to ask for or store a particular key, we must first identify the key on the circle and then proceed clockwise until we reach a server. When a server is deleted or added, the only key from that server is migrated in consistent hashing. If server S3 is decommissioned, all keys from that server will be moved to server S1, but servers S1 and S2 will not be moved. However, when server S3 is destroyed, the keys from S3 are not dispersed evenly across the surviving servers S1 and S2. They were exclusively assigned to server S1, causing server S1 to become overburdened. When a server is added or deleted, it produces a predetermined number of copies known as virtual nodes of each server and distributes them around the circle to equally divide the load. Instead of S1, S2, and S3, we'll get S10 S11...S19, S20 S21...S29, and S30 S31...S39. The weight factor, also known as the factor for many replicas, varies depending on the scenario. On server Si, all keys that are assigned to replicas Sij are saved.

We perform the same technique to discover a key: identify the key's position on the circle and then go ahead till we find a server replica. The key is kept in server Si if the server replica is Sij. If server S3 is decommissioned, all S3 copies with labels S30 S31... S39 must also be decommissioned. The items keys next to S3X labels will now be re-assigned to S1X and S2X automatically. No keys that were previously allocated to S1 and S2 will be moved. When we add a server, the same things happen. If we wish to install a server S4 to replace S3, we'll need to add labels S40, S41,... S49. One-third of keys from S1 and S2 should be transferred to S4 in the ideal situation.

Determining Placement of Keys on Servers

We perform the following to determine which database server an incoming key lives on either to insert it or query for it:

Run the key through the same hash algorithm we used to determine where the database server should be on the ring. We'll obtain an integer value after hashing the key, stored in the hash space and mapped to any position in the hash ring. There are two possibilities:

- The hash value corresponds to a location on the ring without a database server. In this example, we move clockwise around the ring from the point where the key was mapped to the first database server. We enter the key into the first database server on the ring as it travels clockwise. When looking for a key in a ring, the same rationale applies.
- The key's hash value translates directly to the same hash value of a database server. Therefore we put it on that server.

Adding a Server to the Ring

We'll need to remap the keys if we add another server to the hash Ring, server 4. Only the keys that are shared between servers 3 and 0 need to be remapped to server 4. On average, only k/n keys will need to be remapped, where k is the number of keys and n is the number of servers. This is in stark contrast to our modulo-based placement method, which required remapping virtually every key. The impact of introducing a new server4 is seen in the diagram below: because server four now sits between key0 and server0, key0 will be remapped from server0 to server4.

Removing a Server from the Ring

Our consistent hashing method guarantees that the number of keys and servers affected is limited when a production server goes down. Only the keys between server three and server 0 will need to be remapped to server one if server0 goes down. The other keys remain unchanged.

Uses of Consistent Hashing

- Creating the Hash keyspace.
- Using a Ring to represent the hash space.
- DB Servers in Critical Areas (HashRing)
- Choosing the Best Location for Keys on Servers
- A server is being added to the ring.
- Taking a server out of the ring.

Bottlenecks

There is one drawback with this strategy. All of the keys may be mapped to the same server, resulting in that one server receiving all of the burdens while the others stay idle. This is an extremely inefficient arrangement that is prone to failure. A new notion has been created to cope with this. All of the servers are duplicated and put in the ring at different locations. As the number of servers grows, the distribution becomes more consistent, which aids in scalability.

Trade-Off

Consistent Hashing is an important topic in distributed system architecture since it addresses scalability issues with dynamic nodes and assignments while also providing fault tolerance. It's also extremely beneficial in interviews for system design. This idea enables the effective distribution of requests or data in the servers and their mapping to servers. It aids in Horizontal Scaling by increasing throughput and decreasing latency in the application. As you can see, there is no such thing as a flawless, reliable hashing method. There are trade-offs in all of them. However, they're all striving to balance distribution, memory consumption, lookup time, and construction time, which includes node addition and removal costs, much like the ones listed above.

14

YOUTUBE SYSTEM DESIGN

Introduction

Youtube is one of the world's most popular video-sharing platforms. Users may post, watch, share, rate, and report videos and leave comments on them. Content creators may post their material in the form of video to video sharing sites, and users can watch the uploaded content on a variety of screen sizes and internet speeds.

Requirements

Functional Requirements

- Videos should be able to be uploaded by users.
- Video sharing and viewing should be possible for users.
- Users may do searches based on the titles of videos.
- Our services should be able to track video statistics such as likes/dislikes, total views, etc.
- Video comments should be able to be added and seen by users.

Non-functional Requirements

- The platform should have a high level of availability.
- Users in different parts of the world should have the same response time.
- While the platform's user base grows, it should scale.

High-Level Design

Processing Queue: Each video will be added to a processing queue, dequeued for encoding, thumbnail creation, and storage later.

Encoder: To convert each video submitted into a variety of formats.

Thumbnails generator: For each video, we'll need a few thumbnails.

Storage of video and thumbnail files: We'll need some distributed file storage for video and thumbnail data.

User Database: We'd need a database to hold user data, such as name, email, and address.

Metadata Storage: Metadata database will contain all information about videos, such as title, file path in the system, uploading user, total views, likes, dislikes, etc. It will also be utilized to save all of the video comments.

Upload Video: The user uploads a video, then creates a job and places it in the video post-processing queue, returning the job id to the client for future progress checking. Our service picks up the job from the queue, performs series processing, such as checking for duplication and compression, and returns the job id to the client for future progress checking. After post-processing, save the movie from objecting storage (e.g. AWS S3 bucket). Video metadata should be saved (including video URL in S3 bucket)

Watch Video: A user requests access to a video using the video Id. The server returns the URL of the requested video. Let's have a look at the database for user information and video metadata. For our user information and video metadata, I'm going to utilize a NoSQL database, especially AWS DynamoDB, due to our service's scalability requirements and reasonably basic data schema. The video is stored on AWS S3.

The schema design for a NoSQL database differs significantly from that of a SQL database. The initial steps in the SQL schema are identifying entities, denormalizing them, and putting them into tables. Foreign keys are used to represent relationships between entities, and the SQL query language provides query flexibility. We first investigate access patterns for NoSQL databases, particularly DynamoDB. Then, if necessary, construct database schema and denormalize data. DynamoDB has a lot of scalabilities, but it also has a lot of access freedom. All entities should be put into one table with a properly constructed partition key and optional sort key, according to DynamoDB. We can add GSI (Global Secondary Index) to DynamoDB if we wish to enable new access patterns in the future; we do have some flexibility.

We'll support the following access patterns:

- Access the video using a video Id.
- Access the user information with a user Id.
- Given a user Id, gain access to all of the user's videos.

Detailed Design

Where would the videos be kept?

A distributed file storage system such as HDFS or GlusterFS can be used to store videos.

What is the most effective way to manage read traffic?

- We should separate our read and write traffic.
- We can divide our read traffic over several servers since each video will have multiple copies.
- We may use master-slave setups for metadata, with writes going to the master first and then replaying on all slaves.

Such configurations might result in data staleness. For example, when a new video is added, its metadata is entered in the master first, and slaves cannot view it until it is replayed at the slave, providing stale results to the user. This staleness may be acceptable in our system because it is just temporary, and the viewer can watch new movies within a few milliseconds.

What would be the location of thumbnails?

The number of thumbnails will far outnumber the number of videos. If we estimate that each movie will contain five thumbnails, we'll need a storage system to handle a lot of read traffic. Before selecting a thumbnail storage method, there will be two factors to consider:

- Thumbnails are tiny files with a maximum size of 5KB.
- When compared to videos, read traffic for thumbnails will be massive. Users will watch a single video at a time, although they may be seeing a page with 20 thumbnails of other films.

Video Uploads: Because videos might be large, we should enable resuming from the same place if the connection fails while uploading.

Encoding Video: Newly uploaded movies are saved on the server, and a new job to encode them into different formats is added to the processing queue. The uploader is alerted when all of the encodings is complete, and the video is made accessible for viewing and sharing.

Metadata Sharding

We need to disperse our data over several machines to execute read/write operations effectively because we have a large number of new movies every day and our read load is also quite high.

Sharding based on UserID hash

We can attempt storing all of a user's data on a single server. We can provide the UserID to our hash function while saving, and it will map the user to a database server where all of the information for that user's movies will be stored. While searching for a user's movies, we can utilize our hash function to locate the server containing the user's data and read it from there. We'll have to ask all servers to find videos by title, and each server will return a collection of videos. Before delivering the results to the user, a centralized server would aggregate and rank them.

Sharding based on VideoID hash

Each VideoID will be mapped to a random server where the metadata for that video will be stored using our hash function. We'll query all servers to discover a user's videos, and each server will return a collection of videos. Before delivering these results to the user, a centralized server will aggregate and rank them.

Videos Deduplication

Our service will have to cope with widespread video duplication due to a large number of customers submitting a large volume of video data. Duplicate videos might have different aspect ratios or encodings, overlays or extra borders, or be snippets from a larger, original video. The spread of duplicate movies has a variety of consequences:

Data Storage: Keeping several copies of the same movie might be a waste of storage space.

Caching: Duplicate movies reduce cache efficiency by using space that could be utilized for unique material.

Network Storage: Increase the quantity of data transmitted over the network to in-network caching systems by increasing network utilization.

Energy Consumption: Increased storage, an ineffective cache, and network utilization all contribute to energy waste.

End users will see duplicate search results, lengthier video startup times, and interrupted streaming due to these inefficiencies. Deduplication makes the greatest sense for our service early, when a user uploads a video, rather than afterwards when it is post-processed to detect duplicate movies. End users will see duplicate search results, lengthier video startup times, and interrupted streaming due to these inefficiencies. Deduplication makes the greatest sense for our service early, when a user uploads a video, rather than afterwards when it is post-processed to detect duplicate movies.

Load Balancing

We should employ Consistent Hashing among our cache servers, which will aid in load balancing between them. Due to the varied popularity of each video, we will be employing a static hash-based technique to map videos to hostnames, which may result in unequal demand for logical replicas. For example, if a video gets popular, the logical replica corresponding to that video will see more traffic than other servers. These unbalanced loads for logical replicas might lead to unstable load distribution on physical servers. Any busy server in one area can transfer a client to a less busy server in the same cache location to address this problem. For this case, we may employ dynamic HTTP redirections.

Cache

Our service requires a large-scale video distribution infrastructure to serve internationally scattered consumers. Using many geographically distributed video cache servers, our service should bring its material closer to the user. We need a technique that maximizes user speed while simultaneously distributing the load equitably among the cache servers. To cache hot database rows, we can add a cache to metadata servers. Application servers can rapidly verify if the cache has the necessary rows by using Memcache to cache the data before contacting the database. The Least Recently Used (LRU) policy may be an acceptable cache eviction strategy for our system.

Content Delivery Network (CDN)

A content delivery network (CDN) is a network of dispersed computers that provide web content to users based on their geographic location, the origin of the web page, and the presence of a content delivery server. Content is replicated in several locations using CDNs. Videos are more likely to be closer to the user, and videos will stream from a friendlier network with fewer hops. CDN computers make extensive use of caching and may deliver videos almost entirely from memory. Our servers in multiple data centers can deliver less popular videos, such as those with 1-20 daily views not cached by CDNs.

Fault Tolerance

For database server distribution, we utilize Consistent Hashing. Consistent hashing will aid in the replacement of a dead server and the distribution of load among servers.

Bottlenecks

The VideoID hash solution answers our problem of popular people but also redirects the focus to popular videos. What happens if a user gets well-known? There might be many queries on the server that are holding that user, causing a bottleneck in speed. This will have an impact on our service's overall performance. Some people, in comparison to others, may accumulate a large number of movies over time. It's tough to keep a consistent distribution of data from an increasing number of users. Because the service tries to load balance locally, numerous redirections may occur if the host receiving the redirection cannot provide the video. Furthermore, each redirection necessitates an additional HTTP request from the client, resulting in longer delays before the video begins to play. Again, because higher tier caches are only available at a limited number of sites, inter-tier (or cross data-center) redirections send a client to a distant cache location.

15

URL SHORTENING

Introduction

URL shortening is a technique for condensing lengthy URLs into shorter aliases. To produce shorter aliases for long URLs, URL shortening services like bit.ly or TinyURL are highly popular. You must create a web service that returns a short URL if a user provides a long URL and the original long URL if the user provides a short URL. Many applicants may believe that creating this service is not difficult. When a user provides a long URL, convert it to a short URL and update the database; when the user visits the short URL, look for the long URL in the database and redirect the user to the original URL. Is it that easy? When we consider the service's scalability, the answer is no. When asked this topic in an interview, don't go right into the technical specifics. Most candidates make this error and quickly begin citing a slew of tools, databases, and frameworks. In this type of question, the interviewer is looking for a high-level design idea to provide a solution for the service's scalability and durability.

Requirements

First and foremost, gather requirements! Typically, the question appears to be extremely straightforward and broad. That was done on purpose! It's just impossible to develop a functional, scalable system in one hour. Interviewers are interested in hearing how the candidate approaches the topic, identifying obstacles, potential hazards, and edge instances. It also puts your communication skills to the test. Always engage in conversation with your interviewer and offer follow-up questions. It's very important to do this in the beginning to gather basic system needs. We choose the system's features in this section. What are the requirements that we should concentrate on?

- The service should produce a shorter and unique alias for a lengthy URL.
- When a user clicks on a short link, the service should take them back to the original.
- After a set amount of time, links will expire.
- The system should have a high level of availability. This is critical to keep in mind since all URL redirections would fail if the service goes down.
- URL redirection should take place in real-time with the least amount of delay possible.
- Predictability should be avoided while using shortened links.

High-Level Design

Estimating Traffic

Assume our service receives 30 million new URL shortenings per month. Let's say we keep track of every URL shortening request and the abbreviated link for five years. During this time, the service will generate around 1.8 billion records.

30 million x 5 years x 12 months = 1.8 billion

Estimating Storage and URL Length

Let's say we want to create a short URL with seven characters. These characters are made up of 62 letters and numbers (A-Z, a-z, 0-9). Assume that we save URL queries for two years. As a result, the total number of items to be kept is estimated to be about:

300 M * 2 years * 12 months = 7.2 billion.

The size of each object is estimated to be about 500 bytes, resulting in a storage need of around:

7.2 B * 500 bytes = 3.6 TB.

Data Capacity Modelling

Discuss the data capacity model to determine the system's storage capacity. We need to know how much data we'll need to enter into our system. Consider the various columns or attributes recorded in our database and the amount of data that will be retained for five years. Let's make the following assumptions for various features:

Consider average long URL size of 2KB, i.e. for 2048 characters.

Short URL size: 17 Bytes for 17 character

Created_at: 7 bytes

Expiration_length_in_minutes: 7 bytes

The computation above yields a total of 2.031KB per database item for abbreviated URLs. When we compute total storage for 30 million active users, we get 30000000 * 2.031 = 60780000 KB = 60.78 GB each month. In a year, 0.7284 TB of data is generated, and 3.642 TB of data is generated in five years. For this quantity of data, we need to consider the reads and writes on our system. This will determine whether we should utilize a relational database management system (RDBMS) or a no-SQL database.

URL Shortening Logic

We may use hashing algorithms like Base62 or MD5 to transform a lengthy URL into a unique short URL.

Base62 Encoding

The Base62 encoder allows us to utilize a character and number combination that contains A-Z, a-z, 0–9 in total (26 + 26 + 10 = 62). So, for a 7-character URL, we can serve 627 = 3500 billion URLs, which is enough in contrast to base10 (base10 only contains numbers 0-9, so you will get only 10M combinations). If we use base62 and assume the service generates 1000 small URLs every second, this 3500 billion combination will take 110 years to exhaust. We may generate a random integer for the provided lengthy URL, convert it to base62, and utilize the hash as a short URL id.

MD5 Encoding

MD5 generates base62 output as well. However, the MD5 hash generates a long output of more than seven characters. The MD5 hash gives a 128-bit result. Therefore we'll cut 43 bits out of it to make a 7-character URL. MD5 has a high rate of collisions. We may obtain the same unique id for a short URL for two or more separate long URL inputs, leading to data corruption. So we'll need to double-check that this unique id doesn't already exist in the database. Because Base64 encoding uses 6 bits to represent a character, the result of the MD5 hashing method, which creates a 128-bit hash value, will be 22 characters. In this situation, we can choose seven characters at random or swap some characters and choose the first 7. There are a few difficulties that we could run across with this algorithm:

- What if, even after randomization and switching, two identical URLs produced identical shortened URL links?
- https://iq.opengenus.org/author/pul/ could be URL encoded into https%3A%2F%2Fiq.opengenus.org%2Fauthor%2Fpul%2F (UTF-8 encoding scheme). The above algorithm will produce two different shortened URLs.

The following are some possible solutions to the above:

- To solve the first difficulty, attach the user id (primary key) to the original URL once he has logged in and then apply the algorithm.
- Another option is to keep an ever-increasing counter with the URLs and append it to the original. However, at a specific limit, this solution may cause sequence overflow.
- The easy answer to the second problem is to allow only UTF-8 URLs; if the URL is encoded, we must first convert it to UTF-8 before using the algorithm.

Database

We can utilize an RDBMS that has ACID characteristics. However, relational databases have a scalability problem. If you believe you can utilize sharding to address the scalability problem in an RDBMS, the system will become more complicated. There will be conversions and many Short URL resolutions and redirections because there are 30 million active users. Scaling the RDBMS utilizing shard would increase the complexity of the architecture when we want to have our system in a distributed way since read and write will be hefty for these 30M users. In RDBMS, you may need to utilize consistent hashing to balance traffic and DB queries, which is a time-consuming operation. As a result, relational databases are not suitable for handling this volume of traffic on our system, and scaling the RDBMS will not be a wise option.

Database Sharding or Partition

We'll need to expand our database to accommodate a billion URLs. To enhance capacity or storage, we must divide our data over many computers. We can devise a Hashing-based partitioning approach. The machine number for a certain URL might be determined using a hash function. If we wish to divide our data into 512 machines/shards, we'll need to do the following:

Machine Number = HASH (shortened URL) (0-511)

Load Balancing

Load balancers are commonly used to handle restricted bandwidth and single-point failures, which can enhance an application's responsiveness and availability. Between these levels, load balancers might be used:

- The client and the server are two separate entities.
- The database and the server.

To evenly distribute load across the servers, a round-robin technique might be used at first. However, because it does not consider server load, this technique may cause problems because it will continue to send requests even if the server is sluggish. As a result, a preferable strategy would be to utilize the Least Bandwidth method, which delivers the request to the server with the least load.

NoSQL

The main issue with utilizing a NoSQL database is that it will eventually lose its consistency. We write something, and it takes a while for it to replicate to another node, but our system requires high availability, and NoSQL meets that demand. NoSQL is scalable and can easily accommodate 30 million active users. When we wish to extend the storage, we just need to add extra nodes.

Techniques to Generate and Store TinyURL

Bae62 Encoding

Let's look at how we can convert a lengthy URL to a short URL in our database. If we're using base62 encoding to create the Tiny URL, we'll need to follow the procedures below:

Because the small URL should be unique, check whether it already exists in the database (using get(tiny) on DB).

If it's already there for another long URL, then create a new short URL.

If the short URL is missing from the database, add a long URL and TinyURL put (TinyURL, longURL).

This approach works well with a single server, however if there are several servers, it will cause a race situation. When multiple servers collaborate, there's a chance they'll all generate the same unique id or tiny URL for different long URLs, and even after checking the database, they'll be able to insert the same tiny URLs in the database at the same time (which is the same for different long URLs), potentially corrupting the data.

While inserting the small URL, we may use putIfAbsent(TinyURL, big URL) or the INSERT-IF-NOT-EXIST condition, however this requires support from the database, which is present in RDBMS but not in NoSQL. Because NoSQL data is eventually consistent, the putIfAbsent functionality may not be accessible in the NoSQL database.

MD5 Approach

To produce TinyURL, use the MD5 method to encode the lengthy URL and take the first seven characters. Because the first seven characters of distinct lengthy URLs may be the same, check the database (as we described in approach 1) to ensure TinyURL isn't already in use.

Advantages

This method saves some database space, but how? If two users want to generate a tiny URL for the same long URL, the first technique generates two random numbers and requires two rows in the database. In contrast, the second technique uses the same MD5, so both the longer URLs have the same first 43 bits, resulting in deduplication and saving space because we only need to store one row instead of two.

For the identical URLs, MD5 saves some database capacity, but for two long distinct URLs, we'll run into the same issue as we did in method 1. Although we may use putIfAbsent, NoSQL does not support it. So let's move on to the third method for resolving this issue.

Counter Approach

Because counters are continually incremented, we may obtain a fresh number for each new request. Using a counter is a suitable choice for a scalable solution.

Single server approach: The counter will be maintained by a single host or server (for example, a database). When the worker host receives a request, it communicates with the counter host, which returns a unique number and increases the counter. When the next request arrives, the counter host responds with the same unique number, and so on.

TinyURL is generated using a unique number assigned to each worker host.

Bottlenecks

If the counter host is down for an extended period, there will be a problem; moreover, if the number of requests is excessive, the counter host may not handle the strain. As a result, issues include a single point of failure and a single bottleneck.

- What if there are some servers? You can't keep track of a single counter and send the results to all of the servers. Various internal counters for multiple servers with distinct counter ranges can be used to overcome this problem. For example, server 1 has a memory range of 1 to 1 million. Server 2 has a memory range of 1 million to ten million, and so on. However, if one of the counters goes down, it will be impossible to retrieve the range of the failure counter and maintain it for another server.
- Furthermore, resetting the counter will be difficult if a counter hits its maximum capacity due to the lack of a central host to coordinate all of these numerous servers. Because we don't know which server is the master and which is the slave and which one is in charge of coordination and synchronization, the architecture will be screwed up.

Trade-Off

To tackle this problem, we may utilize a distributed service called Zookeeper to handle all of these time-consuming tasks and deal with the many problems that a distributed system faces, including race conditions, deadlock, and data particle failure. Zookeeper is a large-scale coordination service that handles a huge number of hosts. It maintains track of everything, including server names, active servers, dead servers, and host configuration information. It maintains synchronization and offers cooperation amongst numerous servers.

CONCLUSION

System design interviews can be difficult to crack if you aren't adequately prepared. The questions are wide, with several possible explanations, and they necessitate a basic understanding of systems. Organizations like Shopify, Uber use system design interviews and Twitter to assess applicants for technical positions (e.g. software engineers, app developers, etc.). Therefore, if you like to work for a major tech organization, you'll undoubtedly have to pass system design interviews.

The exciting news is we've taken care of some of the leg work on your behalf through the fundamentals we have provided in this book. Instead of scouring the internet for preparation advice, sample questions, and topic descriptions, you can utilise the guide. We will summarize the basic things you need to get hold of as you prepare for your dream

There are several approaches to solving system design questions, however at the end of the day, you need one that continually: Demonstrate to your employer that you have the necessary knowledge, break down the task into small chunks, and you will make a great impression by following the method highlighted below:

Make inquiries to get clarifications. Asking good questions is the first stage in tackling a system design question. You must ensure that you fully comprehend the design's objectives and requirements. This will significantly affect the quality of your system, as the quality of a design is directly proportional to how well it achieves its goals. Start with a high-level design and then delve deep. It's time to begin the system designing after you've specified the system's needs.

Spend the first 20 minutes on high-level design. The employer will like to see the high-level design within the first twenty minutes of the design interview. Because most system design interviews take 45-60 minutes, the topic of how you organize your time arises. In the first twenty minutes of your design interview, you must anticipate swiftly defining the requirements and writing a high-level design for your system. Afterwards, you'll devote the remaining time delving further into the system's details. You simply have to integrate the essential parts when creating the high-level design.

First, delve deep into your most powerful area. Following that, you must delve deeper into the intricacies, beginning with the element with which you are best acquainted. For instance, if you have a front-end development background, you would begin with the front-end and go back through your design.

Taking this strategy will guarantee that you have enough time to delve further into the areas with more expertise. Due to the time constraints of your interview, it also might enable you to present more generic designs for areas where you have the least expertise. Both of these scenarios can assist you in making a better first impression.

Don't fall into a "rabbit hole" Finally, be careful not to fall too far into a "rabbit hole" with your complex ideas. Your employer will be looking for evidence that you may concentrate on the big picture of the design rather than getting too wrapped up in the system details.

Put everything together. Take a few time before the interview's conclusion to describe the solution and emphasize any major issues or areas for modification.

A nice method is to go over the objectives and requirements you gathered at the start of your interview and then show how the solution addresses those objectives. Simultaneously, you must point out any major flaws or trade-offs in the system. To tie all aspects of the system description together, you may also give a short overview of how the system design you developed will work from beginning to end.