System Design Interviews

Tech Takshila

Get Fit For System Design Interview

Introduction

We would like to thank our readers for picking up this book for learning system design interviews. System design interviews is a very vast topic with detailed topics of application oriented designing. Designing software system require high-level architecture of large system. During the interviews candidates are expected to answer question such as “Design storage system for Dropbox” or “How to design a search engine.” In 30 -40 minutes of time lapse. These kinds of questions are usually flexible, unpredictable, usually open-ended.

Here at Tech Takshila, we are a team of professionals who carry experiences from top technical companies such as Facebook, Amazon, Apple, Netflix or Google. This course is designed industrial experts for the ease of learning and knowledge from the industry. This is designed in such a way that it can be studied by a student as well as a professional. This book will help the students cover the gap between school and corporate world whereas, it will help professionals climb the corporate ladder. The objective of this book to provide an overview of system design questions. The knowledge from this book will help the candidate crack the interview successfully.

This book will be an in-dept guide which will help prepare candidate for the system design interview, by providing knowledge on basic software architecture concepts. It will cover topics from basics of system design such as distributed system, CAP theorem and NOSQL to advanced concepts.

The 7 cases studies of real time apps. These case studies will give an insight on how real time apps are made and scaled to a higher level for traffic management. The case studies will cover high-level design, component design, data models and brain exercise to test your knowledge on the topic covered.

Contents

Title i

Introduction ii

I System Design Basic Concepts 1

1 Distributed System 2

1.1 Introduction . . . 2
1.2 Evolution of Distributed System . . . 3
1.3 Types of Distributed system . . . 4
1.3.1 Client server . . . 4
1.3.2 Three-tier . . . 4
1.3.3 Multi-tier . . . 4
1.3.4 Peer-to-peer . . . 4
1.4 Common application of distributed system . . . 4
1.4.1 Microservices . . . 4
1.4.2 Distributed Caching . . . 5
1.4.3 Distributed Data Storage . . . 5
1.5 Caveats of distributed systems . . . 5
1.5.1 Naïve Approach . . . 6
1.5.2 Optimal Approach . . . 6
1.5.3 Caveats uncovered . . . 8
1.6 Conclusion . . . 10
1.6.1 Fitness Check up . . . 10

2 CAP Theorem 11

2.1 Introduction . . . 11
2.2 Historical Background . . . 12
2.3 Current State . . . 12
2.3.1 Making Trade-off decision - Flight Booking System . . . 12

2.3.2 Partition Recovery- E-commerce shopping cart 13
2.4 Conclusion 14
2.4.1 Fitness check up: 15

3 Consistent Hashing **16**
3.1 Introduction 16
3.1.1 Historical Background 17
3.2 Consistent Hashing 18
3.3 Consistent Hashing Applications and Simulations 19
3.3.1 Load Balancing 19
3.3.2 Distributed caching 22
3.4 Fitness check up 24

4 Relational VS No SQL **25**
4.1 Introduction 25
4.2 Relational Databases 26
4.3 NoSQL Databases 27
4.4 Comparison chart 27
4.5 Conclusion 28
4.5.1 Fitness Check up 28

5 Types of NoSQL databases **30**
5.1 Introduction 30
5.2 Overview 31
5.3 Aggregate Oriented Databases 31
5.3.1 Key-Value Data Model 31
5.3.2 Document Data Model 32
5.3.3 Columnar Data Model 32
5.4 Graph Data Model 34
5.5 Case Study Databases 36
5.6 Conclusion 37
5.6.1 Fitness Check up 38

II System Design Case Studies **40**

6 Instagram **41**
6.1 Introduction 41
6.1.1 Problem statement 41
6.1.2 Gathering Requirements 41
6.2 Detailed design 42

6.2.1 Architecture . . . 42
6.2.2 System components . . . 43
6.3 Component Design . . . 43
6.3.1 Posting on Instagram . . . 43
6.3.2 API Design . . . 44
6.3.3 Precompute Feeds . . . 45
6.3.4 Fetching User Feed . . . 45
6.3.5 API Design . . . 46
6.4 Data Models . . . 47
6.4.1 Graph Data Model . . . 47
6.4.2 Columnar Data Models . . . 48
6.4.3 Streaming Data Model . . . 49
6.5 Optimization . . . 49

7 Twitter **52**
7.1 Introduction . . . 52
7.1.1 Problem Statement . . . 52
7.1.2 Gathering Requirements . . . 52
7.2 Detailed Design . . . 53
7.2.1 Architecture . . . 53
7.2.2 System Components . . . 53
7.3 Component Design . . . 54
7.3.1 Tweeting a post . . . 54
7.3.2 API Design . . . 55
7.3.3 Precompute feeds . . . 55
7.3.4 Fetching User Feed . . . 56
7.3.5 API Design . . . 56
7.4 Data Models . . . 57
7.4.1 Graph Data Models . . . 57
7.4.2 Sample Queries supported by Graph Databases . . . 58
7.4.3 Columnar Data Model . . . 59
7.4.4 Streaming Data Model . . . 60
7.5 Optimization . . . 60

8 WhatsApp **63**
8.1 Introduction . . . 63
8.1.1 Problem Statement . . . 63
8.1.2 Gathering requirements . . . 63
8.1.3 Capacity Planning . . . 64

8.2 Detailed Design . . . 64
8.2.1 API Design . . . 65
8.3 Data Model . . . 65
8.4 Component Design . . . 66
8.4.1 One-to-One communication . . . 66
8.4.2 Push Notifications . . . 68
8.4.3 User Activity Status . . . 68
8.5 Optimization . . . 69
8.5.1 Addressing Bottlenecks . . . 70
8.5.2 Monitoring . . . 71
8.5.3 Extended Requirements . . . 71

9 Tinder **73**
9.1 Introduction . . . 73
9.1.1 Problem Statement . . . 73
9.1.2 Gathering Requirements . . . 73
9.2 Detailed Design . . . 74
9.2.1 Architecture . . . 74
9.3 Component Design . . . 75
9.3.1 User Profile Creation . . . 75
9.3.2 User Profile Information - Sample data Model . . . 76
9.3.3 Fetch User recommendations . . . 76
9.3.4 Geo-Sharded Index . . . 77
9.3.5 Swipes and Matches . . . 78
9.3.6 Matches Data Model . . . 79
9.3.7 User Switching Locations . . . 80
9.3.8 Elastic Search Cluster . . . 80
9.4 Optimization . . . 80
9.5 Fitness check up . . . 82

10 Google Docs **83**
10.1 Introduction . . . 83
10.1.1 Gathering Requirements . . . 83
10.2 Detailed Design . . . 84
10.2.1 System Architecture . . . 84
10.3 System Components . . . 84
10.3.1 Creating a document . . . 84
10.3.2 API Design: Document Management service . . . 84
10.3.3 Editing a document . . . 85

10.3.4 Differential Synchronization 85
10.3.5 API Design: Edit Patch Service 87
10.3.6 API Design: document management service 87
10.4 Data Mode 88

III Appendix **89**

A Appendix A **90**
A.1 Instagram 90
A.2 Twitter 91
A.3 WhatsApp 92
A.4 Tinder 93
A.5 Google docs 93
A.6 Netflix 93
A.7 Uber 96

A Appendix B **99**
A.1 Instagram 99
A.2 Twitter 99
A.3 WhatsApp 100
A.4 Tinder 103
A.5 Google Docs 104

A Appendix C **105**
A.1 Instagram 105
A.2 Twitter 106
A.3 Tinder 106

B Answer Keys **108**
B.1 Distributed System 108
B.2 CAP theorem 108
B.3 Consistent hashing 109
B.4 Relational vs No SQL 109
B.5 Types of No SQL databases 109
B.6 Tinder 110

C References **111**
C.1 Distributed Systems 111
C.2 CAP theorem 111
C.3 Consistent hashing 111

C.4 Relational vs No SQL 111
C.5 Types of No SQL databases 112
C.6 Instagram 112
C.7 Twitter 112
C.8 WhatsApp 112
C.9 Tinder 113
C.10 Google Docs 113
C.11 Netflix 113
C.12 Uber 114

Part I

System Design Basic Concepts

1 Distributed System

1.1 Introduction

Often, in a pizza outlet, we observe that there are separate sections specific to the various tasks involved in preparing a pizza which may include: getting the pizza base, adding the topping, adding cheese and cutting slices. In this example, the pizza outlet can be considered a distributed system consisting of separate sub-systems (e.g., getPizzaBase(), addToppings(), etc.) as shown in the image below.

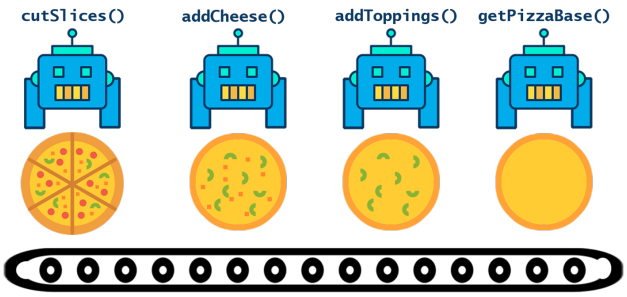


Figure 1.1: Pizza outlet - a distributed system

Distributed System also known as distributed computing and distributed databases can be defined as a level of abstraction in which multiple nodes (hardware or software components) coordinate their actions by communicating with each other on a network and still act as a single unit. We have listed below definitions of distributed system by a few well-known

computer scientists.

According to *Leslie Lamport's* definition, A distributed system is one that prevents you from working because of the failure of a machine that you had never heard of.

And *Andrew Tanenbaum's* definition, A collection of independent computers that appear to its users as one computer.

It seems from these definitions that a vast range of applications, from databases like Cassandra to streaming platforms like Kafka to the entire Amazon.com ecosystem are different flavors of distributed systems. In every case, there are two basic ways that distributed system function work: each machine works towards a common goal and the end user views results as one cohesive unit. Every machine has its own end-user and the distributed system facilitates sharing resources or communication services. Nevertheless, it's essential to understand how distributed systems evolved to its current state.

1.2 Evolution of Distributed System

Over the last few decades, the need to make services available to a wide range of people in real time led to the growth of distributed systems. The evolution was focused around availability, reliability, and reusability of the system. This has also led to the growth of standard communication interfaces between systems. In Fig. 1.2, it shows how the distributed systems paradigm evolved based on the need of the hour.

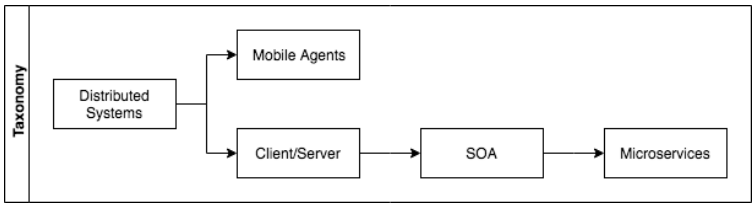


Figure 1.2: Distributed system Taxonomy

The client-server model has been around since the 1960s and 1970s when computer scientists building ARPANET (at the Stanford Research Institute) used the terms server-host (or serving host) and user-host (or using-host). Nevertheless, in 1990s, other paradigms such as Mobile Agents technology came into picture which was geared towards overcoming slow networks by enabling the capability to migrate software and data from one computer to another autonomously. Due to the growth of available and fast network connections and lightweight communication protocols (APIs), the client-server model gained more attention than its contemporaries. After that, Service Oriented Architecture (SOA) came into the picture to provide integration points between different organizations which relied on simple remote procedure calls such as Simple Object Access Protocol (SOAP). However, the need for more lightweight and modular systems led to the growth of Microservices.

1.3 Types of Distributed system

Distributed system and process typically use one of four architecture type below:

1.3.1 Client server

Distributed system architecture consisted of a server as a shared resource like a printer, database, or a web server. It used to have multiple clients, such as user behind the computers that decided to use the shared resource, how to use and display it, change data and send it back to the server. Code repositories like git is a good example where the intelligence is placed on the developers committing the change to the code

1.3.2 Three-tier

In this type of architecture, the information about the client is stored in a middle tier rather than on the client to simplify application deployment. This middle tier can be called as agent that receives requests from clients, that could be stateless, processes the data and then forwards it on to the servers. This architecture model is most common for web application.

1.3.3 Multi-tier

This was first created by enterprise web services for servers which contain business logic and interacts with both the data tiers and presentation tiers.

1.3.4 Peer-to-peer

There are no special servers which need to the intelligent work in this architecture. The decision making and responsibilities of all the servers are split among the machines involved and each could take on client or server roles. Example of such a system is Block chain

1.4 Common application of distributed system

The usage of distributed systems is widely prevalent in day-to-day applications. Some of its applications are listed below.

1.4.1 Microservices

: It is an architectural pattern where an application is structured as a collection of small autonomous services modeled around a business domain and each of those services comprises of a specific business capability. The microservice architecture enables the rapid, frequent and reliable delivery of large, complex applications. For instance, designing a

streaming platform such as Spotify (talk by their VP of Engineering if someone's just curious) may involve creating separate web services for different sub-domains of the business: user-identity management, user-playback history, audio content management and so forth.

1.4.2 Distributed Caching

With the growth of the internet, it was not possible to cache the information for quick look-up on a single host. The usage of distributed systems such as Distributed Hash Tables(DHT) helped us solve this problem by using efficient hashing(e.g. Consisent Hashing Webmaster: Include link to Consistent Hashing) mechanisms and communication protocols(Chord).

1.4.3 Distributed Data Storage

The growth of NoSQL over the past decade led the innovation in the field of distributed data storage systems, which can run over large clusters of commodity hardware. These systems can be catered towards a range of query patterns: simple key-value data stores like Amazon DynamoDB and graph-based stores like Neo4j to time-series database such as InFluxDB.

1.5 Caveats of distributed systems

There's no doubt in the fact that building distributed systems is more complicated compared to centralized systems, however, with an increase in requirement of high computing and data-storage power, there was a need to develop more complex systems - under the purview of distributed systems. Let's try to understand the complexities involved in a distributed system by doing a quick brain exercise.

Brain Exercise 1: There is a contest to vote for your favorite TV series. Ten popular TV series (Friends, GOT, The Big Bang Theory, Breaking Bad and so forth) are part of this contest. The contest will last for a few hours. We need to build an application to support this voting contest. The major requirements which we need to support in this application are:

Users should be able to use mobile and desktop platforms to cast their vote. Administrators of the voting application should be able to access the current votes count. Since the voting will last for only a couple of hours, we expect the frequency of votes(TPS) to be significantly high (i.e., tens of thousands).

We recommend you to think about a solution to this problem before reading further.

1.5.1 Naïve Approach

One possible approach to solve the brain exercise problem can be to build an application where each server/host maintains the count of votes. Followed by, the total votes count is obtained by fetching the votes count from individual servers and summing them up. However, a major issue with this approach is that in case a host goes down, then the votes count stored on that particular node will be lost as well.

1.5.2 Optimal Approach

We will cover the detailed design of this system in one of our later chapters; however, here we are covering a high-level overview to explain the complexities of distributed systems using an example. The voting application can be broken down into two separate components: Online component (aka user plane, data plane) and the Offline component (aka control plane). The online part helps with interfacing with voters on real-time. For example, users can cast their vote and get a confirmation when the vote is persisted onto the database securely. On contrary, control plane supervises the background operations for the system (e.g. tallying up the votes).

Online Component/Data Plane:

In Fig 1.3, the design of the data plane of this voting application is shown. The voters can use the mobile and desktop platforms to cast their vote by placing HTTP requests. Those requests will be forwarded to load balancers using DNS lookups. The load balancer will use scheduling algorithms to direct the traffic to application servers, which will save the votes in the data store.

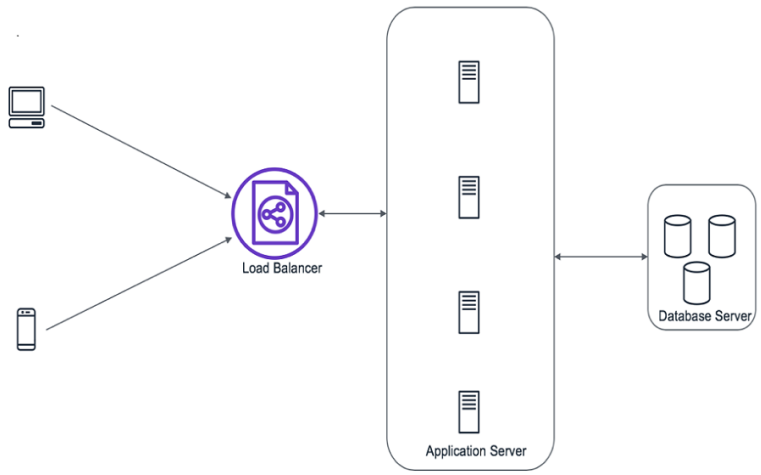


Figure 1.3: Voting Application: Online component

Data Model

You may use any aggregate-oriented NoSQL data store (key-value store such as Riak, Amazon DynamoDB) as they are cheaper and scalable compared to traditional relational databases. One way of modeling the data can be to have separate vote counts for different TV series (using TV series name as key), which gets updated whenever a vote gets cast. However, a skewed distribution of votes (as shown in Fig 1.4) can lead to a scenario where the partition on the disk storing the TV series having higher votes can get overwhelmed by the writes.

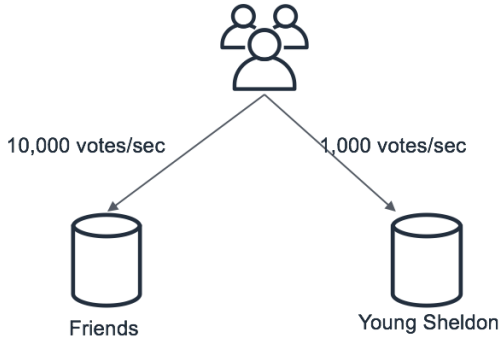


Figure 1.4: Skewed distribution of casted votes

The issue related to skewed distribution can be solved using write-sharding by attaching a random number to the TV series name, as shown in Fig 1.5 below. Using this approach, the writes for a particular TV series name gets distributed across multiple shards (more details in upcoming chapters)

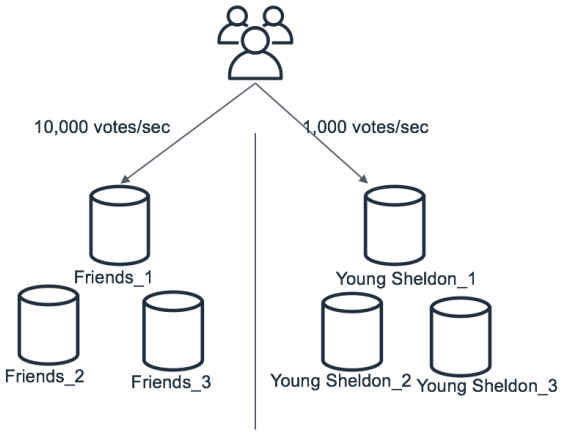


Figure 1.5: Write sharding to solve skewed distribution problem

Offline components

In the offline component, you can use a cron job which triggers a map-reduce operation that scans the sharded records, counts votes of all the TV series and persists

the final votes count in a separate data store as shown in Fig 1.6 We will discuss more details about map-reduce and its applications in future.

After that, the administrator's request to fetch the current votes count is served by the database table which gets populated by the offline operation (Votes Count table in Fig 1.6).

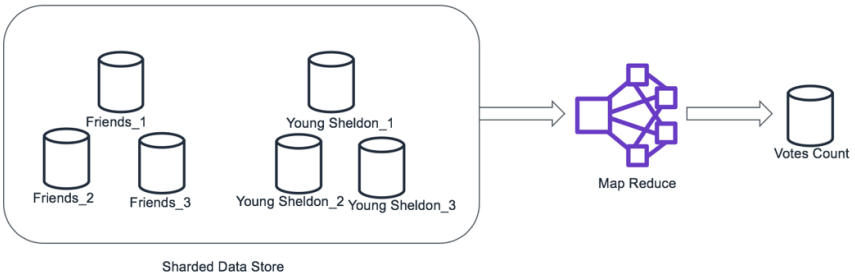


Figure 1.6: Offline operation to compute votes count

1.5.3 Caveats uncovered

You can use the voting application to gain in-depth insights into some of the caveats, which can help software development teams in designing more reliable, robust, and scalable systems.

Business Requirements

System Design depends a lot on the business domain using which we can make the trade-off between reliability(consistency) and performance(availability). For instance, building a financial system requires data to be consistent even though it comes at the cost of reduced performance. However, in the voting application, we may prefer availability over consistency (aka CAP Theorem Webmaster: Include link to CAP Theorem) as it's important that users can cast votes, even if the administrators get the exact vote counts after a minimal delay.

Data Modeling

The choice of the data store and the data schema depends a lot on users requirements and query patterns. For social media applications, we can use graph-based data stores like Neo4j, applications similar to Fitbit, which are related to time-series data may use databases like InfluxDB. In the voting app above, the data is stored in a key-value data-store by using the write-sharding mechanism to tackle the problem of skewed votes distribution.

Node Failure

Failure of nodes is an unavoidable phenomenon, and developers should take this into account while building a robust system. Under the purview of distributed systems, we employ techniques such as sharding and replication to minimize the impact of such node failures. In the voting application, we account for the fact that the hosts used in the application servers can go down, and so the data is persisted in a separate data store rather than on individual servers(naïve approach).

Network Failure

Similar to node failure, it's challenging to prevent network failure, so developers should account for error-handling on networking errors. One way of handling such error-scenarios is the retry policy, which varies a lot depending on business requirements (e.g. latency and availability SLA). For instance, we can have a retry mechanism in the voting application to persist the information in the data store to account for issues related to network failure.

Network Security

Often, complacency related to network security leads to the system being vulnerable to malicious users and programs that keeps on adapting to the existing security measures. Using secure protocols such as HTTPS in the voting application will ensure that malicious users don't cast the votes.

Infrastructure Cost

We should be mindful of the infrastructure cost required for running the system. It's quite common in the industry to see developers being blind-sided about the cost factor in their designs. The usage of NoSQL data store in the voting application is an approach to minimize the infrastructure cost as the NoSQL data stores are efficient at running on large clusters of commodity hardware.

Communication Interface

It's imperative to use best practices while designing APIs for the web-services using light-weight communication protocols. This allows the creation of secure and extensible integration point for those services. In the voting application, using light-weight REST APIs in the data plane provide ease of access from different platforms such as desktop and mobile.

1.6 Conclusion

In this chapter, we have tried to give our readers an overview of distributed systems along with its applications and key learnings which we have had based on our experience. We have covered the main concepts of distributed systems here and will discuss in detail about them in our later chapters.

1.6.1 Fitness Check up

Alice built a blog application using Django framework where the modules for user authentication, submitting posts, and comments are all bundled and deployed on the same stack. What will you call that application? -Monolithic -Microservices

Correct Answer: a

Explanation: Monolithic architecture suits simple light-weight applications where all the business logic can be bundled together. On the other hand, microservices architecture is better suited for complex and evolving applications.

What was the reason that we added a random number at the end of the TV series name in the voting application? -To increase indexing performance -It will solve the problem of hot partitions which may be created due to the skewed distribution of votes.

Correct Answer: b

Explanation: Distribution of votes can be skewed, and this will cause hot partitions. This may lead to the server being overwhelmed with the flood of requests.

-What are the major advantages of using microservices? -Easier to build and maintain applications -Flexibility in using technologies and scalability -All of the above

Correct Answer: c Explanation: Microservices tackles the complexity problem by breaking the application into a set of manageable services which are much faster to develop. It reduces the barrier of adopting new technologies

2

CAP Theorem

2.1 Introduction

When you visit a restaurant, there is always a possibility that the food doesn't taste good. The probability of which is lesser in a well-known restaurant owned by renowned chefs like Gordon Ramsay, in comparison to fast-food chains. This may come at the cost of waiting in long queues at restaurant, which may not be the case with fast food joints.

We can use the same analogy to develop an intuition for CAP Theorem, which is based on three essential aspects of a distributed system.

- Consistency: Every read receives the most recent write
- Availability: Every request received by a functional node must result in a response
- Partition Tolerance: The distributed system should continue to operate despite communication breakages that separate the network into clusters which are unable to communicate with each other.

CAP Theorem states that it's impossible for a distributed system to offer more than two out of the three aspects mentioned above. Similar to the fact that food can taste bad at any restaurant, there's always a possibility of network clusters not able to communicate with each other in a distributed system. Based on how long you are ready to wait for your food, you can make the choice to go between Gordon Ramsay's restaurant (where you may have to wait for a long time) and the fast-food joint (where you will get the

food almost instantly). On the same lines, while designing distributed systems, one has to choose between consistency and availability based on the business requirements, more details will be covered in later sections.

2.2 Historical Background

CAP theorem was proposed in the field of theoretical computer science by famous computer scientist Dr. Eric Brewer at the 2000 Symposium on Principles of Distributed Computing (PODC). During the time, the size of data was growing immensely, and the existing ACID databases weren't able to scale to satisfy that demand. This led to the growth of a new paradigm called BASE (basically available, soft-state, eventual consistency). Brewer analyzed the paradigm changes and its implications and presented his findings, marking the inception of CAP Theorem.

2.3 Current State

It is a matter of the fact that no distributed system is safe from network failures, network partitioning has to be tolerated, and we have to make the trade-off between consistency and availability. The resulting system isn't entirely consistent or available - but a combination which is reasonable for specific business needs. It implies that when a system chooses consistency over availability, the system will return an error or a time-out if particular information cannot be guaranteed to be up to date due to network partitioning. When choosing availability over consistency, the system will always process the query and try to return the most recent available version of the information, even if it cannot guarantee it is up to date. We have listed below some real-world applications of CAP Theorem to make it more intuitive for our readers.

2.3.1 Making Trade-off decision - Flight Booking System

Let consider that you need to build a distributed system which enables customers to book flight tickets. In Fig. 2.1 below, we have Alice and Bob who are in separate geographical locations(Alice is in New York and Bob in Sydney) and both of them want to book the last flight ticket using your distributed system. They are on different geographical locations, their requests are directed to different nodes on your system(Alice's request to the Miami node and Bob's to the Beijing node). For ensuring consistency when Alice wants to book the ticket, then the Miami node needs to communicate with the Beijing node, and both nodes must agree on the serialization of the requests. It provides consistency; however, in the case of any network breakage, both the nodes won't be able to book the flight ticket, sacrificing availability.

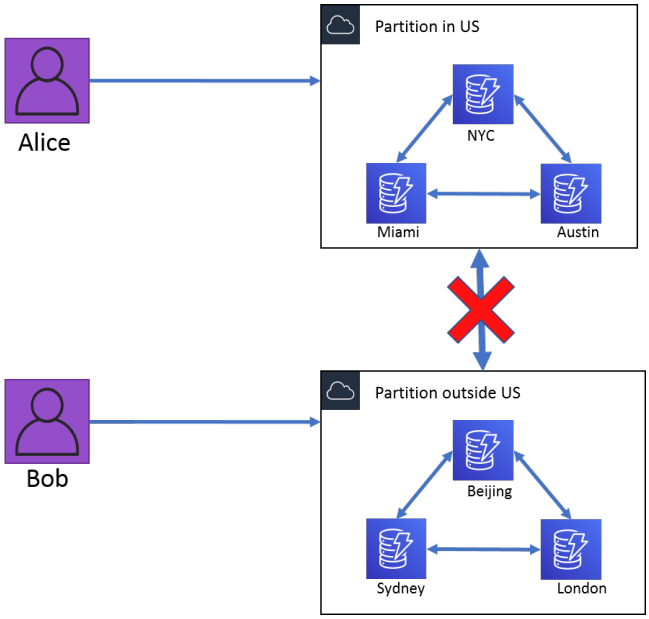


Figure 2.1: Making Trade-off decisions- flight booking system

Brain Exercise: What will you choose in the Flight Booking System: Consistency Or Availability? Why?

We recommend you to think about the solution of the brain exercise before reading further.

We may want to prefer availability over consistency in a flight booking system to provide a better experience to our customers. To gain more availability, we might allow both the nodes to keep accepting flight reservations even if the communication line breaks. The worst possible outcome of this approach is that Alice and Bob both will end up making the flight reservation. However, such situations can be resolved using domain knowledge. It's a pretty common occurrence that the flights are overbooked and then the flight companies address such cases by taking the appropriate measures (e.g., Refunds, moving to another flight, etc.).

2.3.2 Partition Recovery- E-commerce shopping cart

It is sometimes observed on e-commerce websites, couples share the same customer account. Alice and Bob are a couple who are in different geographical locations(Alice is in London, and Bob is in Boston), and they are adding items on an e-commerce website's shared account's shopping cart(shown in Fig 2.2 below). As they are in separate geographical locations, Alice's cart exists on the London node, and Bob's on the New York node of the website's distributed database system. Let's consider a scenario when a network failure occurs in the distributed system, and the two nodes(i.e., London and Boston) are no longer able to communicate with each other. It results in a situation where Alice's cart doesn't show the items added by Bob and vice versa(aka network partitions).

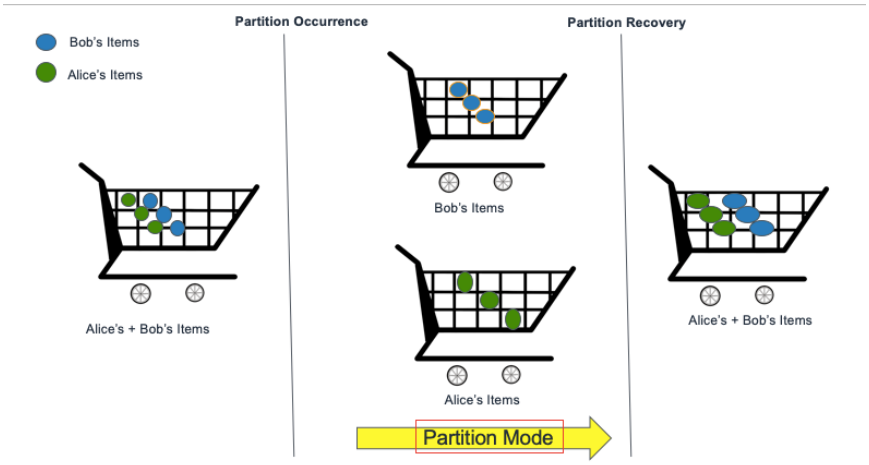


Figure 2.2: Handling Network Partitions: E-commerce carts

Brain Exercise: If you were building this e-commerce system, will you allow customers to keep adding items on the cart when the network failure occurs? How will you handle failure scenarios related to network partitions?

We recommend you to think about the solution of the brain exercise before reading further.

If you are designing an e-commerce application, you probably don't want to restrict your customers from adding items to their cart as it will deteriorate the customer experience and the business may also end up losing money. There are several ways to recover from such network partition scenarios. One possible solution can be to trigger the checkout process only after ensuring that the partition has recovered and then merge the items from Alice's and Bob's cart in a single cart as shown in Fig 2.2. If you are curious to learn more techniques of partition recovery, then we recommend you to watch this talk by Dr. Eric Brewer.

2.4 Conclusion

It is observed that the situations are tied to the domain and requires domain knowledge to determine the resolution. For a news media, we can tolerate old pages for minutes; however, in case of financial trading instruments, we can't rely on stale data. This decision needs the involvement of the development team and the domain experts. CAP Theorem is often used (sometimes abused) when talking about consistency in distributed systems. However, the significant trade-off we make is actually between consistency and latency. Consistency can be improved by involving more nodes in the interaction at the cost of adding more time engaged in the communication. It implies that we should think of availability as the maximum limit of latency that can be tolerated, once that limit is breached, the system is deemed as unavailable.

2.4.1 Fitness check up:

1. We need to build a distributed data store for a banking application. Given that such an application requires strong consistency, it comes at the cost of high latency. What is the reason behind increased latency?

   (a) More nodes are involved in the quorum to decide whether a write is successful or not.

   (b) It's due to the time to recover from a partition.

   Correct Answer: a

   Explanation: More nodes are involved in making the decision if the write was successful. The reason being that more network hops are included in making sure that the read after each write in the banking applicaton is consistent across the board.

2. What should be the ideal choice to build functionality to support comments on a blog?

   (a) Consistency

   (b) Availability

   Correct Answer: b Explanation: In scenarios such as comments on a blog, we would like to choose performance over the reliability as users are generally not too concerned if their comments appear eventually after some delay.

3

Consistent Hashing

3.1 Introduction

In cricket, it's important that players don't miss catching the ball on the field as it may cost them losing the game. Let's assume that a team has a catching practice session where players are placed on a circle, and a machine throws balls on the circle's boundary. The players have to follow the guideline that the one who is closest to the landing place of the ball (in a clockwise direction) is supposed to catch the ball. This should hold true even if someone goes for a drink break.

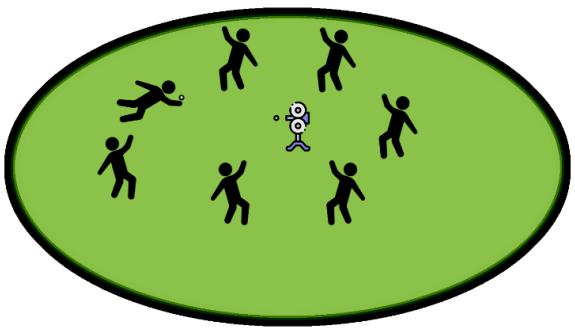


Figure 3.1: Consistent Hashing - cricket analogy

Using this analogy, one can develop an intuition of consistent hashing by treating play-

ers as servers and the balls which the players are supposed to catch as web-requests. The guideline of catching the closest ball in a clockwise direction can be referred to as Consistent Hashing. Now, let's try to understand the need for consistent hashing.

In traditional hash tables, objects are mapped using their identifiers to a specific value by a hash function, which is then used to store the object in any one of hash slots, as shown in following Fig 3.2.

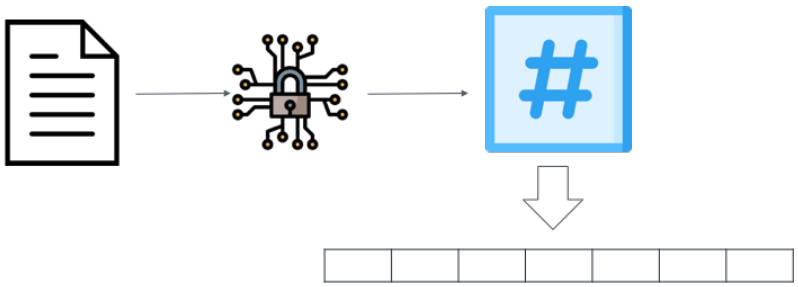


Figure 3.2: Hash Function generating hashes of multiple objects

Any change in the number of hash slots causes nearly all the keys to be remapped, which can be a costly operation. In Fig .3.3, we have shown how all the keys are remapped to the modified hash slots (i.e., after being changed from 7 to 6).

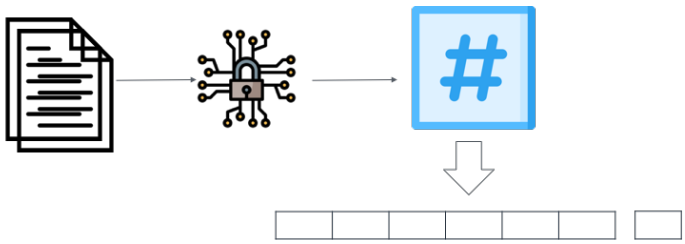


Figure 3.3: All the objects get remapped when the number of hash slots change

However, using consistent hashing whenever the hash table is resized only a small portion of the keys are remapped. Consistent hashing maps objects to the same cache machine, as far as possible. It means when a cache machine is added, it takes its share of objects from all the other cache machines, and when it is removed, its objects are shared among the remaining machines. The main idea behind the consistent hashing algorithm is to associate each cache with one or more hash value intervals where the interval boundaries are determined by calculating the hash of each cache identifier. If the cache is removed, its range is taken over by a cache with a next interval. All the remaining caches are unchanged.

3.1.1 Historical Background

The term "consistent hashing" was introduced in an academic paper from 1997 as a way of distributing requests among a changing population of Web servers. The authors of this paper founded Akamai Technologies a year later, which gave birth to the content delivery network industry.

3.2 Consistent Hashing

We use consistent hashing to distribute the objects across multiple nodes. The underlying hashing algorithm map the objects and the nodes in the same circular search range using a hash-function, as shown in the image below.

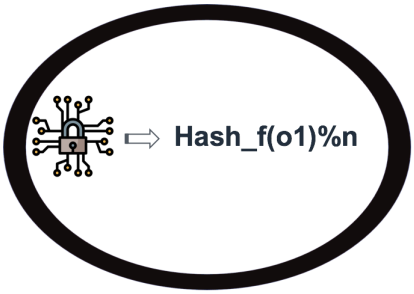


Figure 3.4: Fig3. Circular search range for hashing keys and nodes

The nodes are mapped on this circular range by hashing the *nodeIds* in the search range and then doing a mod over the search range. For instance, let's say the nodeId is *n1* and we are using a hash function *h1(k)* over the search range *0...n-1*, then the value on the circular search range on which the node *n1* will be mapped to will be: *h1(n1) % n*. Let's consider we have five nodes (n1 to n5), and we use the same logic to map them over a circular search range, and finally we map all the nodes as shown in Fig 3.5 below.

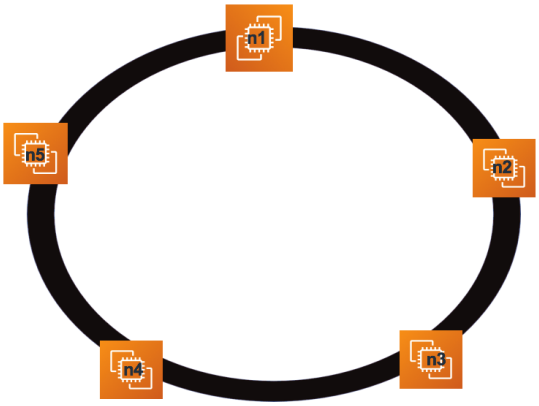


Figure 3.5: Fig4. Nodes mapped over the circular search range

Now we need to distribute objects across these nodes using the same logic for mapping the nodes over the circular ring. Let's assume we have five objects (o1...o5) which we distribute over the same ring in Fig 3.6 using the function: h1(o1) % n.

In Fig 3.6, we can see that both the nodes and the objects are mapped over the same circular range in a way that object o5 is mapped to node n1, o1 to n2, o2 to n3, o3 to n4, o4 to n5 and o5 to n1 is a clockwise fashion. It seems from this example that the objects are uniformly distributed over the nodes. However, things get tricky when nodes go down (the significant advantage of consistent hashing) as the distribution of objects may get skewed.

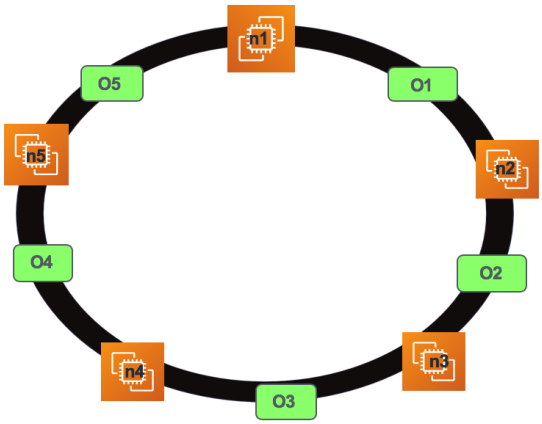


Figure 3.6: Fig5. Nodes and objects are mapped over the same circular search range

Let's assume that nodes *n2* and *n3* goes down. In such a scenario, objects empho1, o2, and *o3* are re-mapped to *n4, o4* to *n5* and *o5* to *n1*, leading to a skewed distribution as shown in Fig 3.7 below.

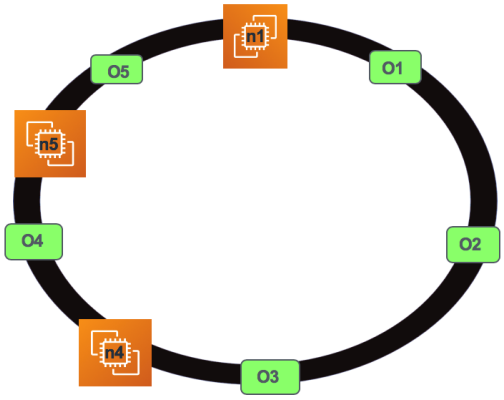


Figure 3.7: Fig6. Skewed distribution of objects when n2 and n3 dies

There are several ways to solve the problem of a skewed distribution of objects across nodes. One possible way is to use multiple hash functions to map nodes on the circular ring resulting in multiple instances of a node on the ring. Another approach can be to replicate the objects across nodes, as explained in the distributed caching implementation below.

3.3 Consistent Hashing Applications and Simulations

In this section, we have provided a brief overview of some well-known applications of consistent hashing along with their simulations.

3.3.1 Load Balancing

Load Balancers are responsible for distributing the user requests amongst application servers. In Fig 3.8, we have shown the role of load balancers to help our readers develop an intuition

of it.. This topic will be covered in detail in further chapters.

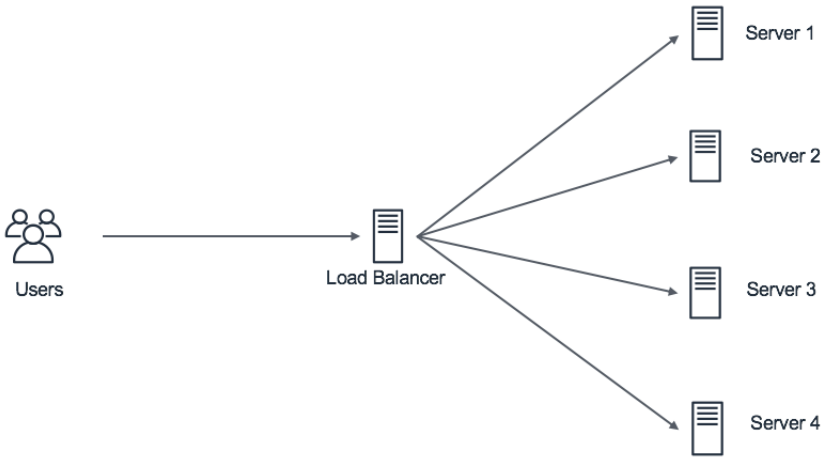


Figure 3.8: Fig7. Load balancer distributing users requests amongst servers

Brain Exercise: Let's assume you are designing a system which can distribute the requests amongst multiple servers. What logic will you apply to distribute the requests amongst the servers?

We recommend you to think of a solution to the brain exercise before reading further.

There are multiple ways of distributing the load amongst servers. Consistent hashing is a commonly used approach in load balancers using which web requests are distributed across servers. Consistent Hashing is applied by mapping the requestIds of web-requests and the serverIds of the application servers on the same circular search range. When a request comes to the load balancer it uses the unique identifier of the web-requests to find the location of the request on the edge of the circle; after that, the system walks around the ring until it encounters a server. In case a server becomes unavailable, then the requests which would have mapped to that server get redirected to the next server on the circle. When a server is added a similar process gets executed and the requests which were initially directed to the server at the next higher angle will start getting directed to the new server. Sample code for simulating a load balancing system using consistent hashing can be found here.

```
class LoadBalancingImpl<T>
{

    private HashFunction hashFunction;

    public LoadBalancingImpl(HashFunction hashFunction)
    {
```

```
        this.hashFunction = hashFunction;
    }

    private SortedMap<Integer, T> circle = new TreeMap<>();


    public void add(T node)
    {
        circle.put(hashFunction.hashCode(node), node);
    }

    public void remove(T node)
    {
        circle.remove(hashFunction.hashCode(node));

    }

    public T getServer(T key)
    {
        if(circle.isEmpty())
        {
            return null;
        }
        int hash = hashFunction.hashCode(key);
        if(!circle.containsKey(hash))
        {
            SortedMap<Integer, T> tailMap =
                    circle.tailMap(hash);
            hash = tailMap.isEmpty() ?
                    circle.firstKey() : tailMap.firstKey();
        }
        return circle.get(hash);
    }
}

 interface HashFunction<T>
 {

    public int hashCode(T t);
```

```
}
 class StringHashFunction implements HashFunction<String>
 {

    public int hashCode(String s)
    {
        int hash = 7;
        for (int i = 0; i < s.length(); i++)
        {
            hash = hash*31 + s.charAt(i);
        }
        return hash;
    }
}

public class LoadBalancingSimulation
{

    public static void main(String[] args)
    {

        LoadBalancingImpl<String> loadBalancing = new LoadBalancingImpl<>
        loadBalancing.add("A");
        loadBalancing.add("D");
        loadBalancing.add("H");
        loadBalancing.add("T");

        System.out.println(loadBalancing.getServer("Z"));
    }
}
```

3.3.2 Distributed caching

Distributed Cache is an extension of cache which spans multiple servers so that it can scale horizontally. In Fig 3.9, it shows distributed cache which spans over multiple servers and communicate amongst each other to attain high availability and reliability.

Brain Exercise: How will you distribute the objects across multiple servers? What will you do to attain high availability?

We recommend you to think of a solution to the brain exercise before reading

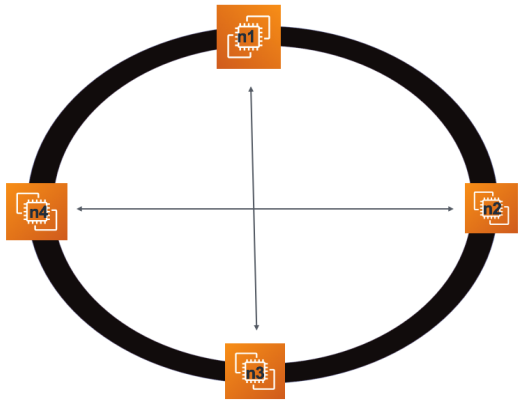


Figure 3.9: Fig8. Distributed server used for caching

further.

Often, consistent hashing is used to enable distributed caching where the caches are spread across multiple hosts. The replicas of a cache are created on multiple servers so that the cache can be retrieved even if one of the servers fail. There are numerous ways of selecting the servers on which cache replicas can be stored. In Fig 3.10, it shows one way of storing replicas by creating virtual nodes using multiple hash functions which map over the same circular range. The following image, *n1* is mapped using the hash function *h0* and *n1'* is mapped using *h1*. Similarly, *n2* is mapped using the hash function h0 and *n1'* is mapped using h1 and so forth. Now, let's say that an object *o1* is allocated to the node *n2* on the same circular range using a hash function *h2* and *n2* dies. This will result in *o1* getting mapped to *n1'*, thus preventing an skewed distribution of replicas (as now node *n3* would not be overwhelmed with requests for objects cached on *n2* before it went down).

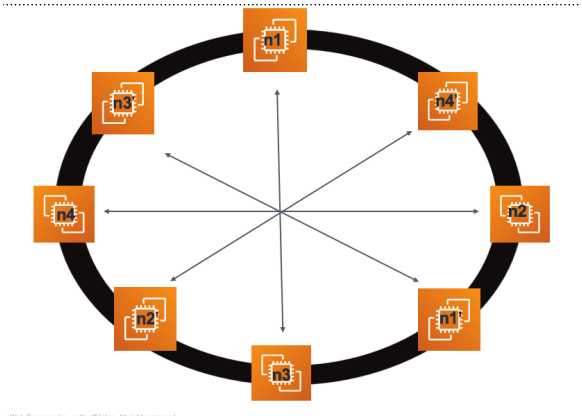


Figure 3.10: Creating virtual nodes for storing replicas

Another simple approach to create replicas of the cache can be to choose the next k (example 2) successors of the server (on which the request was initially directed) to do so. A sample code which simulates this approach can be found here. However, there can be more robust strategies for replicating caches; one such method can be the usage of finger table as in Chord protocol.

3.4 Fitness check up

1. What are the implications of the skewed distribution of objects across nodes?

   (a) Hot Partitions

   (b) Network outage

Correct Answer: a

Explanation: Most of the fetch requests for the objects will be directed to the nodes having a majority of objects resulting in hot partitions

4

Relational VS No SQL

4.1 Introduction

The comparison between RDS and NoSQL data-store can be clearly understood using a real-life analogy of living in downtown vs. living in the suburbs. Due to the scarcity of land in the city, the development is often done vertically in the form of skyscrapers, whereas, in the suburbs, there is an abundance of land, so new constructions are done on land (aka horizontal growth). Same as downtown (i.e. vertical growth), relational databases are generally scaled up by adding more computational resources to the existing hardware (which is costly and soon hits the threshold), whereas, NoSQL stores can be scaled out by simply adding more commodity hardware which are comparatively a lot cheaper.

Figure 4.1: living in downtown vs. suburbs

4.2 Relational Databases

Relational databases have had a long period of dominance and have been the default choice for enterprise data storage, even though there have been alternative database technologies such as object databases since the 1990s. Some of the popular relational databases are: Oracle, MySQL, Postgres and so forth. These databases use structured query language (SQL) for defining and manipulating data and structure of the data stored in them is determined using pre-defined schemas. This implies that all the data needs to follow the same structure and any change in the structure may be disruptive.

Relational databases provide multiple advantages due to ACID properties. However, they have had some major issues from their early days, some of them are listed below.

- **Impedance Mismatch** It refers to the different representations of the relational model and in-memory databases which require translations. The issue lies with the foundation of relational databases which are: i) tuple or row, which is a set of name-value pairs and ii) relation or table, which is a set of tuples. The values in a relational tuple have to unaffected - they can't contain any structure, such as nested record or list. Therefore, a richer memory data structure had to be translated into multiple tables, as shown in Fig 4.1.

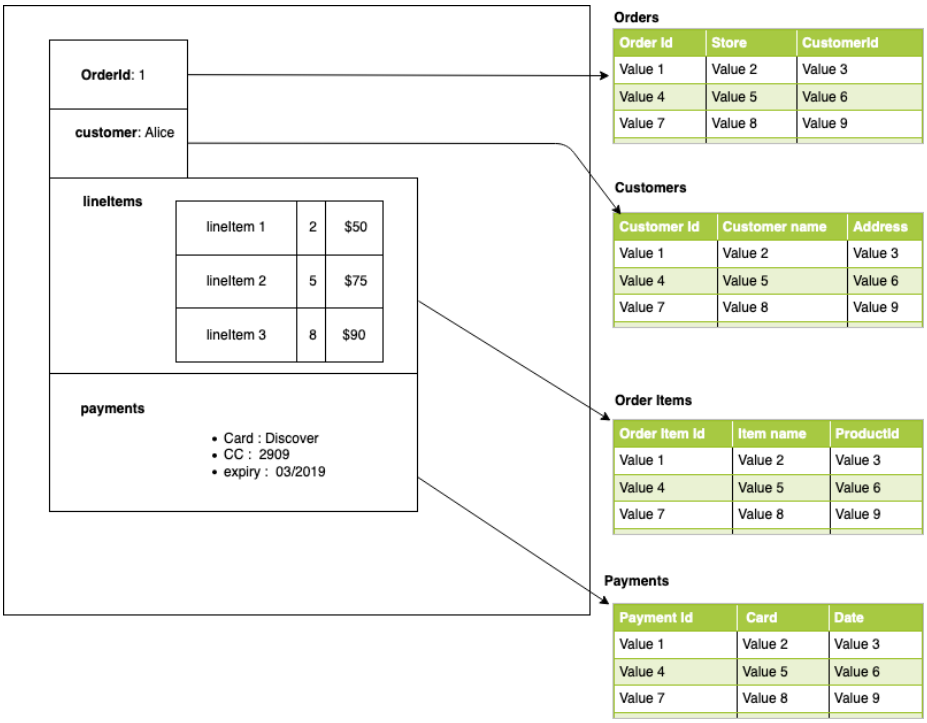


Figure 4.2: An order data structure in split into relations and tuples

The frustration with impedance mismatch has been minimized to a great extent by using object relational mapping frameworks - such as Hibernate and Mybatis. However, the mapping problem with relational databases still remains unsolved.

- **Running on clusters** With the advent of the Internet, there is an apparent increase in the demand for data storage. In order to combat this increased demand, we have two choices: Scaling Up and Scaling Out. Given that scaling up requires bigger machines and more significant bills, the option we had was to scale out by using a lot of small machines in a cluster using commodity hardware. Often, relational databases which run on clusters have a single point of failure which make them inefficient in running on large clusters.

4.3 NoSQL Databases

The term NoSQL database was first coined by Carlo Strozzi in the late 90s when he created an open source relational database: Strozzi NoSQL. The term NoSQL was derived from the fact that the database got manipulated using shell scripts rather than SQL as a query language. However, the usage of the term NoSQL kicked off when Johan Oskarsson used it as a Twitter hashtag for a meetup on June 11, 2009, in San Francisco. The emergence of NoSQL databases is based on the massive data storage needs of the early 21st century which also led to the rise of polyglot persistence - using different data stores in different circumstances based on the understanding of the nature of data. NoSQL databases share some common characteristics which are listed below.

- **Running on Clusters** Most NoSQL databases are efficient at running on large clusters. However, some NoSQL databases such as Graph database aren't strongly geared towards running on clusters but offers a range of different data models which are ideal for dealing with complicated relationships.

- **Schema-less**: These databases generally operate without a schema, allowing you to freely add fields to database records without having the need to change the data-storage structure.

- **Open Source**: Often, NoSQL databases are open-source projects. Nevertheless, NoSQL has been frequently applied to closed-source systems.

4.4 Comparison chart

We have provided comparison between relational and NoSQL data stores on some important aspects below.

Sl. No.	Description	Relational Database	NoSQL Database
1.	**Schema**	Follow a pre-defined schema for structured data	Follow a dynamic schema for unstructured data
2.	**Scalability**	Scale Up – Increase hardware configurations such as memory, CPU and so forth.	Scale Out – Adding more commodity hardware
3.	**Query Language**	Use structured query language (SQL)	Aggregate oriented NoSQL databases don't use standard query language. Graph based NoSQL stores do use query language for traversal (e.g. Gremlin)
4.	**Storing Hierarchical Data**	Not fit for storing hierarchical data	Good for storing hierarchical data as it stores data in a key-value pair way
5.	**Handling complex transactions**	Best fit for handling complex transactions spanning over different data entities as it guarantees atomicity	Doesn't support transactions spanning over multiple aggregates
6.	**Support for complex queries**	Supports queries spanning over multiple data entities using joins and foreign keys	Complex queries need to be implemented in the application logic

Figure 4.3: Comparison Table between Relational and NoSQL databases

4.5 Conclusion

The growth in NoSQL development is not only the result of big data running on clusters, but also, it's the age-old frustration with impedance mismatch problem of relational databases. The primary reasons for considering NoSQL are i) handling significant data access that requires a cluster ii) enhancing application development productivity by using more adequate data access patterns. However, relational databases are still the choice for data which isn't changing frequently and has a lesser hierarchical depth.

4.5.1 Fitness Check up

1. Suppose we need to build a database for storing different customer features of a social-media application. What type of database will you choose?

   - Relational
   - NoSQL

   Correct Answer: b

   Explanation: It's apparent that customer features are dynamic and new customer features may get added/removed quite frequently. A NoSQL data store will be an ideal fit for such a use-case as it provides the flexibility to easily maintain the ever-changing customer features.

2. Your team inherited a small legacy application organization where the data is structured and unchanging and is stored in a relational database. Will you migrate the

data over to a NoSQL data store?

- No
- Yes

Correct Answer: a

Explanation: If you are working with data which is consistent and doesn't need to support a variety of data-types and high volume. In such cases, you may be better off with a relational database.

5

Types of NoSQL databases

5.1 Introduction

The decision to choose a data-store is somewhat similar to buying a car where one needs to choose from a variety of available options. There are several parameters which individuals buying a car take into account. People for whom space is of utmost importance (generally having families) tend to buy larger SUVs. On the other hand, if someone wants to optimize fuel cost is most probable to opt for a hatchback or a sedan. Similarly, the choice of a data-store depends to a great extent on application requirements and the user query patterns. If the application's data contains multi-level relationships, then we may choose a graph-oriented database. On the other end, if the application requires running more analytical queries, then we may opt for a column-oriented database. Let's try to dive deeper into the trade-offs associated with using different types of NoSQL databases.

Figure 5.1: An analogy to explain different types of NoSQL data stores

5.2 Overview

The model using which a database organizes data is referred to as “Data Model”. Before the advent of NoSQL data-stores, relational databases used to store data by normalizing the data-entities using entity relationships rather than query patterns. However, NoSQL data stores enabled us to model data in a way which is optimal for the query pattern of the application users. In the NoSQL space, there are four major categories of data models: key-value, document, column-family, and graph; each of them catered to specific user needs. The common characteristic shared across the first three data models (i.e. key-value, document, column-family) is called aggregate orientation.

5.3 Aggregate Oriented Databases

An aggregate can be defined as a collection of related objects that can be treated as a unit for data manipulation. The aggregate boundaries are decided by how data gets manipulated. For example, a customer purchase order makes a suitable aggregate when a customer is reviewing or making the order. However, it’s not very efficient if we have to determine the product sales history.

Unlike relational databases, aggregate-oriented databases don’t support ACID transactions that extend over multiple aggregates. Instead, they allow atomicity at a single aggregate level. The application needs to handle any atomic operation which extends across aggregates. Let’s dive deeper into each of the aggregate-oriented databases and their structures.

5.3.1 Key-Value Data Model

Key-Value datastores can be understood as a giant hash table which can be queried using a key and an efficient hash function. This functionality allows key-value data stores to perform data look-ups and manipulation in an extremely scalable fashion on large clusters. However, the key-value stores are not optimized for cases where querying based on aggregate value is essential. Some of the use-cases where key-value stores are a good fit for data-storage are Web session, User profiles, and Shopping cart. However, there are few scenarios in which such data stores aren’t best suited. For instance, transactions spanning multiple keys, having correlated keys in the data store and querying using the value part of the key-value pairs. Few examples of key-value data store include Amazon DynamoDB, Riak, and so forth.

5.3.2 Document Data Model

Document data-model stores documents (collection of named fields and data) as aggregates, conceptually similar to the key-value store. Nevertheless, document databases expose their aggregates, enabling an application to query and filter data by using the values in these fields. With a document database, we can submit queries to the database based on the fields in the aggregate, we can retrieve part of the aggregate rather than the whole thing. Unlike RDMS, where every tuple in relation had to follow the same schema documents in this data model is that they can belong to the same collection and yet have different schemas. Such data-stores are suitable for use cases such as orders in e-commerce applications, web application visitors for real-time analytics, managing documents, and profiles in blogging platforms. Similar to key-value data stores, transactions, and querying across documents aren't best suited for such data stores.

The get and put for document-based databases will be similar to key-value based databases. However, unlike key-value based databases, we can update only a subsection of the document. Some of the popular document databases are MongoDB, Amazon DocumentDB, and Couchbase.

Choice between Key-Value and Document Data Stores

The data and application needs can be used to make the trade-off between key-value and document databases. Both key-value and document databases are excellent choices for many database applications. If query patterns and data structures are relatively simple, and the entire aggregate needs to fetched, key-value databases are the right choice. As the complexity of queries and entities increase, document databases become a better option, although the line between key-value and document data stores is very thin.

5.3.3 Columnar Data Model

Unlike other databases where data is stored by serializing on rows, columnar databases store data by serializing on columns instead. Let's try to understand it further using an example shown in the table below.

Table 5.1: Sample data set in tabular format

Row ID	Attribute_1	Attribute_2	Attribute_3
R1	A1	A2	A3
R2	B1	B2	B3

If the data is serialized on the rows then all the columns of a row will be stored together on the disk, and the structure on disk may look something similar to: **R1, [A1, A2, A3] |**

R2, [B1, B2, B3]. However, in case of columnar databases the data is stored by keeping the columns of multiple rows together and the structure on the disk may look like this: **[(R1, A1), (R2, B1)], [(R1, A2), (R2, B2)], [(R1, A3), (R2, B3)]**.

The columnar databases further optimize on data-storage by serializing group of related columns (aka column-families) together. Such data stores are often used in OLAP scenarios where the writes are rare and reads are targeted towards reading a few columns of many rows at once. By storing all the records of related fields together (obtained by grouping the associated columns together into column-families) as aggregates, columnar data-stores can perform analytical queries on those fields much quicker compared to row-based data stores. An excellent example of a columnar data store is Amazon Redshift, which is a cloud hosted data warehouse solution optimized for data analysis over the vast quantity of data.

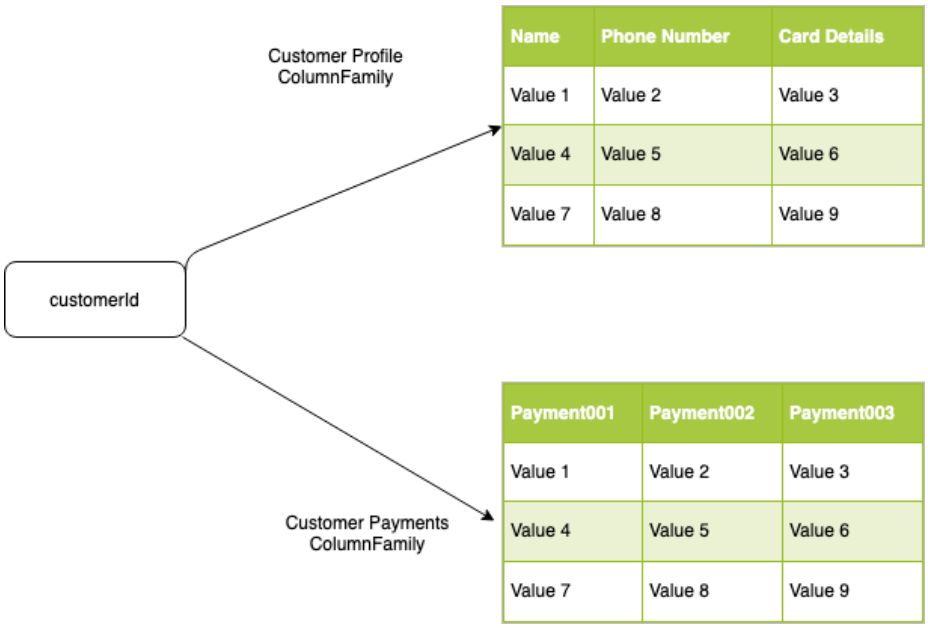


Figure 5.2: Data model used in column-oriented databases

In the above data-model, there are two column families: user-profile and payments. The user-profile column-family has columns related to the user-profile and the payments column-family consist of columns pertaining to payments, thus making the queries for columns within a column-family more efficient as the columns within a column-family are stored together. Such data-models aren't generally suited for OLTP transactions where reads span across multiple columns as in those scenarios different aggregates have to be queried. Given that columnar databases are most suitable for OLAP scenarios, we may want to use the standard SQL queries to fetch analytical data.

5.4 Graph Data Model

The relational databases use foreign keys to store relationships between data entities using foreign keys. However, the limitation of relational database has been that the connections are computed at query time and so it's not able to scale efficiently beyond relationships which are generally more than two levels deep.

The need for data models which can store more complex relationships came up with the advent of social networks, which played a crucial role in the popularity of graph databases. In order to save and retrieve non-trivial relationships from highly-related data, the graph-based data model persisted the data entities as nodes and the relationship between those entities as edges. Traversing stored relationships between entities made the traversal of relationship links quite cheaper compared to computing them at query time as in relational databases.

Such graph-based databases are useful in social networking applications, location-based services, or any application which requires storing complex relationships. However, such databases don't perform well in analytical solutions where updates are applied to properties of all the nodes. The data lookups performance is dependent on the access of one node from another. This performance is enhanced by storing the physical RAM address of adjacent nodes and caching links to directly-related nodes.

In graph data model, the data entities and relationship information are stored as properties in nodes and edges respectively. The query on the graph structure is called traversal which is made efficient by storing the properties as indexes. Let's dive deeper by evaluating few graph queries on the graph data-model in Fig 5.2, by assuming that all the nodes are already added into an index named *nodeIndex*.

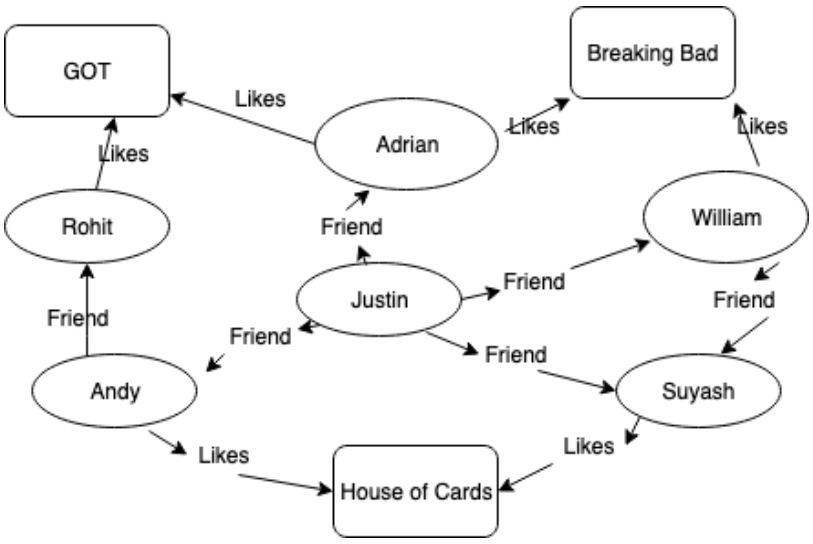


Figure 5.3: A Graph Data Model of favourite TV series of a group of people

```
//Fetch the node containing information about Justin from node index
```

```
Node justin = nodeIndex.get("name", "Justin");
```

Query 1: All friends of Justin

Here, we will fetch all the relationships of justin having edge direction as Direction.OUTGOING and we can filter the relationships having type as Friend.

```
for(Relationship relationship:
justin.getRelationships(OUTGOING, FRIEND)) {
    friendsOfJustin.add(relationship.getEndNode());
}
```

Query 2: Find all people who like House Of Cards and GOT and are friends with each other

Here, we will fetch all the relationships of houseOfCards having edge direction as Direction.INGOING and we can filter the relationships having type as Likes.

```
//Create a list of people who like House of cards
Node houseOfCards = nodeIndex.get("name", "House of Cards");

for(Relationship relationship:
houseOfCards.getRelationships(INGOING,LIKES))
{
    peopleLikingHouseOfCards.add(relationship.getStartNode());
}

//Create a list of people who like House of cards
Node got = nodeIndex.get("name", "GOT");

for(Relationship relationship:
got.getRelationships(INGOING,LIKES))
{
    peopleLikingGOT.add(relationship.getStartNode());
    }

//Get list of friends of people liking House of Cards
for(People people: peopleLikingHouseOfCards)
{
    for(Relationship relationship:
    people.getRelationships(OUTGOING, FRIEND))
```

```
    {
        friendsOfPeopleLikingHouseOfCards.add
        (relationship.getStartNode());
    }
}

//Get list of friends of people liking GOT
for(People people: peopleLikingGOT)
{
    for(Relationship relationship:
    people.getRelationships(OUTGOING, FRIEND))
    {
        friendsOfPeopleLikingGOT.add(relationship.getStartNode());
    }
}
```

At this point, we can apply a boilerplate logic to find the people who like GOT and House of Cards and are friends with each other.

5.5 Case Study Databases

In the recent past, there has been an increasing need to store, query and analyze time series data. Some of the primary use-cases for such databases are: IoT (e.g. consumer wearables such as Fitbit, sensors in oil and gas factories, etc.) and DevOps (e.g. monitoring application logs, customer events and so forth). The requirements which makes time-series database unique include: efficient life-cycle management which ensures that we don't store data that are no longer needed and summarization of large range scan of records so that the summarized results can be shown to users with minimal latency.

Timestamp	Price	Ticker Symbol	Market
2019-05-18T15:13:00Z	$175.80	AAPL	NASDAQ
2019-05-18T15:03:00Z	$1680.09	AMZN	NYSE
2019-05-18T14:55:00Z	$125.10	MSFT	NASDAQ
2019-05-18T14:23:00Z	$123.89	MSFT	NYSE
2019-05-18T13:15:00Z	$1702.10	AMZN	NASDAQ
2019-05-18T13:05:00Z	$174.50	AAPL	NYSE

Figure 5.4: Fictional Stock Price Data

InfluxDB is one of the most widely used time-series database and provides several built-in features for manipulating and analyzing time-series data which otherwise would be non-trivial to implement with other databases such as MongoDB, Cassandra and so forth. InfluxDB came up with its own textual format which is called The Line Protocol, which comprises of four components: measurement, tagset, fieldset and timestamp. The format followed by the protocol is: measurement, tagset fieldset timestamp; an example of stock-price data in this format can be seen below.

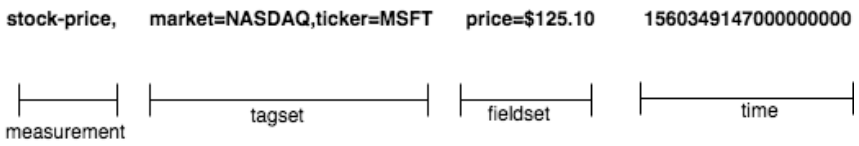


Figure 5.5: Sample Stock Price Data represented in Line Protocol format

We have represented the fictional stock-price data in a graphical format below. In the InfluxDB eco-system, the collection of points for a combination of tag-set and measurement is called a series; e.g., the red line for measurement "stock-price" and tag-set *"ticker=AMZN, market=NYSE"* in Fig 4. The data in InfluxDB is indexed on tag-set and timestamp. The timestamp acts as a unique identifier of a point in the series. These indexes help in reducing the latency of the time-based queries such as finding the maximum stock-price for tag-set ticker=AMZN, market=NYSE in a specific period.

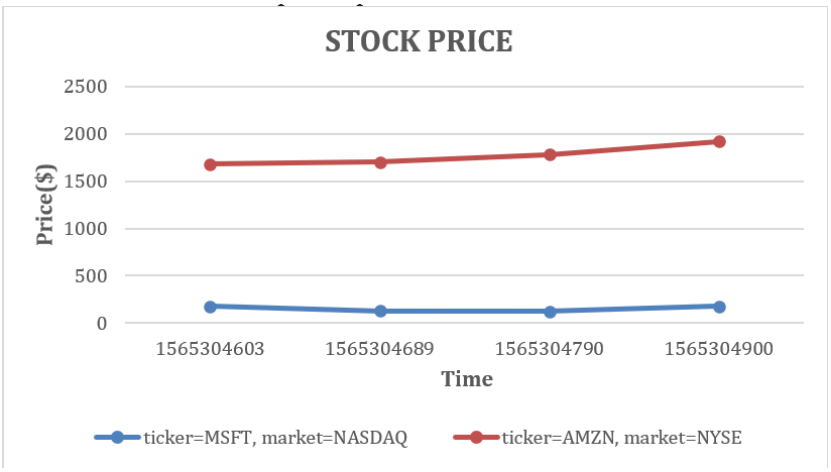


Figure 5.6: Graphical representation of fictional Stock Price Data

5.6 Conclusion

In the current scenario, applications are no longer tied to a single database and have the flexibility to use different data-stores depending upon the data storage needs. The rise of SOA has also played a crucial role in ensuring that applications use the data-store best fit

for the data-access pattern. For instance, within an e-commerce platform, we may have different micro-services for each of its components, and each of those micro-services may use entirely different data-stores depending upon the use-case.

5.6.1 Fitness Check up

1. We need to build a database for storing multiple attributes (e.g. date-of-birth, purchases, membership-status and so forth) of a customer for an application where the attribute values may change frequently. Which data store would you recommend for such applications?

   (a) Key-Value Data Model

   (b) Document Data Model

   (c) Graph-Based Data Model

   Given that all the attributes of a customer can be clubbed together, we would like to choose an aggregate-oriented database. Now, we don't want to fetch the complete aggregate while updating partial attribute values. So, we may want to choose a document data-model over key-value model.

2. Suppose we are assigned a task to create a cold data storage solution for analytical purpose of a firm. What kind of a database would you choose?

   (a) Column-Based

   (b) Graph-Based

   (c) Key-Value

   Often in analytical queries, multiple rows of the same columns are accessed together. The column-based databases store data on the disk by grouping columns of multiple rows adjacent to each other, thus, resulting in reduced overall IO costs making it the obvious choice in this scenario.

3. We need to build an application which can find the most optimal path for customers to go from one city to another using minimum number of hops. Which data-model will be best fit for users to find the most optimal path?

   (a) Column-Based

   (b) Graph-Based

   (c) Key-Value

We can easily transform the application data into a graph-based data model which provides built in functionality to support traversal queries. In these scenarios, aggregate-oriented databases are not a good fit as traversing between aggregates will lead to sub-optimal performance.

Part II

System Design Case Studies

6 Instagram

Note: This is our prototype on designing a photo-sharing system such as Instagram for users to upload and share media files. We have created this design based on our research going through Instagram Engineering tech-talks. These talks are quite informative and detailed, and for the same reason, we have taken inspiration from those talks. We have provided the list of the tech talks we have referenced here

6.1 Introduction

6.1.1 Problem statement

In this chapter we will be designing a photo-sharing platform similar to Instagram where users can upload their photos and share it with their followers. Subsequently, the users will be able to view personalized feeds containing posts from all the other users that they follow.

6.1.2 Gathering Requirements

In scope The application should be able to support the following requirements.

- Users should be able to upload photos and view the photos they have uploaded.
- Users should be able to follow other users.

- Users can view feeds containing posts from the users they follow.
- Users should be able to like and comment the posts.

Out of scope

- Sending and receiving messages from other users. We have covered it in our article on designing WhatsApp.
- Generating machine learning based personalized recommendations to discover new people, photos, videos, and stories relevant one's interest.

6.2 Detailed design

6.2.1 Architecture

The high-level design consists of two important functions; one is to upload photos and other is to view or search photos. Our service will need storage to store the photos and some database servers to store metadata information about the photos

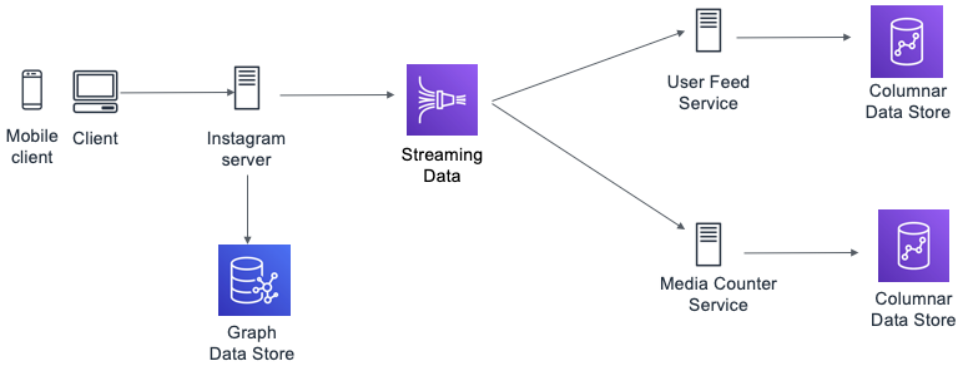


Figure 6.1: Architecture of photo sharing application

Therefore, when the server receives a request for an action (post, like etc.) from a client it performs two parallel operations:

1. Persisting the action in the data store
2. Publishes the action in a streaming data store for a pub-sub model.

It is followed by, the various services (such as, User Feed Service, Media Counter Service) read the actions from the streaming data store and performs their specific tasks. The streaming data store makes the system extensible to support other use-cases (e.g. media search index, locations search index, and so forth) in future.

FUN FACTS: In this talk, Rodrigo Schmidt, director of engineering at Instagram talks about the different challenges they have faced in scaling the data infrastructure at Instagram.

6.2.2 System components

The system will comprise of several micro-services each performing a separate task. We will use a graph database such as Neo4j to store the information. The reason we have chosen a graph data-model is that our data will contain complex relationships between data entities such as users, posts, and comments as nodes of the graph. After that, we will use edges of the graph to store relationships such as follows, likes, comments, and so forth. Additionally, we can use columnar databases like Cassandra to store information like user feeds, activities, and counters.

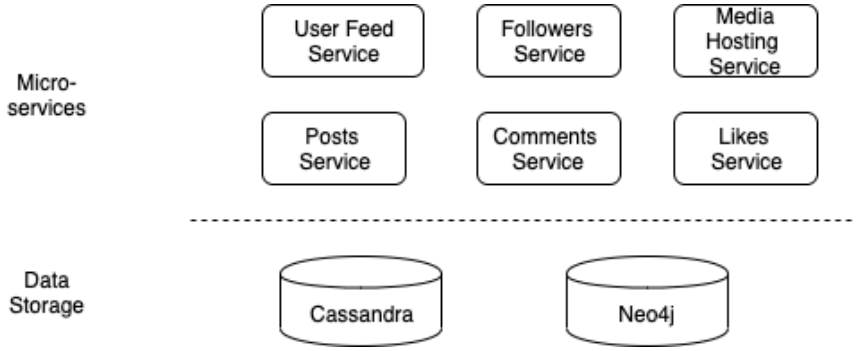


Figure 6.2: High level system components

6.3 Component Design

6.3.1 Posting on Instagram

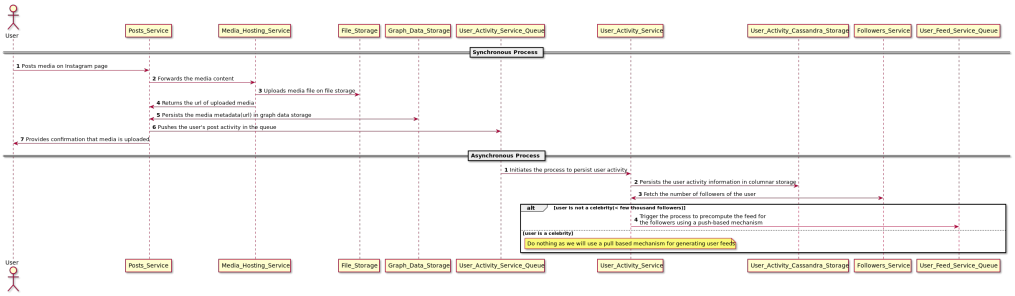


Figure 6.3: Synchronous and Asynchronous process for posting on Instagram

There are two major processes which gets executed when a user posts a photo on Instagram. The synchronous process is the first step, which is responsible for uploading image content on file storage, persisting the media metadata in graph data-storage, returning

the confirmation message to the user and triggering the process to update the user activity. The second process occurs asynchronously by persisting user activity in a columnar data-storage (Cassandra) and triggering the process to pre-compute the feed of followers of non-celebrity users (having few thousand followers). We don't pre-compute feeds for celebrity users (have 1M+ followers) as the process to fan-out the feeds to all the followers will be extremely compute and I/O intensive.

6.3.2 API Design

We have provided the API design of posting an image on Instagram below. We will send the file and data over in one request using the multipart/form-data content type. The MultiPart/Form-Data contains a series of parts. Each part is expected to contain a content-disposition header [RFC 2183] where the disposition type is "form-data".

URL:

```
POST /users/<user_id>/posts
```

Sample Request Body: Content-Type: *multipart/form-data*; boundary=ExampleFormBoundary

```
--ExampleFormBoundary
Content-Disposition: form-data; name="file"; filename="test.png"
Content-Type: image/png

{image file data bytes}
--ExampleFormBoundary--
```

Sample Response:

```
{
    "type": "ImageFile",
    "id": "667",
    "createdAt": "1466623229",
    "createdBy": "9",
    "name": "test.png",
    "updatedAt": "1466623230",
    "updatedBy": "9",
    "fullImageUrl": "https://mybucket.s3.amazonaws.com/myfolder/test.jpg"
",
    "size": {
        "type": "Size",
        "width": "316",
        "height": "316"
```

```
    },
}
```

6.3.3 Precompute Feeds

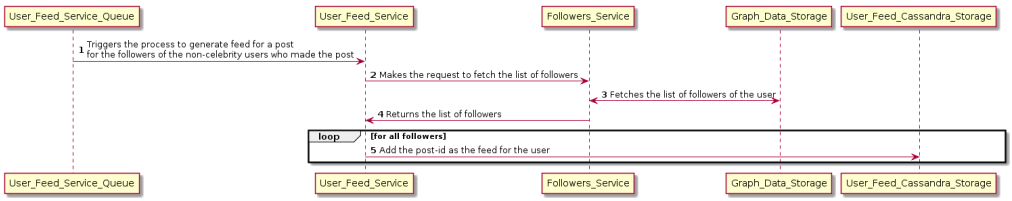


Figure 6.4: Precompute feeds from non celebrity users

This process gets executed when non-celebrity users make a post on Instagram. It is triggered when a message is added in the User Feed Service Queue. Once the message is added in the queue, the User Feed Service makes a call to the Followers Service to fetch the list of followers of the user. After that, the post gets added to the feed of all the followers in the columnar data storage.

6.3.4 Fetching User Feed

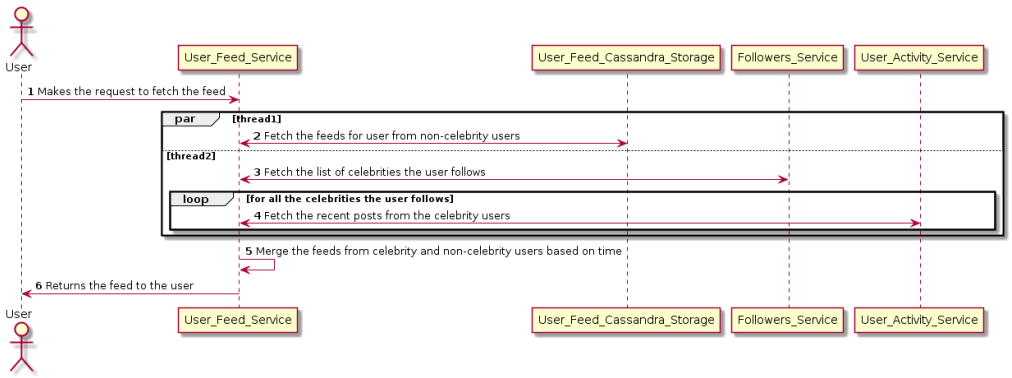


Figure 6.5: Sequence of operations involved in fetching user feed

When a user requests for feed then there will be two parallel threads involved in fetching the user feeds to optimize for latency. The first thread will fetch the feeds from non-celebrity users which the user follow. These feeds are populated by the fan-out mechanism described in the PreCompute Feeds section above. The second thread is responsible for fetching the feeds of celebrity users whom the user follow. After that, the User Feed Service will merge the feeds from celebrity and non-celebrity users and return the merged feeds to the user who requested the feed.

6.3.5 API Design

URL:

```
GET /users/<user_id>/feeds
```

Sample Response: The response below can be mapped directly to the graph data model mentioned in the next section.

```
}
\{
     {
     “feeds”: [
{
“postId”: “post001”,
“postOwnerId” : “user001”,
“postOwnerName” : “Bill Gates”,
“mediaURL” : “https://mybucket.s3.amazonaws.com/myfolder/test.jpg”,
“numberOfLikes” : 400000
“numberOfComments” : 19789
“comments”: [
{
“commenterUserId” : “user004”,
“commenterName” : “JeffBezos”
“comment” : “Amazing!”
“likes” : [
{
“likerId” : “user003”,
“likerUserName” : “Bob”
}
]
}
]

}
    ]
}
```

6.4 Data Models

6.4.1 Graph Data Model

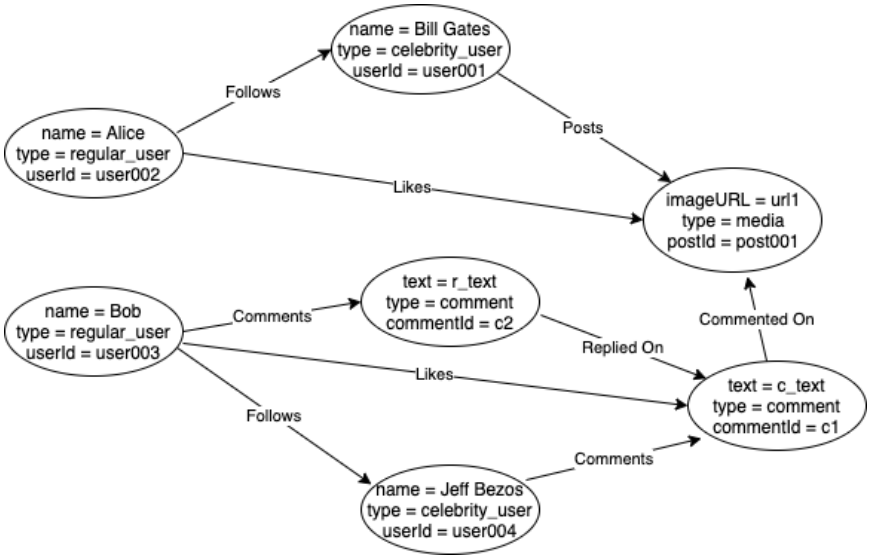


Figure 6.6: Graph representation of users, posts and comments

We can use a graph database such as Neo4j which stores data-entities such as user information, posts, comments, and so forth as nodes in the graph. The edges between the nodes are used to store the relationship between data entities such as followers, posts, comments, likes, and replies. All the nodes are added to an index called *nodeIndex* for faster lookups. We have chosen this NoSQL based solution over relational databases as it provides the scalability to have hierarchies which go beyond two levels and extensibility due to the schema-less behavior of NoSQL data storage.

Sample Queries supported by Graph Database

Fetch all the followers of Jeff Bezos

```
Node jeffBezos = nodeIndex.get("userId", "user004");
List<Node> jeffBezosFollowers = new ArrayList();

for (Relationship relationship:
jeffBezos.getRelationships(INGOING, FOLLOWS))
{
    jeffBezosFollowers.add(relationship.getStartNode());
}
```

Fetch all the posts of Bill Gates

```
Node billGates = nodeIndex.get("userId", "user001");
List<Node> billGatesPosts = new ArrayList();

for (Relationship relationship:
billGates.getRelationships(OUTGOING, POSTS)) {
    billGatesPosts.add(relationship.getEndNode());
}
```

Fetch all the posts of Bill Gates on which Jeff Bezos has commented

```
List<Node> commentsOnBillGatesPosts = new ArrayList<>();

for(Node billGatesPost : billGatesPosts) {
     for (Relationship relationship:
     billGates.getRelationships(INGOING, COMMENTED_ON)) {
commentsOnBillGatesPosts.add(relationship.getStartNode());
     }
}

List<Node> jeffBezosComments = new ArrayList();

for (Relationship relationship:
jeffBezos.getRelationships(OUTGOING, COMMENTS))
{
    jeffBezosComments.add(relationship.getEndNode());
}

List<Node> jeffBezosCommentsOnBillGatesPosts =
 commentsOnBillGatesPosts.intersect(jeffBezosComments);
```

6.4.2 Columnar Data Models

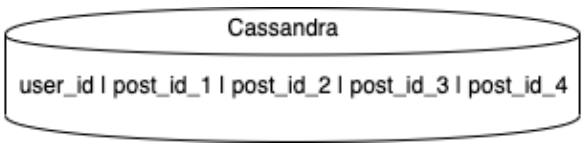


Figure 6.7: Columnar Data Model for user feed and activities

We will use columnar data storage such as Cassandra to store data entities like user feed and activities. Each row will contain feed/activity information of the user. We can also have a TTL based functionality to evict older posts. The data model will look something similar to:

```
User_id -> List<post_id>
```

FUN FACT: In this talk, Dikang Gu, a software engineer at Instagram core infra team has mentioned about how they use Cassandra to serve critical usecases, high scalability requirements, and some pain points.

6.4.3 Streaming Data Model

We can use cloud technologies such as Amazon Kinesis or Azure Stream Analytics for collecting, processing, and analyzing real-time, streaming data to get timely insights and react quickly to new information(e.g. a new like, comment, etc.). We have listed below the de-normalized form of some major streaming data entities and action.

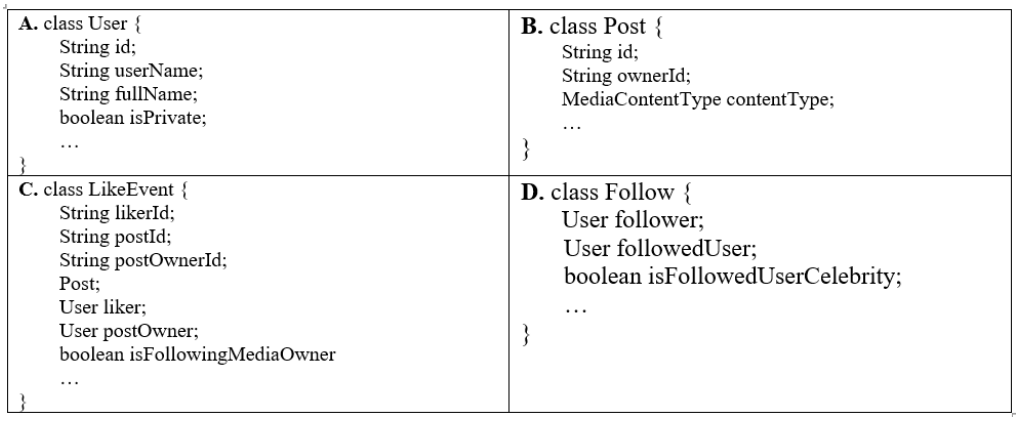


Figure 6.8: De-normalized major data-entities and actions

The data entities **A** and **B** above show the containers which contain denormalized information about the Users and their Posts. Subsequently, the data entities **C** and **D** denote the different actions which users may take. The entity C denotes the event where a user likes a post and entity D denotes the action when a user follows another user. These actions are read by the related micro-services from the stream and processed accordingly. For instance, the *LikeEvent* can be read by the *Media Counter Service* and is used to update the media count in the data storage.

6.5 Optimization

We will use a cache having an LRU based eviction policy for caching user feeds of active users. This will not only reduce the overall latency in displaying the user-feeds to users but

will also prevent re-computation of user-feeds.

Figure 6.9: LRU bases cache for storing user feeds of active users

There is a room for optimization in providing the best content in the user feeds. We can do this by ranking the new feeds (the ones generated after users last login) from those who the user follows. We can apply machine learning techniques to rank the user feeds by assigning scores to the individual feeds which would indicate the probability of click, like, comment and so forth. We can do this by representing each feed by a feature vector which contains information about the user, the feed and the interactions which the user has had with the people in the feed (e.g. whether the user had clicked/liked/commented on the previous feeds by the people in the story). It's apparent that the most important features for feed ranking will be related to social network. Some of the keys of understanding the user network are listed below.

- Who is the user a close follower of? For example, one user is a close follower of Elon Musk while another user can be a close follower of Gordon Ramsay.
- Whose photos the user always like?
- Whose links are most interesting to the user?

We can use deep neural networks which would take the several features (> 100K dense features) which we require for training the model. Those features will be passed through the n-fold layers, and will be used for predicting the probability of the different events (likes, comments, shares, etc.).

FUN FACT: In this talk, Lars Backstrom, VP of Engineering @ Facebook talks about the machine learning done to create personalized news feeds for users. He talks about the classical machine learning approach they used in the initial phases for personalizing News Feeds by using decision trees and logistic regression. He then goes to talk about the improvements they have observed in using neural networks.

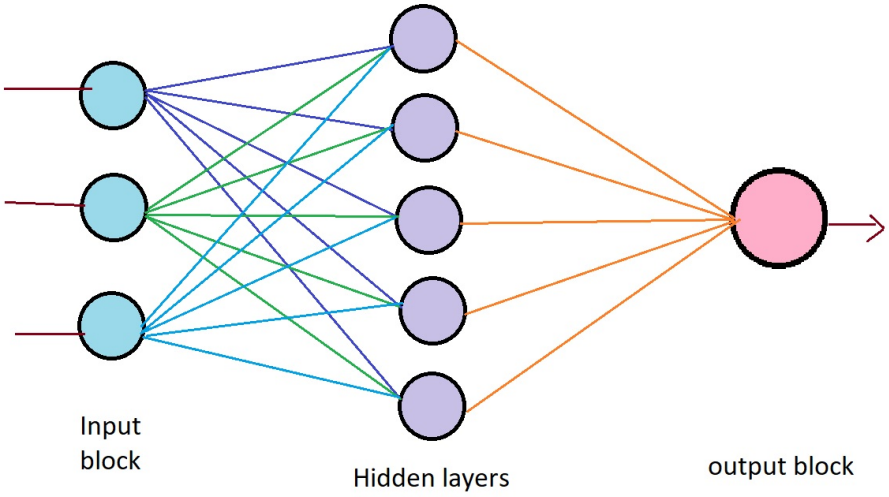


Figure 6.10: Block Diagram of Deep Neural Networks

7 Twitter

Note: : This is our prototype on designing a microblogging and social networking platform on which users interact with each other using messages known as “tweets”. Users can post, like, and retweet tweets for which notifications will be sent to the all the partaking users. We have created this design based on our research going through Twitter and Instagram Engineering tech-talks. These talks are quite informative and detailed, and for the same reason, we have taken inspiration from those talks.

7.1 Introduction

7.1.1 Problem Statement

In this chapter, we will design a microblogging platform similar to Twitter where users can post a tweet and it will be shared it with the user followers. Subsequently, the users will be able to view personalized feeds containing posts from all the other users that they follow.

7.1.2 Gathering Requirements

In Scope The application

- Users should be able to post and view their tweets.
- Users should be able to follow other users.

- Users can view feeds containing tweets from the users they follow.
- Users should be able to like and comment on tweets.

Out of Scope

- Sending and receiving messages from other users. We have covered it in our article on designing WhatsApp.
- Providing personalized notifications to users.

7.2 Detailed Design

7.2.1 Architecture

An application like twitter requires a high-level design but before jumping into the details it can be modeled into block such as:

> 1. Data Modeling: In this block the database will be stored. The database such as username, id, content from the user, request etc. The detailed database used are given in the further system components. 2. Server feeds: This will collect all the feeds from the people user follows and arrange them in chronological order.

Therefore, when the server receives a request for an action (tweet, like etc.) from a client it performs two parallel operations: i) it persists the action in the data store ii) publishes the action in a streaming data store for a pub-sub model. It is followed by the various services (e.g. User Feed Service, Media Counter Service) reading the actions from the streaming data store and performs their specific tasks. The streaming data store makes the system extensible to support other use-cases (e.g. media search index, locations search index, and so forth) in future.

7.2.2 System Components

The system will comprise of several micro-services each performing a separate task. We will use a graph database such as Neo4j to store the information. The reason we have chosen a graph data-model is that our data will contain complex relationships between data entities such as users, tweets, and comments as nodes of the graph. We will then use edges of the graph to store relationships such as follows, likes, comments, and so forth. Additionally, we can use columnar databases like Cassandra to store information like user feeds, activities, and counters.

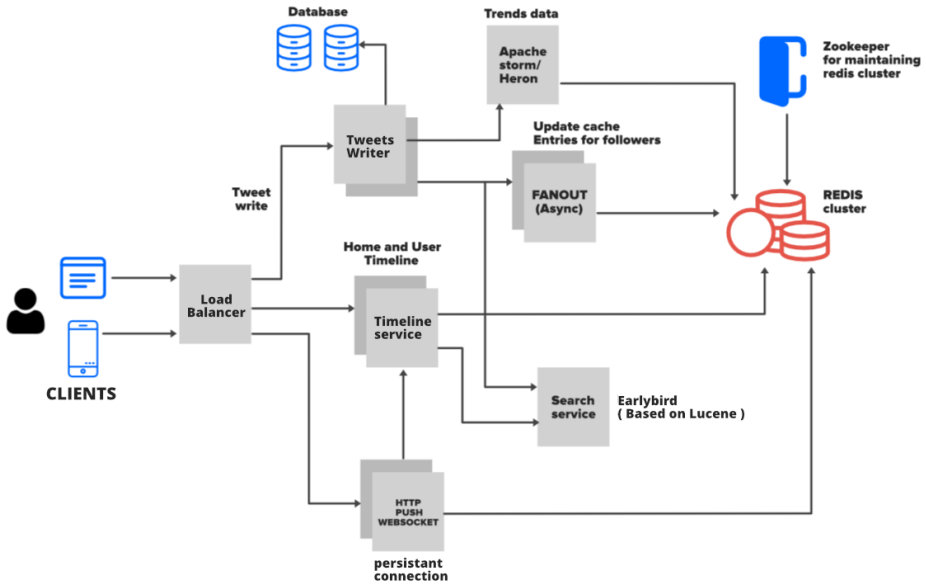


Figure 7.1: High level System components

7.3 Component Design

7.3.1 Tweeting a post

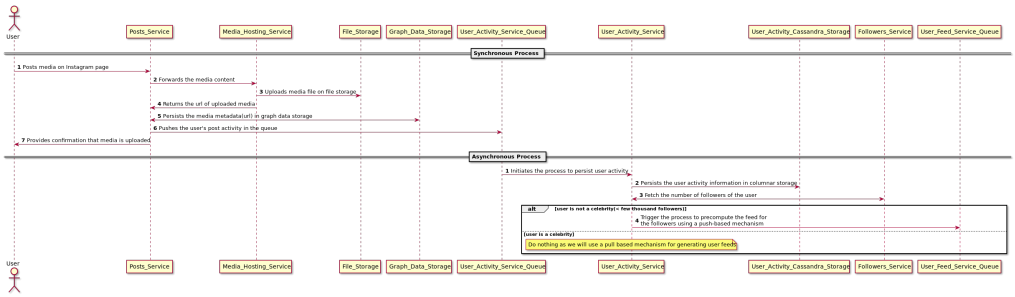


Figure 7.2: Synchronous and Asynchronous process for posting on Twitter

There are two major processes which are executed when a user posts a tweet. Firstly, the synchronous process which is responsible for uploading any media content on file storage, persisting the media metadata in graph data-storage, returning the confirmation message to the user and triggering the process to update the user activity. The second process occurs asynchronously by persisting user activity in a columnar data-storage(Cassandra) and triggering the process to pre-compute the feed of followers of non-celebrity users (having few thousand followers). We don't pre-compute feeds for celebrity users (have 1M+ followers) as the process to fan-out the feeds to all the followers will be extremely compute and I/O intensive.

7.3.2 API Design

We have provided the API design of posting an image on Instagram below. We will send the file and data over in one request using the multipart/form-data content type. The MultiPart/Form-Data contains a series of parts. Each part is expected to contain a content-disposition header [RFC 2183] where the disposition type is "form-data".

URL:

```
POST /users/<user_id>/posts
```

Sample Request Body: Content-Type: *multipart/form-data* ; boundary=ExampleFormBoundary

```
--ExampleFormBoundary
Content-Disposition:
form-data; name="file"; filename="test.png"
Content-Type: image/png

{image file data bytes}
--ExampleFormBoundary--
```

Sample response

```
{
    "type": "ImageFile",
    "id": "667",
    "createdAt": "1466623229",
    "createdBy": "9",
    "name": "test.png",
    "updatedAt": "1466623230",
    "updatedBy": "9",
    "fullImageUrl":
    "https://mybucket.s3.amazonaws.com/myfolder/test.jpg"
",
    "size": {
        "type": "Size",
        "width": "316",
        "height": "316"
    },
}
```

7.3.3 Precompute feeds

This process starts executing when non-celebrity users make a post on Instagram. It is triggered when a message is added in the User Feed Service Queue. Once the message is

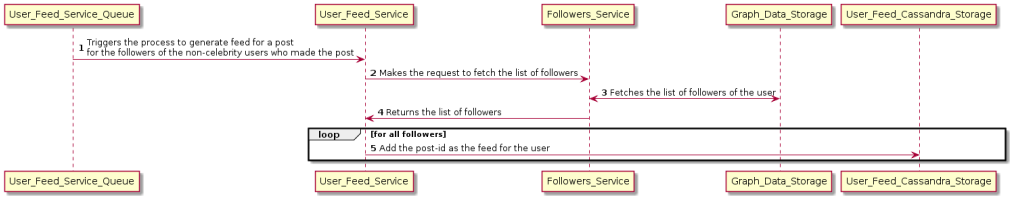


Figure 7.3: Pre-compute feeds from non celebrity users

added in the queue, the User Feed Service makes a call to the Followers Service to fetch the list of followers of the user. After that, the post gets added to the feed of all the followers in the columnar data storage.

7.3.4 Fetching User Feed

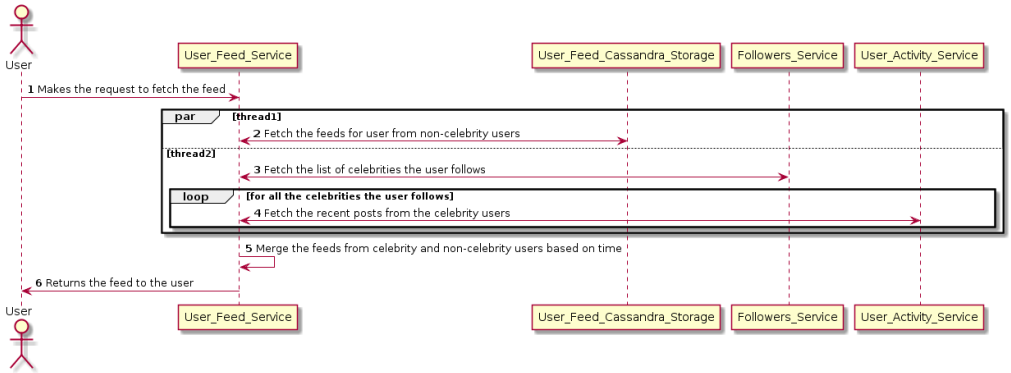


Figure 7.4: Sequence of operations involved in fetching user feed

When a user requests for feed then there will be two parallel threads involved in fetching the user feeds to optimize for latency. The first thread will fetch the feeds from non-celebrity users which the user follows. These feeds are populated by the fan-out mechanism described in the PreCompute Feeds section above. The second thread is responsible for fetching the feeds of celebrity users whom the user follow. After that, the User Feed Service will merge the feeds from celebrity and non-celebrity users and return the merged feeds to the user who requested the feed.

7.3.5 API Design

URL:

```
GET /users/<user_id>/feeds
```

Sample Response

The response below can be mapped directly to the graph data model mentioned in the next section.

```
{
     “feeds”: [
{
“postId”: “post001”,
“postOwnerId” : “user001”,
“postOwnerName” : “Bill Gates”,
“mediaURL” :
“https://mybucket.s3.amazonaws.com/myfolder/test.jpg”,
“numberOfLikes” : 400000
“numberOfComments” : 19789
“comments”: [
{
“commenterUserId” : “user004”,
“commenterName” : “JeffBezos”
“comment” : “Amazing!”
“likes” : [
{
“likerId” : “user003”,
“likerUserName” : “Bob”
}
]
}
]

}
    ]
}
```

7.4 Data Models

7.4.1 Graph Data Models

We can use a graph database such as Neo4j which stores data-entities such as user information, posts, comments, and so forth as nodes in the graph. The edges between the nodes are used to store the relationship between data entities such as followers, posts, comments, likes, and replies. All the nodes are added to an index called nodeIndex for faster lookups. We have chosen this NoSQL based solution over relational databases as it provides the scalability to have hierarchies which go beyond two levels and extensibility due to the

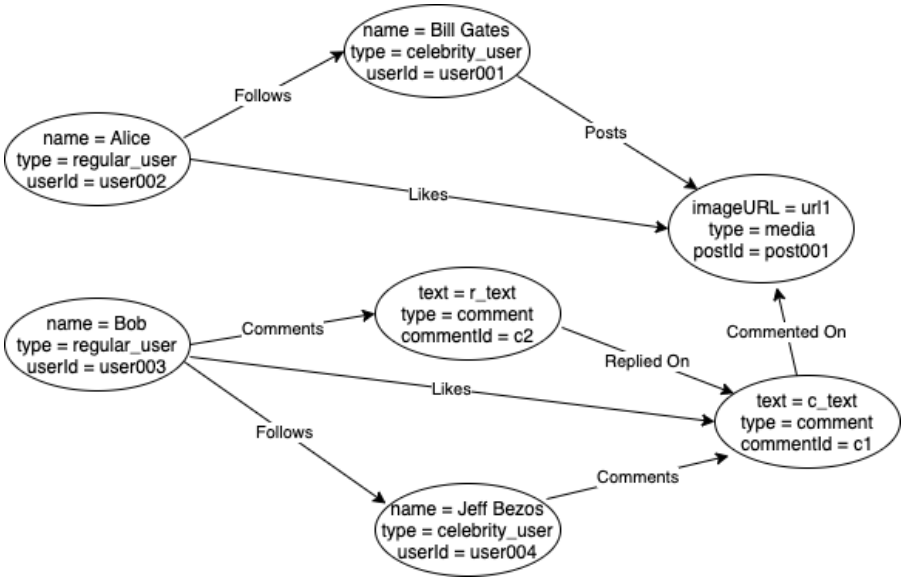


Figure 7.5: Graph representation of users, posts and comments

schema-less behavior of NoSQL data storage.

7.4.2 Sample Queries supported by Graph Databases

Fetch all the followers of Jeff Bezos

```
Node jeffBezos = nodeIndex.get("userId", "user004");
List<Node> jeffBezosFollowers = new ArrayList();

for (Relationship relationship: jeffBezos.getRelationships
(INGOING, FOLLOWS))
{
   jeffBezosFollowers.add(relationship.getStartNode());
}
```

Fetch all the posts of Bill Gates

```
Node billGates = nodeIndex.get("userId", "user001");
List<Node> billGatesPosts = new ArrayList();

for (Relationship relationship: billGates.getRelationships
(OUTGOING, POSTS))
{
    billGatesPosts.add(relationship.getEndNode());
}
```

Fetch all the posts of Bill Gates on which Jeff Bezos has commented

```
List<Node> commentsOnBillGatesPosts = new ArrayList<>();

for(Node billGatesPost : billGatesPosts) {
     for (Relationship relationship: billGates.getRelationships
     (INGOING, COMMENTED_ON)) {
commentsOnBillGatesPosts.add(relationship.getStartNode());
     }
}

List<Node> jeffBezosComments = new ArrayList();

for (Relationship relationship: jeffBezos.getRelationships
(OUTGOING, COMMENTS)) {
    jeffBezosComments.add(relationship.getEndNode());
}

List<Node> jeffBezosCommentsOnBillGatesPosts =
commentsOnBillGatesPosts.intersect(jeffBezosComments);
```

7.4.3 Columnar Data Model

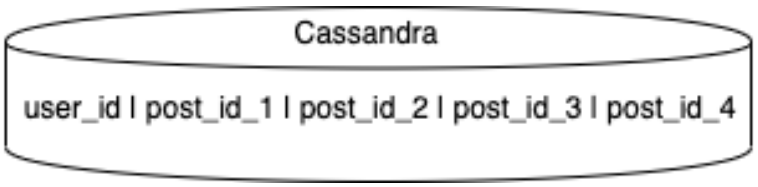


Figure 7.6: Columnar Data Model for user feed and activities

We will use columnar data storage such as Cassandra to store data entities like user feed and activities. Each row will contain feed/activity information of the user. We can also have a TTL based functionality to evict older posts. The data model will look something similar to:

```
User_id -> List<post_id>
```

FUN FACT: In this talk, Dikang Gu, a software engineer at Instagram core infra team has mentioned about how they use Cassandra to serve critical use cases, high scalability requirements, and some pain points.

7.4.4 Streaming Data Model

We can use cloud technologies such as Amazon Kinesis or Azure Stream Analytics for collecting, processing, and analyzing real-time, streaming data to get timely insights and react quickly to new information(e.g. a new like, comment, etc.). We have listed below the de-normalized form of some major streaming data entities and action.

A. class User {
String id;
String userName;
String fullName;
boolean isPrivate;
...
}

B. class Post {
String id;
String ownerId;
MediaContentType contentType;
...
}

C. class LikeEvent {
String likerId;
String postId;
String postOwnerId;
Post;
User liker;
User postOwner;
boolean isFollowingMediaOwner
...
}

D. class Follow {
User follower;
User followedUser;
boolean isFollowedUserCelebrity;
...
}

Figure 7.7: De normalised major data-entities and actions

The data entities **A** and **B** above show the containers which contain denormalized information about the Users and their Posts. Subsequently, the data entities **C** and **D** denote the different actions which users may take. The entity C denotes the event where a user likes a post and entity D denotes the action when a user follows another user. These actions are read by the related micro-services from the stream and processed accordingly. For instance, the LikeEvent can be read by the Media Counter Service and is used to update the media count in the data storage.

7.5 Optimization

We will use a cache having an LRU based eviction policy for caching user feeds of active users. This will not only reduce the overall latency in displaying the user-feeds to users but will also prevent re-computation of user-feeds.

The scope of optimization lies in providing the best content in the user feeds. We can do this by ranking the new feeds (the ones generated after users last login) from those who the user follows. We can apply machine learning techniques to rank the user feeds by assigning scores to the individual feeds which would indicate the probability of click,

Figure 7.8: LRU bases cache for storing user feeds of active users

like, comment and so forth. We can do this by representing each feed by a feature vector which contains information about the user, the feed and the interactions which the user has had with the people in the feed (e.g. whether the user had clicked/liked/commented on the previous feeds by the people in the story). It's apparent that the most important features for feed ranking will be related to social network. Some of the keys of understanding the user network are listed below.

- Who is the user a close follower of? For example, one user is a close follower of Elon Musk while another user can be a close follower of Gordon Ramsay.
- Whose photos the user always like?
- Whose links are most interesting to the user?

We can use deep neural networks which would take the several features (> 100K dense features) which we require for training the model. Those features will be passed through the n-fold layers, and will be used for predicting the probability of the different events (likes, comments, shares, etc.).

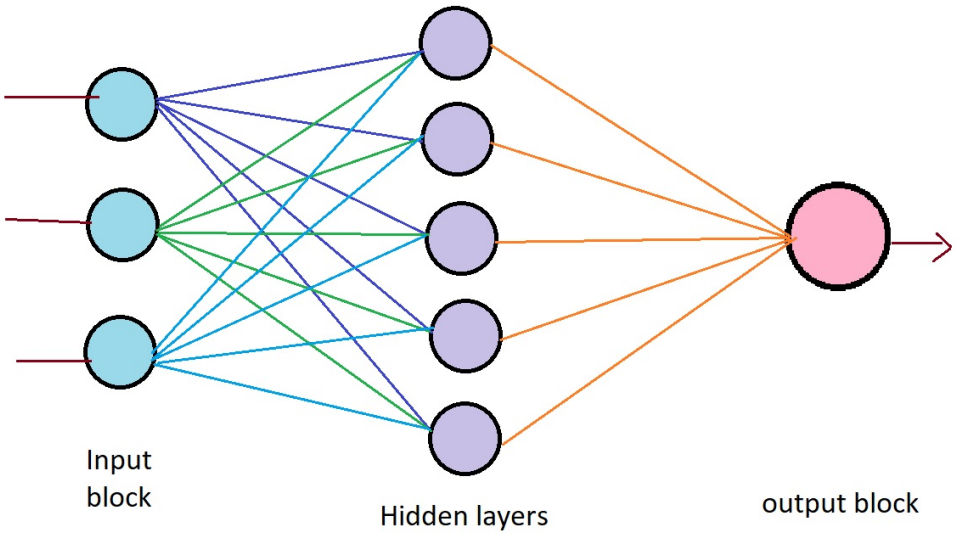


Figure 7.9: Block diagram of deep neural networks

FUN FACT: In this talk, Lars Backstrom, VP of Engineering @ Facebook talks about the machine learning done to create personalized news feeds for users. He talks about

the classical machine learning approach they used in the initial phases for personalizing News Feeds by using decision trees and logistic regression. He then goes to talk about the improvements they have observed in using neural networks.

8 WhatsApp

8.1 Introduction

Note:This is one way of designing an instant messenger system such as WhatsApp. However, we don't guarantee that WhatsApp is designed in this way. This is our prototype on designing a similar system

8.1.1 Problem Statement

In this chapter. we will design an instant messenger platform such as WhatsApp or Signal which users can utilize to send messages to each other. An essential aspect of the application is that the chat messages won't be permanently stored in the application.

FUN FACT: Some of the chat messengers such as FB Messenger stores the chat messages unless the users explicitly delete it. However, instant messengers such as WhatsApp don't save the messages permanently on their server.

8.1.2 Gathering requirements

The instant messenger application should meet the following requirements.

- It should be able to support one-on-one conversations between users.
- It should be able to show Sent/Delivered/Read confirmation to other users.

- It should be able to provide information about the last time a user was active.
- It should allow users to share images.
- It should be able to send push notifications to other users

8.1.3 Capacity Planning

We need to build a highly scalable platform which can support traffic at the scale of WhatsApp. Additionally, while doing capacity planning, we need to ensure that we think through the worst-case scenarios of peak traffic. Some of the numbers which we can use for capacity estimations of an application (like WhatsApp) are listed below.

- Number of users on the application every month: 1 Billion
- Number of active users per second at peak traffic: 650, 000
- Number of messages per second at peak traffic: 40 Million

The entire application will comprise of several microservices each performing a specific task. The number of servers required in the data plane (Webmaster include the link to the distributed systems chapter) (let's call it to chat microservice) handling the traffic of chat messages can be estimated using the following equation.

= *(chat messages per second Latency)/ concurrent connections per server*

Let's assume that the number of concurrent connections per server is 100K, and the latency of sending a message is 20 milliseconds. In such a scenario, the estimated number of servers required in the chat servers' fleet (using the equation mentioned above) will be 8 (i.e., 40 Million*20 ms/100K). In standard practice, it I s recommended to add a few more servers to account for handling failures of these servers. In a subsequent section, we will see the impact which these chat servers will have on the overall infrastructure cost.

FUN FACT: In this talk, Rick Reed(software engineer @ WhatsApp) talks about optimizing their Erlang-based server applications and tuning the FreeBSD kernel to support millions of concurrent connections per server. This helped them to a great extent in keeping their server footprint as small as possible.

8.2 Detailed Design

The required features of this instant messenger application can be modeled using two micro-services: Chat service and Transient service. The Chat Service will be the one serving the traffic of online chat messages sent by active users. The service will check if the user to whom the message is sent is online or not. If the user is online, then the message will be forwarded to that user instantly. Otherwise, the message will be handled by the

Transient Service. This service will be responsible for maintaining all the messages (text or image) sent to offline users. The data will be stored in the Transient Storage temporarily until the offline user comes back online. We will provide more details about the individual components in one of the later sections.

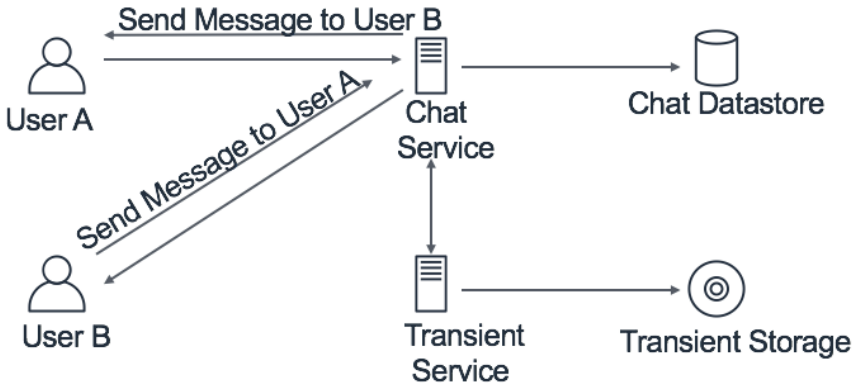


Figure 8.1: High level Design - WhatsApp

FUN FACT:WhatsApp actually uses a much similar approach as discussed by the same WhatsApp engineer (Rick Reed) in a different talk.

8.2.1 API Design

We can expose a REST endpoint to interact with the Chat Service. The definition of the API endpoint to send messages is mentioned below.

sendMessage(String fromUser, String toUser, ClientMetaData clientMetaData, String message)

Request: *fromUser*:The userId who is sending the request *toUser*:The userId to whom the request is being sent *clientMetaData*: The metadata to store client's information such as device details, locations etc. *message*: The message being sent as part of the communication.

8.3 Data Model

We will store details such as the server to which the user is connected and the last time user was active. We can use a document-based database such as MongoDB to store user information such as the last activity time of the user (aka heartbeat time). The data-model will look similar to the table below.

[h]

userID (HK)	**HeartBeatTime**	**connected server**
User A	T1	Server A
User B	T2	Server B

8.4 Component Design

In this section, we will talk about two different scenarios for sending messages in a one-to-one communication. After that, we will discuss the other features which we need to support, such as push notifications and user activity status. In the end, we will look into the different mechanisms for doing optimizations and handling failure scenarios.

8.4.1 One-to-One communication

Here, we will talk about the two different scenarios associated with sending messages to another user. The first scenario involves sending a text message to an online user. In the second scenario, we have described the sequence of operations involved in sending an image to an offline user.

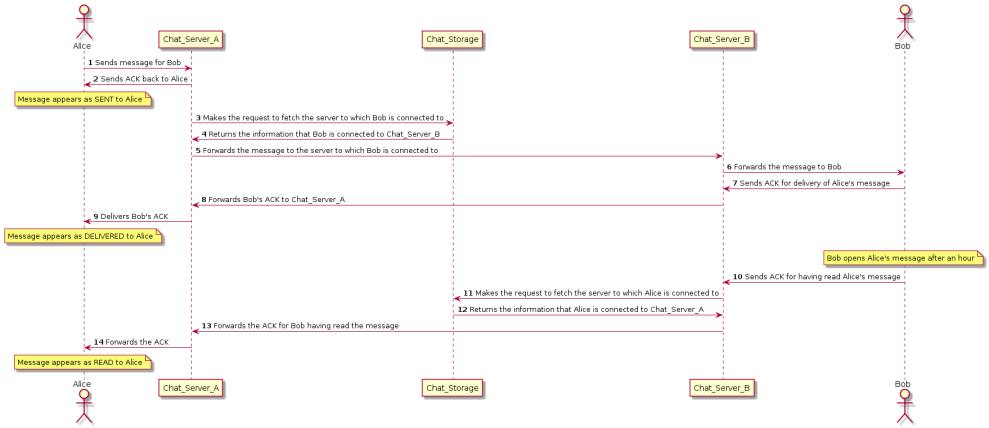


Figure 8.2: One-to-One communication between online users

Scenario-1: Sending text to an online user The details about each of the steps in the sequence diagram for sending a text message to an online user is mentioned below.

1. Alice sends a message to Bob which gets directed to the chat server with which Alice is connected.

2. Alice gets an acknowledgement from the chat server it's connected (i.e. Chat$server_A$)$and the message is m$
$storage to fetch information about the chat server to which Bob is connected$.

3. Chat$storage returns the information that Bob is connected to Chat server_B$.Chat$server_A forwards the message to Ch$

4. The message gets delivered to Bob using a push mechanism (Webmaster include link).

5. Bob sends ACK back to Chat$server_B$.$The ACK is forwarded to Chat server_A to which Alice is connected$.

6. The ACK gets delivered to Alice and is marked as DELIVERED.

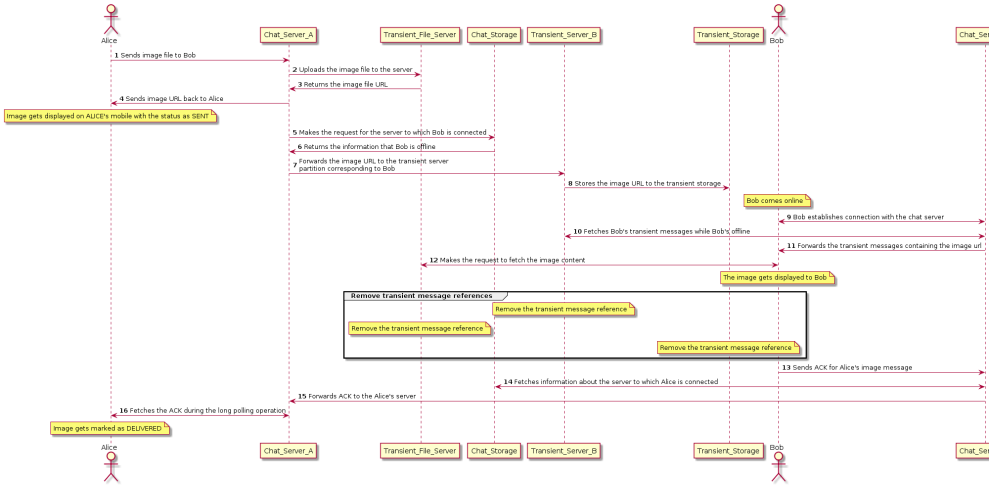


Figure 8.3: Sequence diagram: Sending image to an offline user

Scenario 2: Sending image to an offline user The details about each of the steps in the sequence diagram for sending an image to an offline user is mentioned below.

1. Alice sends an image to Bob which gets forwarded to Chat$_{S}erver_A, the server with which Alice is connec$

2. File Server returns the image url of the uploaded file to Chat$_{S}erver_A. The image url is returned to Alice which is$

3. Chat$_{S}erver_A makes request for the server to which Bob is connected. Chat_{S}torage returns information that Bob$

4. Chat$_{S}erver_A forwards the message containing the image-url to the transient server The transient sever stores$
$url in transient storage.$

5. Bob comes online and performs heart-beat (Webmaster Include Link) with Chat$_{S}erver_B. Chat_{S}erver_B fetch$

6. Chat$_{S}erver_B forwards the transient messages to Bob. Bob fetches the image from the file server. At this point, th$

7. Bob's device sends ACK for Alice's image to Chat$_{S}erver_B Fetches information about the server to which Alic$

8. Forwards the ACK to Chat$_{S}erver_A The ACK gets delivered to Alice marking the message as DELIVERED.$

Transient Data storage We can implement a queue-based mechanism to store and retrieve the transient messages using a FIFO based policy. We can use existing cloud-based technologies for this purpose, such as Amazon SQS or Windows Azure Queue Service. We can use these queues to store transient messages sent to offline users. All the references to these transient messages are removed from the system once the messages are delivered to the offline user.

8.4.2 Push Notifications

There are two approaches to deliver messages to users by using push technology : client pull or server push. If we go down the route of client pull, we can either decide between long vs. short polling. On the other hand, there are two ways to implement the server push approach: WebSocket and Server-Sent Events(SSE). Websockets has been the de-facto communication protocol for chat applications. We have provided more details about it in the section below.

Using the polling technique, the client asks the server for new data regularly. The trade-off decision to choose the polling technique can be taken using the data-points mentioned below.

9. **Short Polling e.g., AJAX-based timer**

   - **Pros**:more straightforward and not very server consuming if the time between requests is long
   - **Cons**: not ideal for scenarios when we need to be notified of server events with minimal delay

2. **Long Polling e.g., Comet Based on XHR**

   - **Pros**: notifications of server events happen with no delay
   - **Cons**:more complex and consume more server resources

The approach to push server messages to clients is mainly of two types. The first one is WebSocket which is a communication protocol. It provides duplex communication channels over a single TCP connection. It's ideal for scenarios such as chat applications due to its two-directional communication. The other one is called Server-sent events (SSE) which allows a server to send "new data" to the client asynchronously once the initial client-server connection has been established. SSEs are more suitable in a publisher-subscriber model such as real-time streaming stock prices; twitter feeds updates and browser notifications.

8.4.3 User Activity Status

The last time when a user was active is a standard functionality which can be found on instant messengers. We have shown the data-model to store the related information in Table 1 above.

In Fig 8.4, we have shown the mechanism using which a connection is maintained between client and server using WebSockets. Once an initial connection is established between client and server, the communication switches to a bi-directional binary protocol. The connection is kept alive between client and server using heartbeats. We store the last

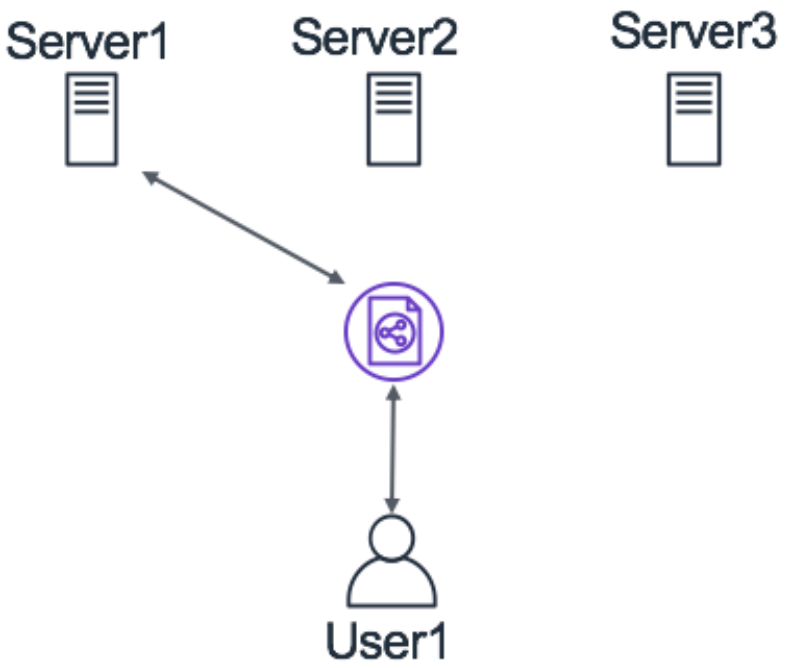


Figure 8.4: Heartbeat between client and server

time a heartbeat was received from a user in a database. The data-storage can then be queried to fetch the last time a user was active.

8.5 Optimization

We can use the parameters listed below to suggest optimizations in the system. We have provided details about the approach which should be followed for suggesting optimizations in a system in this article.

- **Latency**: We can use distributed cache (Include Link to distributed caching) such as Redis to cache the information of user activity status and their recent chats in-memory. This may help in reducing the overall latency of the application and provides a better customer experience. Even some of the database solutions do provide in-memory caching solutions such as Amazon DynamoDB Accelerator.

- **Infrastructure cost**: It's apparent from the system that significant contribution to the infrastructure cost will be from chat servers. The cost incurred from the chat servers can mount quickly if its server foot-print isn't kept in check. One way of doing that is to increase the number of connections per host. It will significantly reduce the number of servers required for maintaining the service. We can accomplish this task by tuning the server configurations and choosing suitable technologies. For instance, engineers at WhatsApp were able to achieve millions of connections per host by optimizing their Erlang-based server applications and tuning the FreeBSD kernel

- **Availability** : We can maintain multiple copies of the transient messages so that even if the messages in one of the copies is lost then it can still be retrieved from the other copy. This would imply maintaining replicas of those transient messages. The client would be responsible for fetching the messages from two queues and will merge them. We will discuss more in the next section.

8.5.1 Addressing Bottlenecks

: The major bottlenecks in the system which are more vulnerable to failures are the chat servers and the transient storage solutions. We have recommended some approaches to handle such failures in the section below.

- **Chat server failure**: The chat servers in the system will be holding connections with the users. There are two ways of handling the failure of chat servers. One approach can be to transfer those TCP connections to another server; however, implementation of such fail-overs isn't trivial. The second, which is comparatively more straightforward is to have the user clients initiate the connection automatically in case of connection loss. The information of the server to which the user is connected needs to be updated in the database is agnostic of the approach we take. For instance, in Fig 5.6, we have shown an illustration of the handling this failure scenario. We can see that User1 is connected to Server1, and when this server goes down, then the connection is re-established with another server (i.e. Server2), and this information is updated in the database.

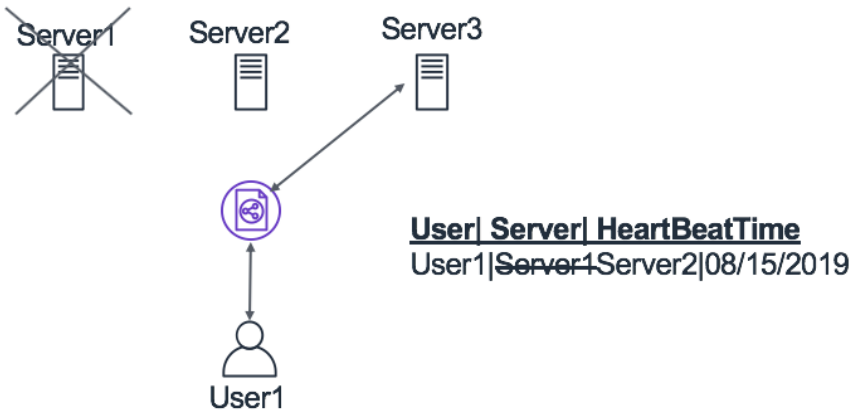


Figure 8.5: Handling chat server failure

- **Transient storage failure**: The transient storage is another component which is prone to failures and can cause loss of messages in-transit to offline users. We can make replicas of the transient storage of each user to prevent the loss of messages which were sent to them when they were offline. When a user comes back online

then both the original and the replica instances of the user's transient storage are queried and merged. In Fig 8.6, we have illustrated a mechanism to handle transient storage failure which gets initiated when user comes back online.

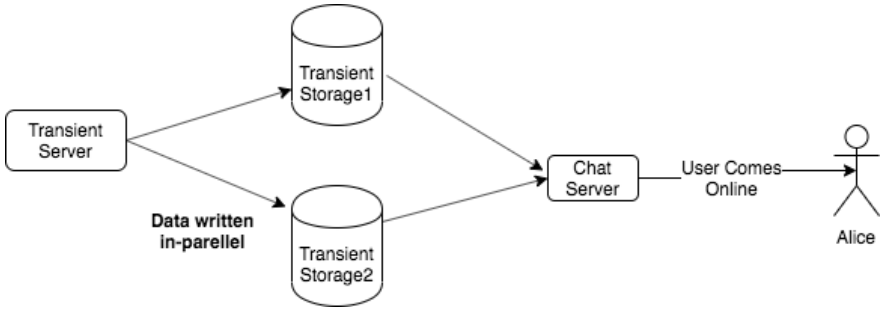


Figure 8.6: Handling transient storage failures

8.5.2 Monitoring

We want to ensure that our service can meet the user demands with high availability and low latency. We can define service level agreements (SLAs) for these metrics and create moderate and severe monitors which can trigger an alarm when these SLAs are violated. For this application, we can define the following SLAs for the sendMessage API.

1. **Availability SLA**: p99.999
2. **Latency SLA**: p99.999 of 5 milliseconds

The availability SLA implies that that the monitor will trigger an alarm if more than 1 out of 1000 requests fail. Likewise, the latency SLA means that an alert will be triggered if the server takes >5ms to respond for more than 1 out of 100 requests it receives.

Additionally, we can put failure alarms in different error scenarios. One such scenario can occur when the chat server isn't able to fetch transient messages for a user from all the replicas of transient storage. This maps to Step10 illustrated in Fig 3 above where $Chat_{S}erver_{B}requeststhetransientservertofetchthemessagessenttoBobwhenBobwasoffline.Let'sassumeth$

8.5.3 Extended Requirements

We can extend the system to support group chats using which we can get messages delivered to multiple users. We can create a data-model to store the group data-entity, which will be identified by GroupChatID and will be used to maintain a list of people who are part of that group. The system described above is extensible to support the scenarios for sending message to online and offline users. We can build a component which will be responsible for ensuring that the messages get delivered to all the users in the group depending upon their activity status.

Another aspect that can be covered as an extension to this problem is that of security, specifically end-to-end encryption, where only the communicating users can read the messages. Each user has a public key that is shared with all the other users with whom the user is communicating. For instance, two users Alice and Bob, are communicating with each other. Alice has Bob's public key and vice versa; however, their private key isn't shared. When Alice sends a message to Bob, the message is encrypted using Bob's public key and sent over the network. The server directs the encrypted message to Bob, who uses the private key to decrypt the message. In this way, the server only has access to the encrypted message, and only Alice and Bob can read the actual messages they exchanged.

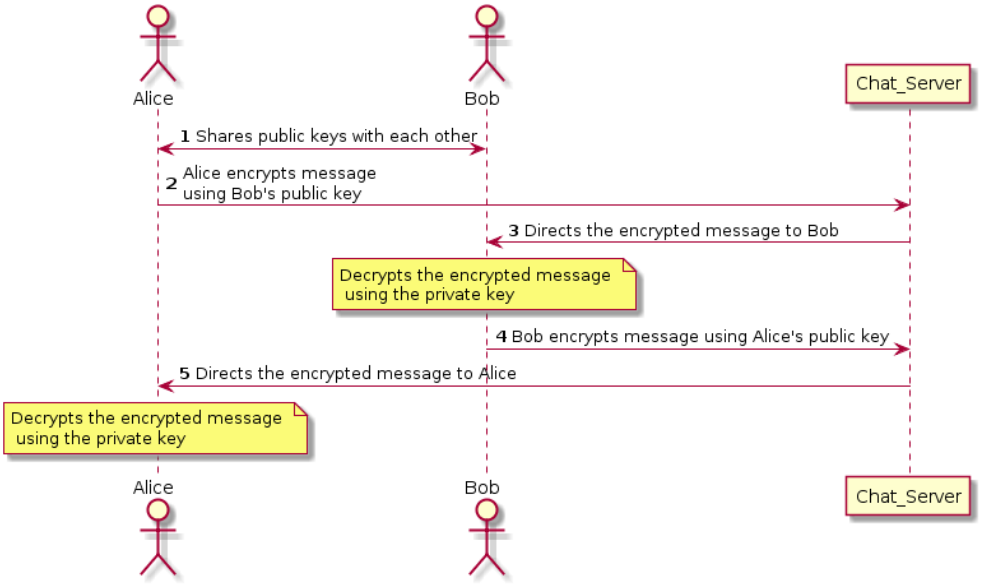


Figure 8.7: End-to-End encryption while sending messages

9 Tinder

9.1 Introduction

Note: This is a prototype on designing a location-based social search application that allows users to use a swiping motion to like or dislike other users similar to Tinder. This has been created by design based on ourresearch going through Tinder engineering blog. These articles are quite informative and detailed. The detailed reference list of the tech talks and blogs has been provided in reference section.

9.1.1 Problem Statement

Design a location-based social search application similar to Tinder which if often used as a dating service. It allows users to use a swiping motion to like (swipe right) or dislike (swipe left) other users, and allows users to chat if both parties like each other (a "match").

9.1.2 Gathering Requirements

In Scope The application should be able to support the following requirements.

- User should be able to create their Tinder profile by adding their bio and uploading photos.
- User should be able to view recommendations of other users in geographically nearby regions.

- Users should be able to like (swipe right) or dislike (swipe left) other recommended users.

- Users should get notifications when matched with other users.

- Users should be able to move to a different location and still get recommendations of nearby users.

Out of Scope Sending and receiving messages from other users. We have covered it in our article on designing WhatsApp (link here)

9.2 Detailed Design

9.2.1 Architecture

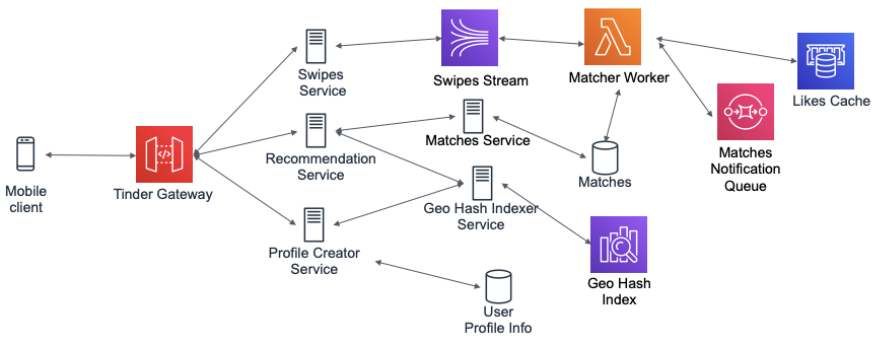


Figure 9.1: High level Design - Tinder

There will be a fleet of micro-services behind the Gateway which will be serving the user requests. The Profile Creator Service will be invoked when the user profile gets created. This service will store the user information in a database and add the user to the corresponding geo-sharded index so that the user shows up in recommendations of nearby users. This index gets queried by the Recommendation Service when it receives the request to generate recommendations for other users. Once the user starts swiping through those recommendations, the Swipes Service receives those swipes and places them in a data-streams (e.g. AWS Kinesis/ SQS). There are a fleet of workers which read data from those streams for generating matches. The workers do this by querying the LikesCache to determine if it's a match, in which case the match notification is sent to both the users using technologies such as WebSockets.

9.3 Component Design

9.3.1 User Profile Creation

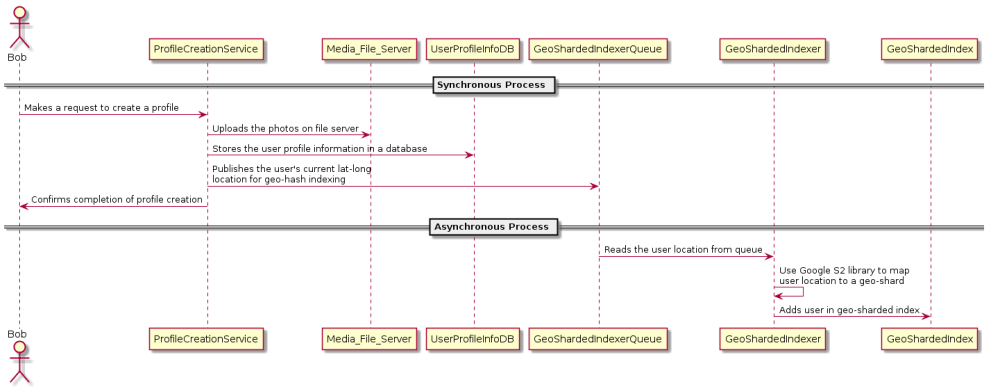


Figure 9.2: Sequence Diagram for User Profile Creation

The sequence diagram above shows the sequence of operations which gets executed when a user creates profile on Tinder. Within the synchronous process, the user media (e.g. photos) is uploaded on a file server and the user information including user's location is persisted in a key-value store like Amazon DynamoDB. Additionally, this user is added to a queue for adding the user to a geo-sharded index.

The asynchronous process reads the user information from the queue and passes this information to the GeoShardingIndexer. The indexer uses geo libraries like Google's S2 library to map user's location to a geo-shard and add the user to the index associated with that shard. This helps the user to show up in recommendations of other nearby users. For instance, in the image below we have shown how a user from North America gets mapped to the corresponding index so that the user gets shown in recommendations of nearby users.

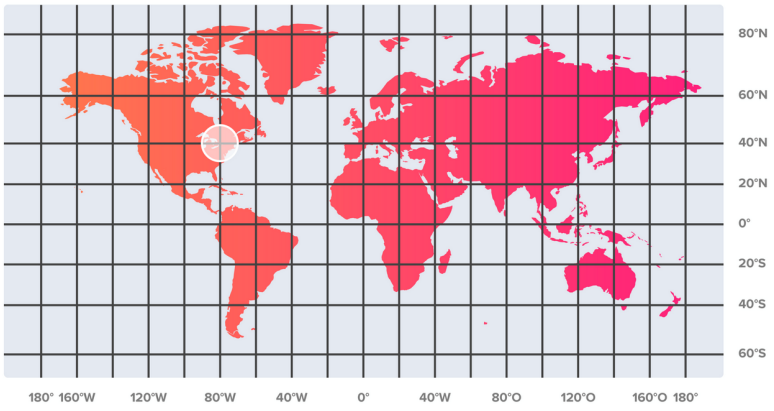


Figure 9.3: User from North America will be mapped to the corresponding shard (image: tinder engineering blog)

9.3.2 User Profile Information - Sample data Model

We have shown below a json blob for storing the user profile information. We can use a key-value store such as Amazon DynamoDB or Riak for maintaining this data.

```
{
“userId”(PK) : “AWDGT567RTH”,
“name” : “Julie”,
“age” : 25,
“gender” : “F”,
“location”: {
“latitude” :
“longitude” :
},
“media”: {
“images”: [
“https://mybucket.s3.amazonaws.com/myfolder/img1.jpg”,
“https://mybucket.s3.amazonaws.com/myfolder/img2.jpg”,
“https://mybucket.s3.amazonaws.com/myfolder/img3.jpg”


]
},
“recommendationPreferences”: {
“ageRange”: {
“min”: 21,
“max”: 31
},
“radius”: 50
}
}
```

9.3.3 Fetch User recommendations

In the previous section, shows how users get added into the geo-sharded index. Let’s see how the user gets shown in the recommendation of other users. When a user request arrives at the Recommendation Engine it is forwarded to the request to the GeoShardedIndexer. The indexer determines the geo-shards to be queried based on user location and radius using geo-libraries like Google’s S2. Followed by, the indexer queries all the geo-sharded

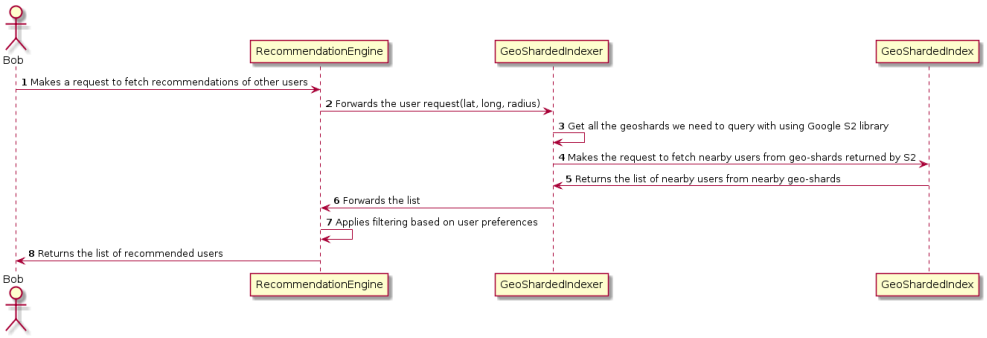


Figure 9.4: Fetch User recommendations

indexes (more details in next section) are mapped to the shards returned by Google S2 to fetch the list of all the users in those indexes and returns that list to the RecommendationEngine. The engine applies filtering on the list based on user preferences and returns the final list of recommendations to the user.

9.3.4 Geo-Sharded Index

A naïve approach of maintaining this index would be to have an Elasticsearch cluster with one index and the default number of shards. However, this approach won't hold up to the scaling expectations which an application like Tinder requires. We should leverage the fact that Tinder's recommendations are location-based. For instance, when we are serving a user from India, we don't need to include the users from USA. This fact can be used by keeping an optimal index size for better performance. We can optimize the index size by sharding the active user records based on their geo-locations so that the active user count will remain balanced across shards. We can represent the balance of a geo-sharding configuration with N shards by the standard deviation of active user counts across shards as mentioned below.

Balance(Shard1, Shard2,. . . , ShardN) = standard-deviation(Active User Count of Shard1, Shard2,. . . , ShardN)

The geo-sharding configuration with the minimal standard deviation would be best balanced. Geo-libraries like Google's S2 Library can be used which are based on hierarchical decomposition of sphere into "cells" using Quad-Trees. A visualization of generated geo-sharded map for our user-case is shown below. The inference from the given graph is, that geo-shards are physically closer and larger for areas having lower number of active users. For instance, in the image below shards are larger on water bodies like seas and oceans as they only have users from some islands, however, shards are smaller on land. In the image below, it can seen that North America has three shards, however, entire England and Greenland along-with a large portion of Atlantic Ocean share a single shard due to lesser density of active users.

The S2 library provides two major functions: i) given a location point (lat, long), return

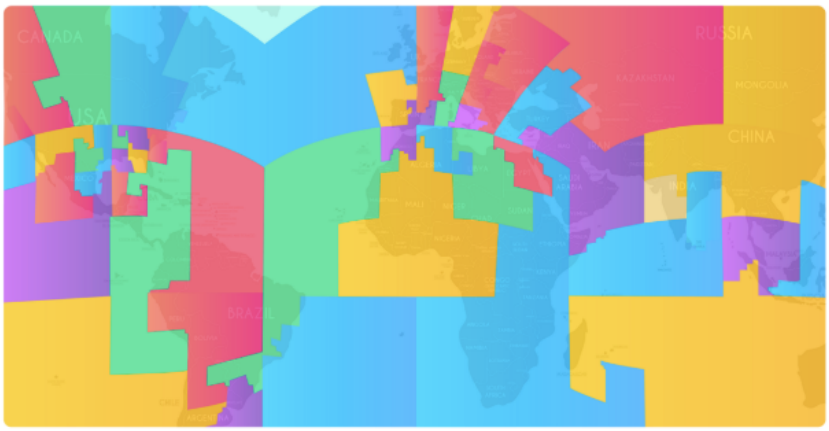


Figure 9.5: Geo-sharded map generated by Google S2, each shard us a S2 cell (image: Tinder engineering blog)

the S2 cell that contains it ii) given a circle (lat, long, radius), return the S2 cells that cover the circle. Each S2 cell can be represented by a geo-shard which will be mapped to an index in our system. When a profile gets created, the user is added to the search index for that corresponding S2 cell. To fetch recommendations for a user, we query the indexes of the nearby S2 cells depending on the circle radius as shown in the image below.

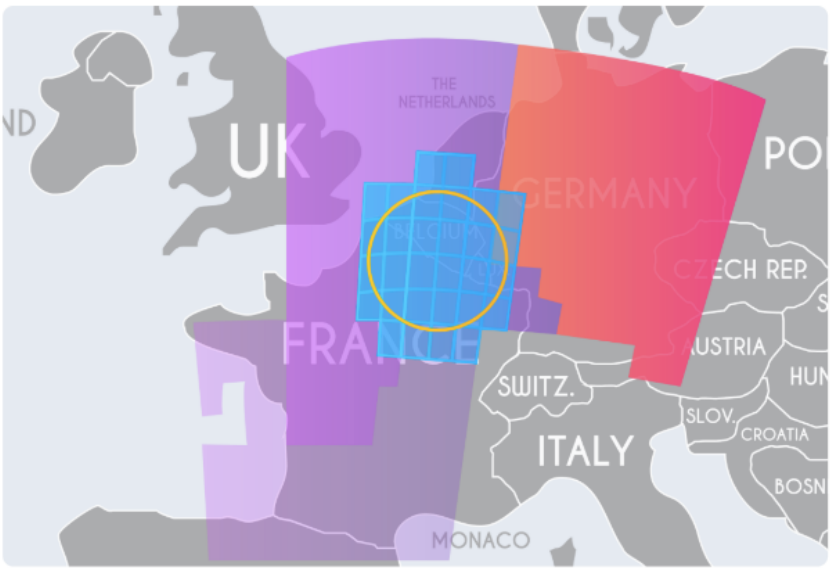


Figure 9.6: Fetching recommendations for a user from nearby shards(image: Tinder engineering blog

9.3.5 Swipes and Matches

The above image shows the sequence of operations which gets executed when a user swipes left/right. The swipes ingester processes the swipes and puts the left swipes into a stream

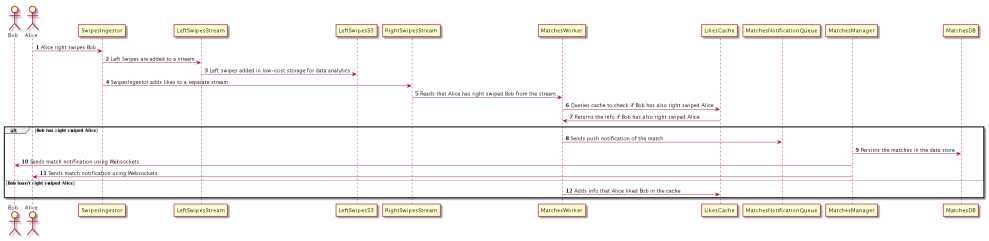


Figure 9.7: : Sequence of operations for user swipes and matching

which persist those swipes to a low-cost data storage (e.g. Amazon S3). These left swipes can be used for data analysis for some user cases.

On the other hand, the right swipes are put in a separate stream and is ultimately read by matcher worker thread. The matcher worker threads reads the likes message from the stream and checks if the corresponding entry exist in the LikesCache. For instance, in the image above Alice likes Bob and the match worker checks if an entry exists for Bob liking Alice in the cache. If both Alice and Bob likes each other then it's called a match and a match notification is sent both the users using server push mechanism like Websockets. If Bob hasn't liked Alice yet, an entry is made in the LikesCache for Alice liking Bob.

9.3.6 Matches Data Model

We can use a key-value store (e.g. Amazon DynamoDB) to persist the information about matches (users liking each other). The hash key used for this data store can be a composite key of the unique identifiers of the users who liked each other. The value in the data-store will contain metadata information related to the match.

Key	Value

userId1_userId2
 	
(e.g. AWDGT567RTH_
ARTHT567WDG)
	{
	"matchTimestamp": T2
	"likes":[
	{
	"likerId" : "AWDGT567RTH",
	"userLikedId" : "ARTHT567WDG",
	"timestamp": T1
	},
	{
	"likerId" : "ARTHT567WDG",
	"userLikedId" : "AWDGT567RTH",
	"timestamp": T2
	}
	]
	
	}

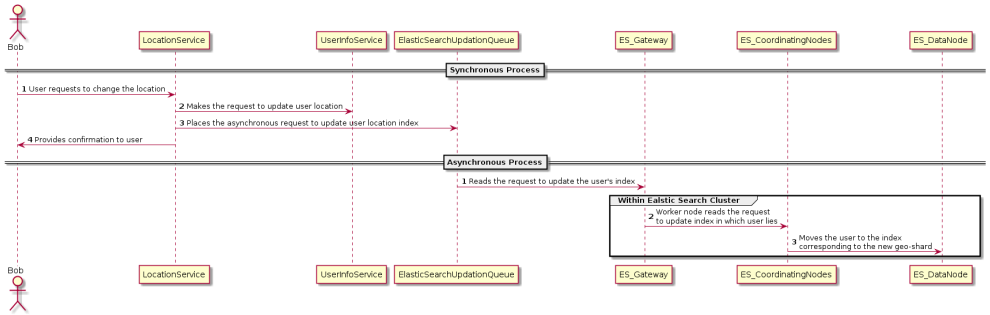


Figure 9.8: User switching locations

9.3.7 User Switching Locations

When a user changes locations it has to be ensured to provide recommendations to the user from the new location and vice-versa. The user location gets updated to the new location so that the updated location is used for fetching recommendations for the user. Additionally, the index mapped is also updated to user's new location with user's information so the user shows up in recommendations at the location. This process gets executed asynchronously.

The elastic search cluster (explained below) containing the geo-sharded indexes which reads the message from a queue to update user's index. The elastic search is co-ordinating nodes moving the user's information from the index mapped to user's old location to the index mapped to user's new location. This ensures that the user shows up in recommendations of other users in the new location.

9.3.8 Elastic Search Cluster

The cluster will comprise of multiple master nodes, each having two auto-scaling groups(ASG), one containing only coordinating nodes (this is where all the requests are sent) and another containing all the data nodes. Each data node will contain certain number of indexes (combination of primaries and replicas) of randomly distributed shards. For each user query, the responsibility of the co-ordinating node is to query the data-nodes of the target shards for handling the user query. The increase in reliability and robustness of the elastic search cluster by sharding the user data using their geographical locations and creating replicas of those shards.

9.4 Optimization

One of the most important aspects of an application like Tinder is the recommendations (of potential matches) it is provide to a user. In one the sections above, it shown how to generate recommendations for a user by querying indexes corresponding to the nearby geo-shards of a user. The system can be optimized by applying machine learning to rank

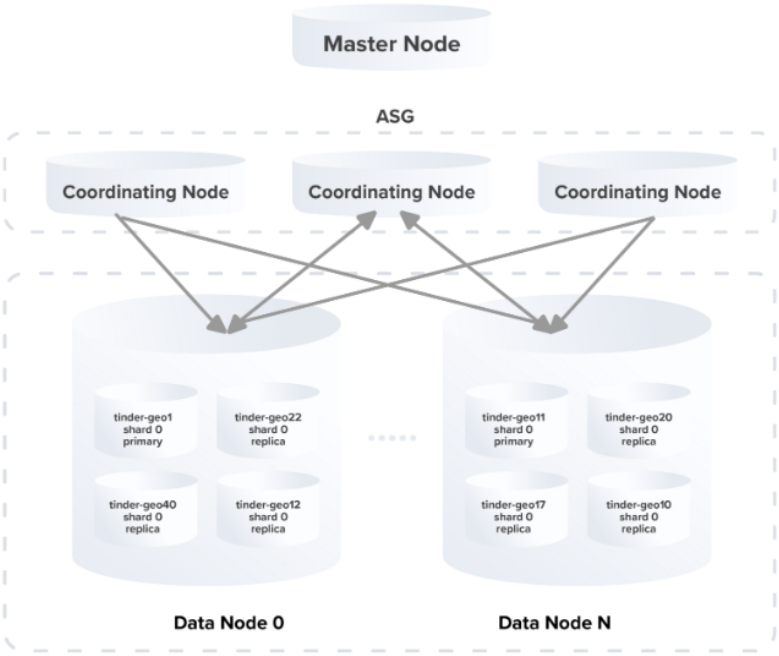


Figure 9.9: Geo-sharded cluster architecture

the recommendations. The machine learning model will optimize on the user's potential to right swipe recommended potential matches. The features which may impact user's decision to swipe left or right are listed below:

- **User demographics data**: Age, Gender, Race, Location, Profession and so forth
- **User's Tinder history data**: Swipe Left, Swipe Rights, Historical geo-locations, Daily usage time
- **Extracted Information from User's Bio**: Likes, Dislikes, Preferences
- **Extracted Information from user's pictures**:: Facial features, Hair color, Body type

A regression problem can be framed by using these features to find the probability that the user will swipe right a recommendation. We can then leverage algorithms such as Logistic Regression to compute those probabilities, which will be used for ranking the recommendations. In addition to this, we can also optimize the mechanism to send match notifications to user by prefetching the information that a user has been swiped right by a recommended user. Pre-fetching this information, we can notify the user about a match (if and when it occurs) right-away, hence preventing the network call. For instance, if Alice is shown in Bob's recommendation and Alice has already swiped Bob right then Bob gets an instantaneous match notification (without any network hop) in case Bob swipes Alice right too.

9.5 Fitness check up

1. Why are we geo-sharded index for storing generating recommendations for user?
   (a) Usage of geo-sharded index optimizes the query performance to fetch user recommendations
   (b) It doesn't provide a major advantage and we will be better off using a data store.
2. What is the main advantage of using a composite key of the userIds who liked each other in the MatchesDB?
   (a) It helps prevent hot partition problems.
   (b) It optimizes the query performance to fetch the matches of a given user.

10 Google Docs

10.1 Introduction

10.1.1 Gathering Requirements

In scope The application should be able to support the following requirements. The user should be able to do following tasks

- create a document
- share a document
- edit a document

Out of scope

- User authentication for accessing the document
- User should be able to chat on the document
- User should be able to provide edit and view permissions to different users
- User should be able to comment on the document

10.2 Detailed Design

10.2.1 System Architecture

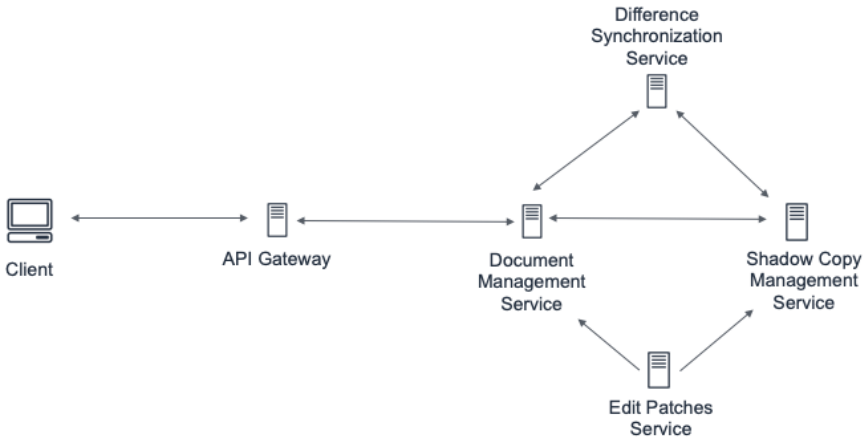
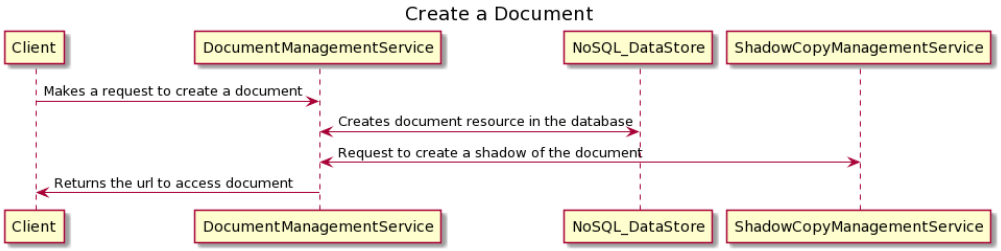


Figure 10.1: System architecture for Google docs

10.3 System Components

10.3.1 Creating a document

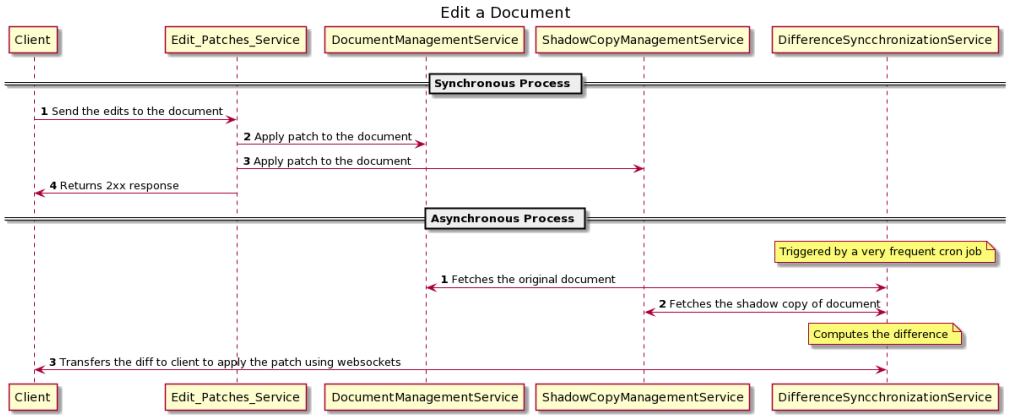


10.3.2 API Design: Document Management service

URL: POST /collaborativeEditing/documents

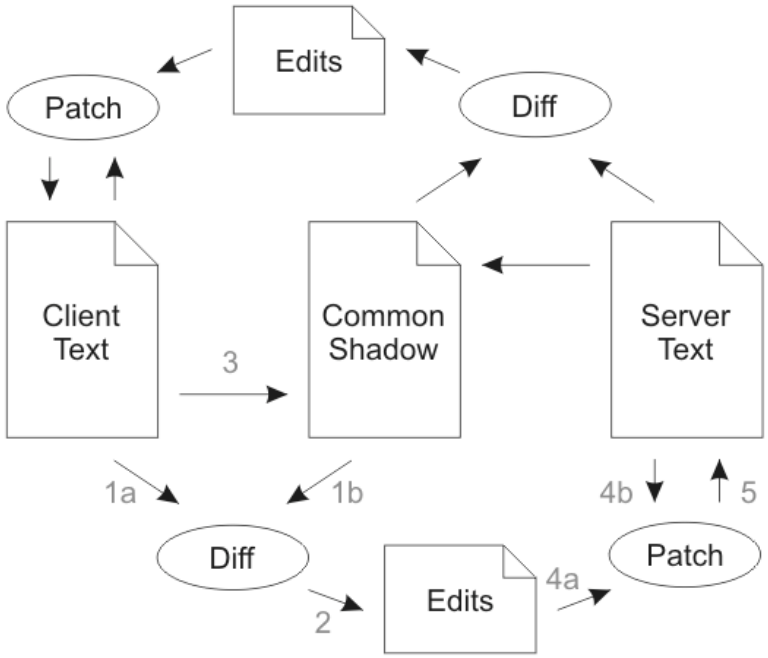
Request Body: *Format*: Either html or markdown *Content*: The html or markdown content of the new document *title*: The title of the new document *type*: Either document, spreadsheet or slides

Sample Response: *documentId*: The url and the identifier to access the document



10.3.3 Editing a document

10.3.4 Differential Synchronization



The above diagram shows the simplest form of Differential Synchronization. The following walk-through starts with Client Text, Common Shadow and Server Text all being equal. Client Text and Server Text may be edited at any time. The goal is to keep these two texts as close as possible with each other at all times.

1. Client Text is diffed against the Common Shadow.
2. This returns a list of edits which have been performed on Client Text.
3. Client Text is copied over to Common Shadow. This copy must be identical to the value of Client Text in step 1, so in a multi-threaded environment a snapshot of the text should have been taken.

4. The edits are applied to Server Text on a best-effort basis.

5. Server Text is updated with the result of the patch. Steps 4 and 5 must be atomic, but they do not have to be blocking; they may be repeated until Server Text stays still long enough.

The process now repeats symmetrically in the other direction. This time the Common Shadow is the same as Client Text was in the previous half of the synchronization, so the resulting diff will return modifications made to Server Text, not the result of the patch in step 5. The enabling feature is that the patch algorithm is fuzzy, meaning patches may be applied even if the document has changed. Thus, if the client has typed a few keystrokes in the time that the synchronization took to complete, the patches from the server are likely to have enough recognizable context that they may still be applied successfully. However, if some or all of the patches fail in step 4, they will automatically show up negatively in the following diff and will be patched out of the Client Text. Here's an example of actual data flow.

1. Client Text, Common Shadow and Server Text start out with the same string: "Macs had the original point and click UI."

2. Client Text is edited (by the user) to say: "Macintoshes had the original point and click interface." (edits underlined)

3. The Diff in step 1 returns the following two edits:

```
@@ -1,11 +1,18 @@
 Mac
 +intoshe
 s had th
 @@ -35,7 +42,14 @@
 ick
 -UI
 +interface
```

4. Common Shadow is updated to also say: "Macintoshes had the original point and click interface."

5. Meanwhile Server Text has been edited (by another user) to say: "Smith Wesson had the original point and click UI." (edits underlined)

6. In step 4 both edits are patched onto Server Text. The first edit fails since the context has changed too much to insert "intoshe" anywhere meaningful. The second edit succeeds perfectly since the context matches.

7. results in a Server Text which says: "Smith Wesson had the original point and click interface."

8. Now the reverse process starts. First the Diff compares Server Text with Common Shadow and returns the following edit:

```
@@ -1,15 +1,18 @@
-Macintoshes
+Smith & Wesson
had
```

9. Finally, this patch is applied to Client Text, thus backing out the failed "Macs" -> "Macintoshes" edit and replacing it with "Smith Wesson". The "UI" -> "interface" edit is left untouched. Any changes which have been made to Client Text in the meantime will be patched around and incorporated into the next synchronization cycle.

10.3.5 API Design: Edit Patch Service

URL: POST

`/collaborativeEditing/documents/<document_id>/section/<section_id>`

Request Body: *format*: Either HTML or markdown *diff_content*: the difference between client text and shadow *type*: Either document, spreadsheet or slides

Sample response: *responsecode*: 2xx, 5xx, 4xx

10.3.6 API Design: document management service

URL:

POST

`/collaborativeEditing/documents/<document_id>/section/<section_id>`

Request Body: *format*: Either HTML or markdown *diff_content*: the difference between client text and shadow *type*: Either document, spreadsheet or slides *location*: Where we insert the new content. Defaults to : `APPEND`.

```
0: APPEND - Append to the end of the document.
1: PREPEND - Prepend to the beginning of the document.
2: AFTER_SECTION - Insert after the section specified by section_id.
```

3: BEFORE_SECTION - Insert before the section specified by section_id.
4: REPLACE_SECTION - Delete the section specified
by section_id and insert the new content at that location.
5: DELETE_SECTION - Delete the section specified by section_id
(no content required).

Sample Response: responsecode: 2xx, 5xx, 4xx

10.4 Data Mode

We will use a key-value data store to store the document information.

{
documentId(HK): String
authorId: String,
title: String,
type: Enum(Document, Slide, Spreadsheet),
content: Map<String, SectionContent>

For example, an entry in this map will be <UUID_1,
{content: String, type: Enum (Document, Slide, Spreadsheet)}
}

Part III

Appendix

A

Appendix A

A.1 Instagram

```
actor User
autonumber
== Synchronous Process ==
User->Posts_Service: Posts media on Instagram page
Posts_Service->Media_Hosting_Service: Forwards the media content
Media_Hosting_Service->File_Storage: Uploads media file on file storage
Media_Hosting_Service->Posts_Service: Returns the url of uploaded media
Posts_Service<->Graph_Data_Storage:
Persists the media metadata(url) in graph data storage
Posts_Service->User_Activity_Service_Queue:
Pushes the user's post activity in the queue
Posts_Service->User: Provides confirmation that media is uploaded
== Asynchronous Process ==
autonumber
User_Activity_Service_Queue->User_Activity_Service:
Initiates the process to persist user activity
User_Activity_Service->User_Activity_Cassandra_Storage:
Persists the user activity information in columnar storage
User_Activity_Service<->Followers_Service:
```

```
Fetch the number of followers of the user
alt user is not a celebrity(< few thousand followers)
User_Activity_Service->User_Feed_Service_Queue:
Trigger the process to precompute the feed for
\nthe followers using a push-based mechanism
else user is a celebrity
note over User_Activity_Service: Do nothing as we will use a pull
based mechanism for generating user feeds
end
```

A.2 Twitter

```
actor User
autonumber
== Synchronous Process ==
User->Posts_Service: Posts media on Instagram page
Posts_Service->Media_Hosting_Service: Forwards the media content
Media_Hosting_Service->File_Storage: Uploads media file on file storage
Media_Hosting_Service->Posts_Service: Returns the url of uploaded media
Posts_Service<->Graph_Data_Storage:
Persists the media metadata(url) in graph data storage
Posts_Service->User_Activity_Service_Queue:
 Pushes the user's post activity in the queue
Posts_Service->User: Provides confirmation that media is uploaded
== Asynchronous Process ==
autonumber
User_Activity_Service_Queue->User_Activity_Service:
 Initiates the process to persist user activity
User_Activity_Service->User_Activity_Cassandra_Storage:
Persists the user activity information in columnar storage
User_Activity_Service<->Followers_Service:
Fetch the number of followers of the user
alt user is not a celebrity(< few thousand followers)
User_Activity_Service->User_Feed_Service_Queue:
 Trigger the process to precompute the feed for
 \nthe followers using a push-based mechanism
else user is a celebrity
```

```
note over User_Activity_Service:
Do nothing as we will use a pull based
 mechanism for generating user feeds
end
```

A.3 WhatsApp

```
actor Alice

autonumber
Alice->Chat_Server_A:Sends message for Bob
Chat_Server_A->Alice:Sends ACK back to Alice
note over Alice:Message appears as SENT to Alice

Chat_Server_A->Chat_Storage:
Makes the request to fetch the server to which Bob is connected to
Chat_Storage->Chat_Server_A:
Returns the information that Bob is connected to Chat_Server_B
Chat_Server_A->Chat_Server_B:
Forwards the message to the server to which Bob is connected to
actor Bob
Chat_Server_B->Bob:Forwards the message to Bob
Bob->Chat_Server_B:Sends ACK for delivery of Alice's message
Chat_Server_B->Chat_Server_A:Forwards Bob's ACK to Chat_Server_A
Chat_Server_A->Alice:Delivers Bob's ACK
note over Alice:Message appears as DELIVERED to Alice
note over Bob: Bob opens Alice's message after an hour
Bob->Chat_Server_B:Sends ACK for having read Alice's message
Chat_Server_B->Chat_Storage:
Makes the request to fetch the server to which Alice is connected to
Chat_Storage->Chat_Server_B:
Returns the information that Alice is connected to Chat_Server_A
Chat_Server_B->Chat_Server_A:
Forwards the ACK for Bob having read the message
Chat_Server_A->Alice:Forwards the ACK
note over Alice:Message appears as READ to Alice
```

A.4 Tinder

```
actor Bob
== Synchronous Process ==
Bob->ProfileCreationService:Makes a request to create a profile
ProfileCreationService->Media_File_Server:
Uploads the photos on file server
ProfileCreationService->UserProfileInfoDB:
Stores the user profile information in a database
ProfileCreationService->GeoShardedIndexerQueue:
Publishes the user's current lat-long\nlocation for geo-hash indexing
ProfileCreationService->Bob: Confirms completion of profile creation
== Asynchronous Process ==
GeoShardedIndexerQueue->GeoShardedIndexer:
Reads the user location from queue
GeoShardedIndexer->GeoShardedIndexer:
Use Google S2 library to map\nuser location to a geo-shard
GeoShardedIndexer->GeoShardedIndex: Adds user in geo-sharded index
```

A.5 Google docs

```
title: Create a Document
Client->DocumentManagementService :
Makes a request to create a document
DocumentManagementService<->NoSQL_DataStore :
Creates document resource in the database
DocumentManagementService<->ShadowCopyManagementService:
Request to create a shadow of the document
DocumentManagementService->Client :
 Returns the url to access document
```

A.6 Netflix

```
actor Content_Uploader
autonumber
Content_Uploader->Content_Storage_Service:Uploads the raw video file
note over Content_Storage_Service:Segments the video file into chunks
Content_Storage_Service->File_Storage:Stores the segmented files into t
```

```
File_Storage->Video_Encoder:Reads the encoded files from file storage
note over Video_Encoder:Encodes each of the segments in different codec a
Video_Encoder->File_Storage:Encoded segments are stored on the file stora
File_Storage->Video_Distributor:Fetches the encoded file segments
Video_Distributor->CDN:Distributes the encoded file segments on CDN
Video_Distributor->Data_Storage:Persists the CDN info in data storage
```

Content Upload(Modified):

```
participant "Content Uploader" as Content_Uploader
participant "Storage Service" as Storage_Service
participant "File Storage Server" as File_Storage
participant "Video Encoder" as Video_Encoder
participant "Video Distributor" as Video_Distributor

autonumber

Content_Uploader->Storage_Service:Uploads the raw video file
note over Storage_Service:Segments the video file into chunks
Storage_Service->File_Storage:Stores the segmented files into the file st
File_Storage->Video_Encoder:Reads the encoded files from file storage
note over Video_Encoder:Encodes each of the segments in different codec a
Video_Encoder->File_Storage:Encoded segments are stored on the file stora
File_Storage->Video_Distributor:Fetches the encoded file segments
Video_Distributor->CDN:Distributes the encoded file segments on CDN
Video_Distributor->Storage_Service:Persist CDN info in data storage
```

CDN Health Checker:

```
Cron_Job->CDN_Health_Checker_Service:Triggers the process to check CDN he
CDN_Health_Checker_Service<->CDN:Fetches the health metrics and optimal E
CDN_Health_Checker_Service->Data_Storage:Persists the CDN info in data st
```

Title Indexer:

```
autonumber
Cron_Job->Title_Indexer_Service:Triggers the process to index the title
Title_Indexer_Service<->Data_Storage:Fetches the newly uploaded content
note over Title_Indexer_Service:Creates indexes of the video titles
Title_Indexer_Service->Elastic_Search:Updates the title indexes in the
```

```
Playback_Service:

actor Client
autonumber
Client->Playback_Service: Places a play request
Playback_Service<->Authorization_Service:Authenticates user's request
Playback_Service<->Steering_Service:Picks the CDN url from which the pl
Playback_Service->Client:Returns the CDN url to the client
Client<->CDN:Retrieves the content from the CDN
Client->Playback_Service:Publishes events for playback experience
Playback_Service->Playback_Experience_Service:Track events to measure t
```

```
Playback_Service(Modified):

participant Client
participant "Playback Service" as Playback_Service
participant "Authorization Service" as Authorization_Service
participant "Steering Service" as Steering_Service
autonumber
Client->Playback_Service: Content Play request
Playback_Service->Authorization_Service: Authenticate User's request
activate Authorization_Service
Authorization_Service->Playback_Service: Authenticated
deactivate Authorization_ServicePlayback_Service->Steering_Service: Pic
activate Steering_Service
Steering_Service->Playback_Service: Returns CDN URL
deactivate Steering_Service
Playback_Service->Client:Returns CDN URL
Client->CDN:Request the content from the CDN
activate CDN
```

```
CDN->Client: Stream requested MEDIA file
deactivate CDN
Client->Playback_Service:Publishes events for playback experience
Playback_Service->Playback_Experience_Service:Track events to measure the

Content Discovery Service:

actor Client
autonumber
Client->Content_Discovery_Service: Searches for a video title
Content_Discovery_Service<->Elastic_Search: Checks if the video title exi
alt video title exists in the search index
Content_Discovery_Service<->Data_Storage:Fetches details of the video tit
Content_Discovery_Service->Client:Returns the video details to client
else
Content_Discovery_Service<->Content_Similarity_Service:Fetches similar vi
Content_Discovery_Service<->Data_Storage:Fetches details of the similar v
Content_Discovery_Service->Client:Returns the similar video details to cl
end
```

A.7 Uber

```
autonumber
actor Rider
actor Driver

participant Trip_Dispatcher
== Update Driver Location ==
autonumber
Driver->Driver_Location_Manager:
 Updates the driver's current location
Driver_Location_Manager->Car_Location_Index:
Indexes the updated driver location
== List nearby cars ==
autonumber 1

Rider->Trip_Dispatcher:
```

```
Requests for nearby cars to a given location
activate Rider

Trip_Dispatcher->Driver_Location_Manager:
Places query to fetch the nearby cars using the sharding key
activate Trip_Dispatcher
Driver_Location_Manager<->Car_Location_Index:
Fetches the nearby cars within a specific radius
Driver_Location_Manager->Trip_Dispatcher:
Returns the list of nearby cars
Trip_Dispatcher->Rider:
Shows the nearby cars as shown in the image below
deactivate Trip_Dispatcher
deactivate Rider
== Request for a ride ==
autonumber 1
Rider->Trip_Dispatcher: Requests for a ride
activate Rider
Trip_Dispatcher<->Driver_Location_Manager:
Fetches the list of nearby cars
activate Trip_Dispatcher
Trip_Dispatcher->Driver: Notifies driver of the ride request
activate Driver
Driver->Trip_Dispatcher:Confirms the ride
deactivate Driver
Trip_Dispatcher->ETA_Calculator:
Request for fetching the estimated
time for the car to reach the rider
note over ETA_Calculator: Applies the logic to compute the ETA
ETA_Calculator->Trip_Dispatcher: Returns the computed ETA
Trip_Dispatcher->Rider: Provides the driver info and ETA
deactivate Trip_Dispatcher
deactivate Rider

autonumber
actor Driver

== On Trip ==
autonumber
```

```
Driver->Kafka_Stream: Publish GPS location to kafka streams
Kafka_Stream->Trip_Recorder:
Consumes the GPS locations from the Kafka Streams
Trip_Recorder->Location_Store:
Persist the locations in the data store
== On Trip Completion ==
autonumber 1
Driver->Trip_Recorder: Notifies Trip Completion
activate Trip_Recorder
Trip_Recorder->Map_Matcher:
Invokes the process for generating the route map
activate Map_Matcher
Map_Matcher<->Location_Store:
Fetches the GPS locations of the trip
note over Map_Matcher:
Applies the Map Matching algorithm to generate the map route
Map_Matcher->Trip_Recorder:
Returns the matched map route
deactivate Map_Matcher
Trip_Recorder<->Price_Calculator: Fetches the price for a trip
Trip_Recorder->Driver: Returns the map and price information
deactivate Trip_Recorder
```

A

Appendix B

A.1 Instagram

```
autonumber
User_Feed_Service_Queue->User_Feed_Service:
Triggers the process to generate feed for a post
\nfor the followers of the non-celebrity users who made the post
User_Feed_Service->Followers_Service:
 Makes the request to fetch the list of followers
Followers_Service<->Graph_Data_Storage:
Fetches the list of followers of the user
Followers_Service->User_Feed_Service:
Returns the list of followers
loop for all followers
User_Feed_Service->User_Feed_Cassandra_Storage:
Add the post-id as the feed for the user
end
```

A.2 Twitter

```
autonumber
```

```
User_Feed_Service_Queue->User_Feed_Service:
Triggers the process to generate feed for a post
\nfor the followers of the non-celebrity users who made the post
User_Feed_Service->Followers_Service:
Makes the request to fetch the list of followers
Followers_Service<->Graph_Data_Storage:
Fetches the list of followers of the user
Followers_Service->User_Feed_Service:
Returns the list of followers
loop for all followers
User_Feed_Service->User_Feed_Cassandra_Storage:
Add the post-id as the feed for the user
end
```

A.3 WhatsApp

```
actor Alice

autonumber
Alice->Chat_Server_A:Sends image file to Bob
Chat_Server_A->Transient_File_Server:
Uploads the image file to the server
Transient_File_Server->Chat_Server_A:
Returns the image file URL
Chat_Server_A->Alice:Sends image URL back to Alice
note over Alice:Image gets displayed on ALICE's mobile
with the status as SENT

Chat_Server_A->Chat_Storage:
Makes the request for the server to which Bob is connected
Chat_Storage->Chat_Server_A:
Returns the information that Bob is offline
Chat_Server_A->Transient_Server_B:
Forwards the image URL to the transient server
\npartition corresponding to Bob
Transient_Server_B->Transient_Storage:
Stores the image URL to the transient storage

actor Bob
```

```
note over Bob:Bob comes online

Bob->Transient_Server_B:
Requests for transient messages after coming online
Transient_Server_B->Bob:Returns the image URL to Bob
Bob<->Transient_File_Server:
Makes the request to fetch the image content
note over Bob:The image gets displayed to Bob
group Remove transient message references
note over Transient_Server_B:
Remove the transient message reference
note over Transient_File_Server:
Remove the transient message reference
note over Transient_Storage:
Remove the transient message reference
end
Bob->Chat_Server_B:Sends ACK for Alice's image message
Chat_Server_B<->Chat_Storage:
Fetches information about the server to which Alice is connected
Chat_Server_B->Chat_Server_A:
Forwards ACK to the Alice's server
Alice<->Chat_Server_A:
Fetches the ACK during the long polling operation
note over Alice:Image gets marked as DELIVERED

actor Alice
actor Bob
autonumber
Alice<->Bob:Shares public keys with each other
Alice->Chat_Server:Alice encrypts message\nusing Bob's public key
Chat_Server->Bob:Directs the encrypted message to Bob
note over Bob:Decrypts the encrypted message \n using the private key
Bob->Chat_Server:Bob encrypts message using Alice's public key
Chat_Server->Alice:Directs the encrypted message to Alice
note over Alice: Decrypts the encrypted message \n using the private ke

Scenario2
autonumber 2
Alice->Chat_Server_A:Sends message for Bob
```

```
Chat_Server_A->Alice:Sends ACK back to Alice
note over Alice:Message appears as SENT to Alice
Chat_Server_A->Chat_Storage:
Makes the request to fetch the server to which Bob is connected to
Chat_Storage->Chat_Server_A:
Returns the information that Bob is connected to Chat_Server_B
Chat_Server_A->Chat_Server_B:
Forwards the message to the server to which Bob is connected to actor Bob
Chat_Server_B->Bob:Forwards the message to Bob
Bob->Chat_Server_B:Sends ACK for delivery of Alice's message
Chat_Server_B->Chat_Server_A:Forwards Bob's ACK to Chat_Server_A
Chat_Server_A->Alice:Delivers Bob's ACK
note over Alice:Message appears as DELIVERED to Alice
note over Bob: Bob opens Alice's message after an hour
Bob->Chat_Server_B:Sends ACK for having read Alice's message
Chat_Server_B->Chat_Storage:
Makes the request to fetch the server to which Alice is connected to
Chat_Storage->Chat_Server_B:
Returns the information that Alice is connected to Chat_Server_A
Chat_Server_B->Chat_Server_A:
Forwards the ACK for Bob having read the message
Chat_Server_A->Alice:Forwards the ACK
note over Alice:Message appears as READ to Alice


Scenario1(Modified)
participant Alice
participant Chat_Server_A
participant Transient_File_Server
participant Chat_Storage
participant Transient_Server_B
participant Transient_Storage
participant Chat_Server_B
participant Bob
autonumber
Alice->Chat_Server_A:Sends image file to Bob
Chat_Server_A->Transient_File_Server:
Uploads the image file to the server
Transient_File_Server->Chat_Server_A:Returns the image file URL
```

```
Chat_Server_A->Alice:Sends image URL back to Alice
note over Alice:
Image gets displayed on ALICE's mobile with the status as SENT
Chat_Server_A->Chat_Storage:
Makes the request for the server to which Bob is connected
Chat_Storage->Chat_Server_A:Returns the information that Bob is offline
Chat_Server_A->Transient_Server_B:
Forwards the image URL to the transient server
\npartition corresponding to Bob
Transient_Server_B->Transient_Storage:
Stores the image URL to the transient storage
note over Bob:Bob comes online

parallel {
    Bob->Chat_Server_B:Requests for transientmessages after coming onli
    Chat_Server_B->Transient_Server_B:Relay request
}
Transient_Server_B->Bob:Returns the image URL to Bob
Bob->Transient_File_Server:Makes the request to fetch the image content
note over Bob:The image gets displayed to Bob

opt Remove transient message references
note over Transient_Server_B:Remove the transient message reference
note over Transient_File_Server:Remove the transient message reference
note over Transient_Storage:Remove the transient message reference
end

Bob->Chat_Server_B:Sends ACK for Alice's image message
Chat_Server_B->Chat_Storage:
Fetches information about the server to which Alice is connected
Chat_Server_B->Chat_Server_A:Forwards ACK to the Alice's server
Alice->Chat_Server_A:Fetches the ACK during the long polling operation
note over Alice:Image gets marked as DELIVERED
```

A.4 Tinder

```
actor Bob
autonumber
```

```
Bob->RecommendationEngine:
Makes a request to fetch recommendations of other users
RecommendationEngine->GeoShardedIndexer:
Forwards the user request(lat, long, radius)
GeoShardedIndexer->GeoShardedIndexer:
Get all the geoshards we need to query with using Google S2 library
GeoShardedIndexer->GeoShardedIndex:
 Makes the request to fetch nearby users from geo-shards returned by S2
GeoShardedIndex->GeoShardedIndexer:
 Returns the list of nearby users from nearby geo-shards
GeoShardedIndexer->RecommendationEngine: Forwards the list
RecommendationEngine->RecommendationEngine:
Applies filtering based on user preferences
RecommendationEngine->Bob: Returns the list of recommended users
```

A.5 Google Docs

```
title: Edit a Document
== Synchronous Process ==
autonumber
Client->Edit_Patches_Service : Send the edits to the document
Edit_Patches_Service->DocumentManagementService:
Apply patch to the document
Edit_Patches_Service->ShadowCopyManagementService:
Apply patch to the document
Edit_Patches_Service->Client: Returns 2xx response
== Asynchronous Process ==
autonumber
note over DifferenceSyncchronizationService:
 Triggered by a very frequent cron job
DifferenceSyncchronizationService<->DocumentManagementService :
 Fetches the original document
DifferenceSyncchronizationService<->ShadowCopyManagementService :
Fetches the shadow copy of document
note over DifferenceSyncchronizationService: Computes the difference
DifferenceSyncchronizationService<->Client:
Transfers the diff to client to apply the patch using websockets
```

A

Appendix C

A.1 Instagram

```
actor User
autonumber
User->User_Feed_Service: Makes the request to fetch the feed
par thread1
User_Feed_Service<->User_Feed_Cassandra_Storage:
Fetch the feeds for user from non-celebrity users
else thread2
User_Feed_Service<->Followers_Service:
 Fetch the list of celebrities the user follows
loop for all the celebrities the user follows
User_Feed_Service<->User_Activity_Service:
 Fetch the recent posts from the celebrity users
end
end
User_Feed_Service->User_Feed_Service:
 Merge the feeds from celebrity and non-celebrity users based on time
User_Feed_Service->User: Returns the feed to the user
```

A.2 Twitter

```
actor User
autonumber
User->User_Feed_Service: Makes the request to fetch the feed
par thread1
User_Feed_Service<->User_Feed_Cassandra_Storage:
Fetch the feeds for user from non-celebrity users
else thread2
User_Feed_Service<->Followers_Service:
Fetch the list of celebrities the user follows
loop for all the celebrities the user follows
User_Feed_Service<->User_Activity_Service:
Fetch the recent posts from the celebrity users
end
end
User_Feed_Service->User_Feed_Service:
Merge the feeds from celebrity and non-celebrity users based on time
User_Feed_Service->User: Returns the feed to the user
```

A.3 Tinder

```
actor Bob
actor Alice
Alice -> SwipesIngestor: Alice right swipes Bob
SwipesIngestor->LeftSwipesStream:
 Left Swipes are added to a stream
LeftSwipesStream->LeftSwipesS3:
Left swipes added in low-cost storage for data analytics
SwipesIngestor->RightSwipesStream
:SwipesIngestor adds likes to a separate stream
RightSwipesStream->MatchesWorker:
Reads that Alice has right swiped Bob from the stream
MatchesWorker->LikesCache:
Queries cache to check if Bob has also right swiped Alice
LikesCache->MatchesWorker:
 Returns the info if Bob has also right swiped Alice
```

```
alt Bob has right swiped Alice
MatchesWorker->MatchesNotificationQueue:
Sends push notification of the match
MatchesManager->MatchesDB:Persists the matches in the data store
MatchesManager->Bob: Sends match notification using Websockets
MatchesManager->Alice: Sends match notification using Websockets
else Bob hasn't right swiped Alice
MatchesWorker->LikesCache:
Adds info that Alice liked Bob in the cache
end
```

Appendix-D : User Switching Location

```
actor Bob
==Synchronous Process==
autonumber
Bob->LocationService: User requests to change the location
LocationService->UserInfoService:
Makes the request to update user location
LocationService->ElasticSearchUpdationQueue:
Places the asynchronous request to update user location index
LocationService->Bob: Provides confirmation to user
==Asynchronous Process==
autonumber

ElasticSearchUpdationQueue->ES_Gateway:
Reads the request to update the user's index
group Within Ealstic Search Cluster
ES_Gateway->ES_CoordinatingNodes:
Worker node reads the request\nto update index in which user lies
ES_CoordinatingNodes->ES_DataNode:
 Moves the user to the index\ncorresponding to the new geo-shard
end
```

B

Answer Keys

B.1 Distributed System

1. Correct Answer: a Explanation: Monolithic architecture suits simple light-weight applications where all the business logic can be bundled together. On the other hand, microservices architecture is better suited for complex and evolving applications.

2. Correct Answer: b Explanation: Distribution of votes can be skewed, and this will cause hot partitions. This may lead to the server being overwhelmed with the flood of requests.

3. Correct Answer: c Explanation: Microservices tackles the complexity problem by breaking the application into a set of manageable services which are much faster to develop. It reduces the barrier of adopting new technologies.

B.2 CAP theorem

1. Correct Answer: a Explanation: More nodes are involved in making the decision if the write was successful. The reason being that more network hops are included in making

sure that the read after each write in the banking applicaton is consistent across the board.

2. Correct Answer: b Explanation: In scenarios such as comments on a blog, we would like to choose performance over the reliability as users are generally not too concerned if their comments appear eventually after some delay.

B.3 Consistent hashing

Correct Answer: a Explanation: Most of the fetch requests for the objects will be directed to the nodes having a majority of objects resulting in hot partitions.

B.4 Relational vs No SQL

1. Correct Answer: b Explanation: It's apparent that customer features are dynamic and new customer features may get added/remove quite frequently. A NoSQL data store will be an ideal fit for such a use-case as it provides the flexibility to easily maintain the ever-changing customer features.

2. Correct Answer: a Explanation: If you are working with data which is consistent and doesn't need to support a variety of data-types and high volume. In such cases, you may be better off with a relational database.

B.5 Types of No SQL databases

1. Given that all the attributes of a customer can be clubbed together, we would like to choose an aggregate-oriented database. Now, we don't want to fetch the complete aggregate while updating partial attribute values. So, we may want to choose a document data-model over key-value model.

2. Often in analytical queries, multiple rows of the same columns are accessed together. The column-based databases store data on the disk by grouping columns of multiple rows adjacent to each other, thus, resulting in reduced overall IO costs making it the obvious choice in this scenario.

3. We can easily transform the application data into a graph-based data model which provides built in functionality to support traversal queries. In these scenarios, aggregate-oriented databases are not a good fit as traversing between aggregates will lead to sub-optimal performance.

B.6 Tinder

1. a. Explanation: We do geo-sharding to provide recommendations to users which are geographically closer to them. In addition to this, index optimizes the query performance by providing effective data-structures support for data storage and stores the data in memory.

2. b. We can place to query to fetch the items from the key-value data store (like Amazon DynamoDB) to fetch the keys which contains a given userId. This will optimize the computation and I/O cost associated with fetching matches of a user.

C

References

C.1 Distributed Systems

https://en.wikipedia.org/wiki/Clienthttps://ieeexplore.ieee.org/documen
https://en.wikipedia.org/wiki/Mobile$_a$*genthttps*://*en.wikipedia.org*/*wiki*/*Microservice*
//*en.wikipedia.org*/*wiki*/*Fallacies*$_o$*f*$_d$*istributed*$_c$*omputing*

C.2 CAP theorem

https://www.infoq.com/cap-theorem/ https://towardsdatascience.com/cap-t
https://www.amazon.com/NoSQL-Distilled-Emerging-Polyglot-Persistence/dp
https://fenix.tecnico.ulisboa.pt/downloadFile/1126518382178117/10.e-CAP

C.3 Consistent hashing

https://en.wikipedia.org/wiki/Consistent$_h$*ashinghttp*://*www.tom*−*e*−
white.com/2007/11/*consistent*−*hashing.html*

C.4 Relational vs No SQL

https://martinfowler.com/books/nosql.html https://en.wikipedia.org/wiki

C.5 Types of No SQL databases

https://docs.microsoft.com/en-us/azure/architecture/guide/technology-choi
http://www.informit.com/articles/article.aspx?p=2429466 https://www.youtu
//panoply.io/data−warehouse−guide/redshift−columnar−storage−101/https://en.wikipedia.org/w
orientedDBMShttps://en.wikipedia.org/wiki/Graphdatabasehttps://docs.aws.amazon.com/cli/latest/r
item.htmlhttps://docs.aws.amazon.com/documentdb/latest/developerguide/getting−
started.connect.htmlhttps://docs.influxdata.com/influxdb/v1.7/introduction/getting−
started/https://www.youtube.com/watch?v=OoCsY8odmpM

C.6 Instagram

- https://www.youtube.com/watch?v=BfMH4GQWnkhttps://www.youtube.com/watch?v=
  hnpzNAPiC0E

- https://instagram-engineering.com/what-powers-instagram-hundreds-of-insta
- https://instagram-engineering.com/types-for-python-http-apis-an-instagram
- http://highscalability.com/blog/2012/4/9/the-instagram-architecture-faceb
- https://docs.oracle.com/cloud/latest/marketingcsgs/OMCAC/op−
  api−rest−1.0−assets−image−content−post.htmlhttps://instagram−engineering.com/under−
  the−hood−instagram−in−2015−8e8aff5ab7c2

C.7 Twitter

- https://www.infoq.com/news/2015/06/twitter-storm-heron/
- https://www.infoq.com/news/2019/06/machines-understand-emotions/
- https://www.infoq.com/news/2011/09/twitter-storm-real-time-hadoop/
- https://www.infoq.com/news/2017/01/scaling-twitter-infrastructure/?it
  presentationsaboutTwitteritmmedium=linkitmcampaign=Twitterhttps://www.infoq.com/presenta
  notifications/

- https://www.infoq.com/presentations/Twitter-Timeline-Scalability/

C.8 WhatsApp

- https://developers.facebook.com/videos/f8-2016/a-look-at-whatsapp-enq
- http://highscalability.com/blog/2014/2/26/the-whatsapp-architecture-f

C.9 Tinder

- https://medium.com/tinder-engineering/geosharded-recommendations-pa
- https://medium.com/tinder-engineering/geosharded-recommendations-pa
- https://medium.com/tinder-engineering/geosharded-recommendations-pa
- https://medium.com/tinder-engineering/taming-elasticache-with-auto-
- https://medium.com/tinder-engineering/how-we-improved-our-performan
- https://medium.com/tinder-engineering/how-we-improved-our-performan
- https://www.youtube.com/watch?v=Lq4aNihcS8A
- https://www.youtube.com/watch?v=8zh4iUNFjKc
- https://www.youtube.com/watch?v=o3WXPXDuCSU

C.10 Google Docs

- https://quip.com/dev/admin/documentationedit-a-document
- https://neil.fraser.name/writing/sync/

C.11 Netflix

Netflix originals https://www.youtube.com/watch?v=CZ3wIuvmHeM
https://www.youtube.com/watch?v=RWyZkNzvC-c https://www.youtube.com/wat
https://www.youtube.com/watch?v=6oPj-DW09DU https://www.youtube.com/cha
Architecture https://openconnect.netflix.com/Open-Connect-Overview.pdf
https://medium.com/netflix-techblog/mezzfs-mounting-object-storage-in-n
https://medium.com/netflix-techblog/simplifying-media-innovation-at-net
https://www.youtube.com/watch?v=CZ3wIuvmHeM https://www.youtube.com/wat
Data Model https://www.youtube.com/watch?v=OQK3E21BEn8 https://mediu
Fun Facts Chaos Engineering: https://www.youtube.com/watch?v=RWyZkN
Other https://www.youtube.com/watch?v=psQzyFfsUGU https://www.youtub
Written https://medium.com/@narengowda/netflix-system-design-dbec30f
https://openconnect.netflix.com/Open-Connect-Overview.pdf https://mediu

C.12 Uber

Backend of Requesting Ride(Arka) https://www.youtube.com/watch?v=AzptiVdU
Mapping(Kiran) https://www.youtube.com/watch?v=ChtumoDfZXI
ETA Calculation: https://www.youtube.com/watch?v=FEebOd-Pdwg
Uber Payments: https://www.youtube.com/watch?v=Dz6dAZs8Scg
Life of a Trip: https://youtu.be/GyS3m5SyRuQ?list=PLLEUtp5eGr7DcknAkGa
Chaos Engineering: https://www.youtube.com/watch?v=ywSAwYsIk8k