Interview Questions For FAANG

SYSTEM DESIGN AND ARCHITECTURE

BY RAVI TANDON

Chapter 1: The Fundamentals Of System Design

The Elements Of A Distributed Systems

A distributed system is defined as “An interconnected set of nodes that are linked by a network and share information between them.” Although a distributed system consists of multiple components, it serves a single purpose to the system’s user. Some examples of distributed systems in computer science are the backends of Google, Facebook, LinkedIn, etc. Systems like the Internet, Blockchain or even the human nervous system is a distributed system. For the sake of this blog series, we will focus on only those in Computer Systems 😊😊

We will discuss some of the critical elements of a typical distributed system for building user-facing applications next.

Microservices

The microservice architecture is a design paradigm where an application is structured to collect several independent services. The characteristics of these services are as follows:

- Organized around a specific business capability
- Own by a small team

- Independently deployable
- Highly scalable
- Loosely coupled
- Highly maintainable and testable

Load Balancers

A load balancer is a device that acts as a reverse proxy and distributes the network traffic across several different servers. Load balancers are used to scale distributed systems by distributing traffic between other servers. Load balancers are typically grouped into two distinct categories: Layer 4 and Layer 7. The load balancers at layer 4 act upon data found in network and transport layer protocols (IP, TCP, FTP, UDP). Layer 7 load balancers distribute requests based on application data (HTTP). Some of the standard algorithms that load balancers use are:

- Round Robin
- Weighted Round Robin
- Least connections

- Least response time

Databases

A database is a structured system to put your data under a pre-specified format and constraints. Some of the requirements when a database is needed are as follows:

- Scalability: Applications that have billions of rows or terabytes of data use DBMS.
- Security: Applications that require a strict deposit of access to data in an organization use DBMS.
- Consistency: Applications that require the data to remain consistent use DBMS. Some of how the data can become corrupt are as follows:
    - Lack of consistency constraints
    - Unsafe deletes
    - Overwriting

A typical DBMS provides ACID capabilities to ensure that the data remains consistent.

- Availability: Applications that require the data to remain available throughout the lifetime use DBMS. A typical DBMS provides replication of servers to ensure the availability of data.

Caches

A cache is temporary storage that is relatively smaller in size with faster access time. Caches are used to reduce latency and reduce the load on your servers and databases. There are several levels at which caches are implemented. These are as follows:

1. Client Caching: Caches are located on the client-side like browser, file-system, servers acting as reverse-proxy.
2. CDN Caching: CDN (Content Delivery Network) is also used as a cache for serving static content (eg. HTML, CSS, Javascript, Image, Videos, etc.).

3. Web Server Caching: Web Servers also implement local caches and can retrieve, return information without contacting downstream servers.
4. Database Caching: Databases by default include some level of caching so that they do not have to run queries repeatedly. This can boost the performance of databases when they are under a lot of load.
5. Application Caching: In application caching, the cache is placed between application and data stores. These caches usually store database query results and/or objects that the application uses. Typical application caches include Memcached, Redis, DynamoDB, etc.

One of the challenges with caching is ensuring consistency of the data between the cache and the underlying data layer (i.e. server or database).

File System Storage

A file is an unstructured collection of records. Typically file systems support two basic formats. These are as follows:

- Block Storage: Organize data in blocks on disk. The blocks are then divided into sectors and tracks. Current day personal computers (i.e., laptops and desktops) store data using Block Storage.
- Object Storage: Organize data into containers of flexible sizes, referred to as objects. Modern-day cloud systems use Object Storage for storing their data. E.g., Amazon S3. They are designed to be massively scalable, highly performant, and inexpensive.

The characteristics of a distributed file system (DFS) are as follows:

- Performance: A DFS should provide high throughput and low latency of access. The performance should be similar to what a local file system can provide.
- Availability: Distributed file systems are built across a network. Therefore, they can have several possibilities of failure. The data should remain accessible in case of partial node failures.
- Scalability: A DFS should scale horizontally and support the storage of petabytes of data. The system should scale as more

nodes are added to the DFS. The client should not experience any disruption as more nodes are added to the system.

- Reliability: A DFS should be highly reliable and ensure that the probability of data loss is minimized. Typically, a DFS would create backup copies of the data to prevent it from being lost permanently.
- Ease Of Use: A DFS should provide a simple and easy-to-use interface for the client to use. Some of the common operations that clients use, such as reading, writing, copying, etc., should be easy to perform.
- Data Integrity: A DFS should ensure that the data remains consistent and shouldn't be lost in case multiple users access it. Typically DFS should provide atomicity and consistency of operations in some form or the other.
- Heterogeneity: A DFS should support the storage of heterogeneous data (structured, unstructured, files, records, etc.). Different types of clients should be able to access the file system and store data on it.

Network

The final and central piece of a distributed system is the network that binds all the nodes together. Typically, clouds such as Amazon's AWS, Microsoft's Azure, etc., provide the infrastructure for networks on which the servers can run. These networks span Wide Area Networks and are spread across geographies in the world. The cloud networks are typically designed to enable the client/server communication model using protocols such as HTTP.

Messaging Queues

Messaging queues make it possible for services to communicate with each other asynchronously. The typical use cases of messaging queues tend to be sending notifications to clients. These notifications can be alerts, emails, messages, etc. The basic architecture of a messaging queue consists of the following:

1. Producer: The service that produces messages and writes them on the messaging queue.

2. Consumer: The service that consumes messages and reads them off the messaging queue.

The essential characteristics of a messaging queue are as follows:

1. Scalable: A messaging queue should be able to scale to millions of messages per hour. It should scale horizontally and preserve the basic guarantees when more nodes are added to the system.
2. Distributed: A messaging queue should be distributed to scale up and support multiple consumers, producers, and messaging using a cluster of nodes.
3. Reliability: A messaging queue should be reliable and not drop messages. If the error rate is high, i.e., many messages get settled, then the burden falls on the client to perform checks on the data. This makes the message queue inefficient to use.
4. Performance: A message queue should support high throughput and low latency when delivering messages.

5. Easy to use interface: A messaging queue should provide a simple API interface to integrate with the client applications (i.e., producers and consumers) for broader adoption.

References

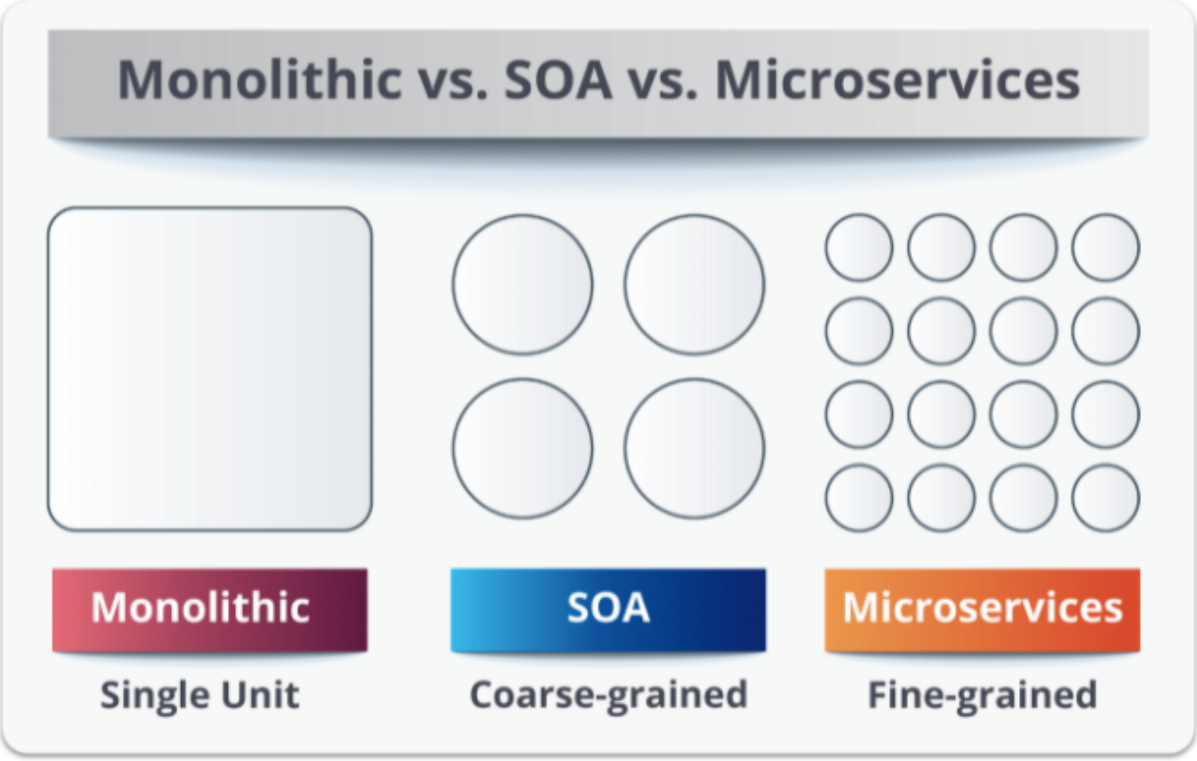
1. What are microservices?
2. Load Balancer
3. Database Introduction
4. ACID Guarantees Wikipedia
5. What is Caching?
6. What is DFS (Distributed File System)?
7. Distributed Networking (Wikipedia)
8. System Design - Message Queues
9. Kafka: A Distributed Messaging System For Log Processing

Chapter 2: Principles Of The Microservice Architecture

Introduction

The Microservice Architecture is a design paradigm where a large complex application is broken down into a suite of small function component services that have the following characteristics:

- Loosely coupled
- Highly maintainable and testable
- Independently developed, deployed, and maintained
- Organized around business capabilities
- Owned by a small team



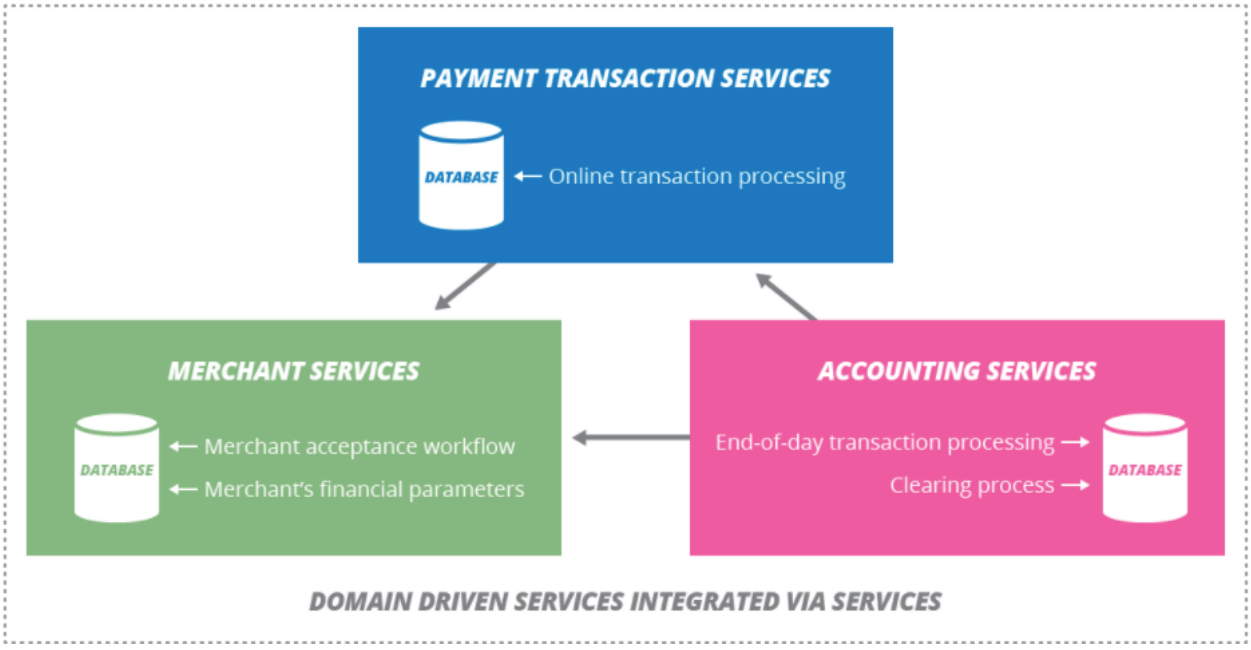
The image above (Courtesy: Edureka. co) shows the primary distinction between a Monolithic, Service Oriented, and Microservices Architecture. The monolithic architecture consists of a single large component that encapsulates the application. The Service-Oriented Architecture (also called SOA) breaks the services into coarse-grained modules while the Microservices architecture breaks them into very small granules.

Principles Of The Microservice Architecture

This section will delve deeper into the core seven principles required for a Microservice Architecture to work well.

Domain-Driven Design

A typical application usually caters to a collection of business domains. A domain is an independent area of a problem space that has abounded context and emphasizes a common language to talk about the problems.



Let's take the example of a typical credit card processing application. In such an application, there are primarily three different domains in the payment processing workflow. These are as follows:

- Merchant Subdomain: Handles financial parameters & merchant acceptance workflow.
- Accounting Subdomain: Handles end-of-day transaction processing, clearance, & reconciliation.
- Payment Subdomain: Handles online transaction processing.

Given the above subdomains, the application can be broken down into multiple microservices, and separation of concerns can be thus established.

Microservices are, therefore, organized and built around encapsulated business capabilities. A business capability represents the function that the business unit does in a particular domain. Incorporating Domain-Driven Design allows the architecture to isolate system ability into various domains.

Culture Of Automation

Slicing up a monolithic application into a suite of component services in a highly decomposed ecosystem naturally lends itself to independently

deployable units. Organizations that adopt the microservices architecture should adopt practices that lead to automation of microservices. Some of the functions that can be automated include builds, testing, deployments, alerts, etc., across microservices.

Infrastructure Automation

Infrastructure as Code (IaC) is configuring and managing the infrastructure through a descriptive model. It is all about treating your infrastructure configuration and provisioning the same way you treat your application source code. The last decade has seen significant advancements in IAAC (Infrastructure As A Code). Tools such as Terraform, Ansible, AWS CloudFormation, Azure Resource Manager, Google Cloud Deployment Manager, etc., make the provisioning of infrastructure (servers, load-balancers, caches, databases, etc.) seamless.

Testing Automation

Given the decentralized model, the microservice architecture requires rigorous testing to ensure that the system is stable when released. This requires investment from many different tools to ensure that the Microservices are sufficiently tested pre-release. Some of these tools

include Apache JMeter, Gatling, Jaeger, Cypress, etc. These tools help teams write scalability, end-to-end tests and monitor and observe the system behavior when the tests are run.

Continuous Delivery

Continuous delivery is a software engineering approach in which teams produce software in short cycles, ensuring that the software can be reliably released at any time and, when removing the software, without doing so manually. It aims at building, testing, and releasing software with more incredible speed and frequency. To make the delivery of applications seamless and low-overhead, it is necessary that each of the microservices can be deployed through the continuous delivery model. In this model, teams need to invest in testing automation and delivery platforms such as Harness, Spinnaker, Jenkins, Gitlab, etc.

Automated deployment. Image from red-gate.com

Encapsulation: Hiding Implementation Details

The principle of encapsulation states that a service should only be consumed through a standardized API and that it shouldn't expose its internal implementation details (business logic, auth implementation, persistence logic, etc.) to its consumers. Let's consider the design of a system with the following two strategies:

Design 1

A single shared database with multiple services accessing it.

Design 2

Each service owns its database schema and exposes interactions through APIs.

In design 1, changes to the database schema such as deleting a column, changing the column's data type, and adding a new column will most probably require changes upstream to many different services. This can cause unknown ripple effects, introduce bugs and require high maintenance. Therefore going with design 2 makes much more sense.

How To Design For Encapsulation

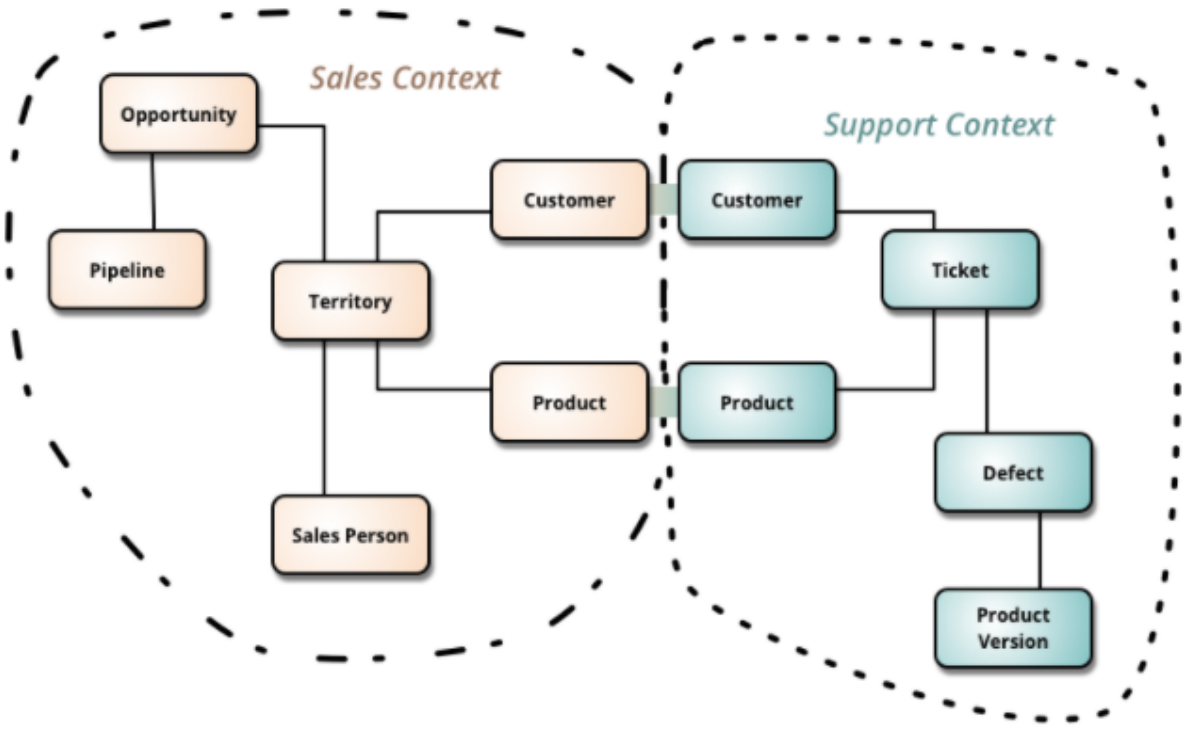
We will talk about two important concepts, namely APIs and Bounded Contexts, to better understand the idea of encapsulation and separation of concerns.

Application Programming Interfaces

Application Programming Interfaces, or APIs commonly referred to, are interfaces defined between components for exchanging information based on particular actions. Common APIs are designed to follow the CRUD (Create, Read, Update, and Delete) model. APIs expose only a limited set of information and in a standard contract that hides the complexity of the underlying data model, persistence layer, networking architecture, etc.

Bounded Contexts

Bounded Contexts are a class of components that share a shared understanding of concepts. They are a central pattern in Domain-Driven Design (DDD). DDD deals with large models by dividing them into different Bounded Contexts and being explicit about their interrelationships.



Let's refer to the image above. There are two different bounded contexts. These are the Sales Context and the Support Context. The Sales Context deals with concepts such as Opportunity, Pipeline, Territory, Sales Person exclusively. The ideas such as Customer and Product are shared between Sales and Support Contexts. Similarly, the Support Context understands concepts such as Ticket, Defect, Product Version exclusively, and Customer, Product in a shared manner.

Why Is Bounded Context Important To Consider

It is essential to understand that the same concept, such as a Customer, may have a different meaning based on their context. Let's take an example. In the Sales Context, the Customer may mean a client I have sold my product to, while in the Support Context, it may refer to a client who has raised a support ticket. While designing the APIs, it is essential to consider what information to share vs. what information to hide. It is easier to share confidential details incrementally than to hide information that is already shared. If the Support Context only requires information about the tickets that a customer has raised in the past, then sharing the complete customer object may not be a good idea.

Decentralization

Let's first understand why decentralization is needed in a microservice architecture. To build highly scalable and resilient systems, individual component owners require *autonomy.* Autonomy is giving people as much freedom as possible to do the job at hand. Some of the common principles to follow when building a decentralized architecture are as follows:

Self-Service

The idea of self-service consists of providing the teams the ability to provision resources (such as virtual machines, service instances, storage units, etc.). Traditionally, the cycles to get any resource provisioned would require teams to raise tickets; the ticket may have to go through approval from the IT department. This would significantly slow down the overall process and make the process of software development slow. With the advent of the DevOps model, and the availability of hyper-scale cloud platforms such as AWS, GCP, Azure, etc., it has become significantly easier for teams to be self-served and managed.

Shared Governance

Centralized governance suffers from the problem of solving every problem using the same hammer. They work on the philosophy of standardizing on single technology platforms even though there's never one right for all jobs. Splitting a Monolith into several microservices that are loosely coupled services allows conscious and customized tech-stack choices when building each of them, typically along the lines of *what fits the job at hand best.*

This article by the team at Gilt talks about a shared governance architecture, where they disbanded a “central architecture team.“ Under this model, each microservice team has its own set of “architects.” They then comprise an “architecture board.” Their general purpose is to ensure the sharing of best practices and to identify the very few standards we need to govern– mainly where technology crosses team boundaries (e.g., how software built by one team talks with software built by another team).

Smart Endpoints And Dumb Pipes

The microservice architecture posits that the business logic should be encapsulated within the microservices. The communication medium (such as message buses) should be dumb. This is very different from SOA’s Enterprise Service Bus (ESB) model, which includes sophisticated features of message routing, data transformation rules, understanding of the business context, etc., in the message bus. The major drawback of making the pipes smart tends to be that changes in the business logic will require downstream changes in microservices, message buses, and

other microservices. This can lead to significant bugs in the system and increase the cost of incremental changes and maintenance.

References

1. Maximizing Microservices Architecture Part 1: The What and Why
2. Maximizing Microservices Architecture Part 2: Principles
3. Principles Of Microservices by Sam Newman
4. Domain-Driven Design for Services Architecture
5. Top 10 Infrastructure as Code Tools to Boost Your Productivity
6. Bounded Context By Martin Fowler

Chapter 3: System Design: Challenges In Distributed Systems (Availability)

Introduction

In the last two chapters, we deep-dived into the basic building blocks of a distributed system and the principles of Microservice Architecture. This blog will go through the common challenges that have to be considered while designing a distributed system. Specifically, we will focus on the CAP Theorem and discuss the availability of a distributed system.

CAP Theorem

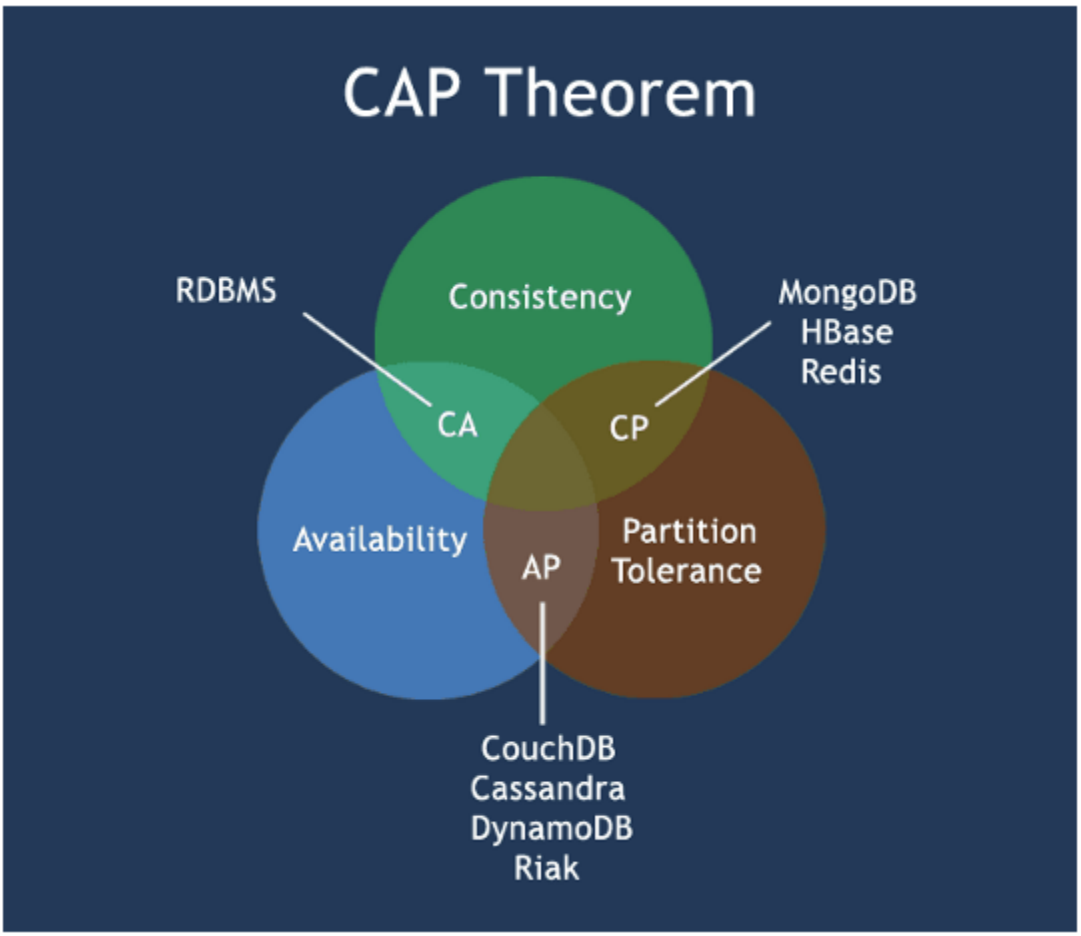
*The **CAP (Consistency, Availability, and Partition Tolerance**) theorem states that in networked shared-data systems or distributed systems, we can achieve two of three guarantees for a database: Consistency, Availability, and Partition Tolerance.*

The CAP Theorem is also known as Brewer's theorem and was introduced by the computer scientist Eric Brewer at the Symposium on Principles of Distributed Computing in 2000.

Consistency: Consistency means that all clients see the same data simultaneously (for strict consistency), no matter which node they connect to in a distributed system. For eventual consistency, the guarantees are a bit loose. Eventual consistency guarantees that clients will eventually see the same data on all the nodes at some point of time in the future.

Availability: Availability means that all non-failing nodes in a distributed system return a response for all read and write requests in a bounded (or reasonable) amount of time, even if one or more other nodes are down.

Partition Tolerance: Partition Tolerance means that the system continues to operate despite arbitrary message loss or failure in parts of the system. Distributed systems guaranteeing partition tolerance can gracefully recover from partitions once the partition heals.



Types Of NoSQL Database Systems

With the above theorem, we can classify the systems into the following three categories:

1. **CA Database**: A CA database delivers consistency and availability across all the nodes. It can't do this if there is a partition between any two nodes in the system and therefore

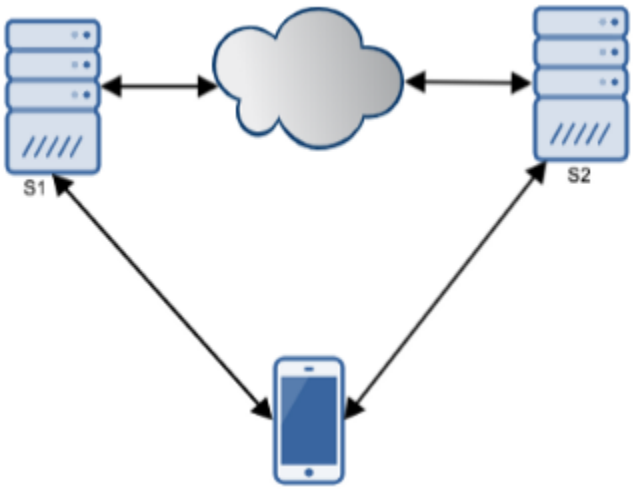
doesn't support partition tolerance. *RDBMS* are good examples of these databases.

2. **CP Database**: A CP database delivers consistency and partition tolerance at the expense of availability. When a partition occurs between two nodes, the system shuts down the non-available node (i.e., makes it unavailable) until the partition is resolved. Some of the examples of these databases are *MongoDB, HBase, and Redis*.
3. **AP Database**: An AP database delivers availability and partition tolerance at the expense of consistency. When a partition occurs, all nodes remain available, but those at the wrong end of a partition might return an older version of data than others. (When the partition is resolved, the AP databases typically resync the nodes to repair all the inconsistencies in the system). Some of the examples of these databases are *CouchDB*, *Cassandra*, and *DynamoDB*, etc.

Note: As a part of this course, we will discuss these technologies in detail in the subsequent blogs.

Proof Of The Cap Theorem

Let's look at a simple proof of the Cap Theorem using the diagram below.



In the figure above, we have a straightforward distributed system where S1 and S2 are two servers. The two servers can talk to each other. Let's start with the assumption that the system is partition tolerant. We will prove that that system can be either consistent or available.

Suppose there is a network failure and S1 and S2 cannot talk to each other. Now assume that the client makes a write to S1. The client then sends a read to S2. Given S1 and S2 cannot talk, they have a different view of the data. If the system has to remain consistent, it must deny the

request and thus give up on availability. If the system is available, then the system has to give up on consistency.

This proves the CAP theorem.

Addendum: Spanner, Google's highly available global-scale distributed database, seemingly achieves consistency, availability despite being a globally distributed database. We will discuss how it achieves the impossible in a follow-up blog.

Availability

The availability of a distributed system is usually defined as the ratio of the total time that the system is up (i.e., can serve requests from the client within a bounded time) to the overall running time of the system. For example, Amazon's S3 system guarantees an availability SLA of 99.99% over a given year.

Why is availability critical for businesses?

Due to a recent outage to Facebook services (Facebook App, Messenger App, and WhatsApp), Facebook lost an estimated USD 100 million. Last year, an outage in AWS took down a good chunk of the internet and caused many websites to be non-functional for a few hours. According to Gartner, companies lose an average of $336,000 per hour of downtime, with top e-commerce sites risking up to $13 million in lost sales per hour of downtime.

For any application to work well, they must remain available during their lifetime (esp. during peak hours). If the services go down, the users may experience high latencies, service disruption, or even data inconsistencies (if the application is PA). It can be severely detrimental to the users.

How is system availability measured?

This section discusses the three techniques that are used to measure the availability of a distributed system.

Time-Based

The most common measure of system availability is Time Based. In their book, Service Availability: Principles and Practice, Toeroe and Tam defines service availability as

Availability = MTTF / (MTTF + MTTR)

where MTTF is Mean Time To Failure and MTTR is Mean Time To Recovery. The Mean Time To Failure is the average time between the end of one outage and the start of another. Mean Time To Recovery is the average time taken to recover from outages.

Request Count Based

Count-based measurements use the success ratio of the received requests. The definition of availability can thus be measured as:

Availability = number of successful requests / number of total requests

The downside with this approach is that it is prone to bias. Power users tend to be 1000x active than the least active users and thus are 1000x more represented by this metric. So, if the system is being actively used by its power users, it will seem to be much more available than when it is

not being used. It seems incorrect. Furthermore, it doesn't capture the change in availability appropriately. Let's say a system is up 3 hours and down for 3 hours. There will be many more requests during the 3 hours (when it is available) than during the 3 hours when it was down. It brings in misrepresentation and bias.

User Uptime

In their paper, Meaningful Availability, Google came up with a new user metric, user-uptime, as follows:

$$User - Uptime = \frac{\sum_{u \in users} uptime(u)}{\sum_{u \in users} (uptime(u) + downtime(u))}$$

Here are the definitions for the terms used in the above mentioned formula:

1. *uptime(u): Defines the cumulative time for which the system was up for all users.*

2. *downtime(u): Defines the cumulative time for which the system was down for all users.*

It is a very conservative metric, where we consider any request that fails as contributing to the downtime even if the user doesn't perceive it. Let's take a concrete example to understand this better. For instance, while uploading an attachment to an email, the initial request might have failed. But an automatic retry done by the system may have succeeded. So the user may not even have noticed the downtime. The paper suggests that those few milliseconds when a request fails should still be considered as downtime.

Techniques For Availability

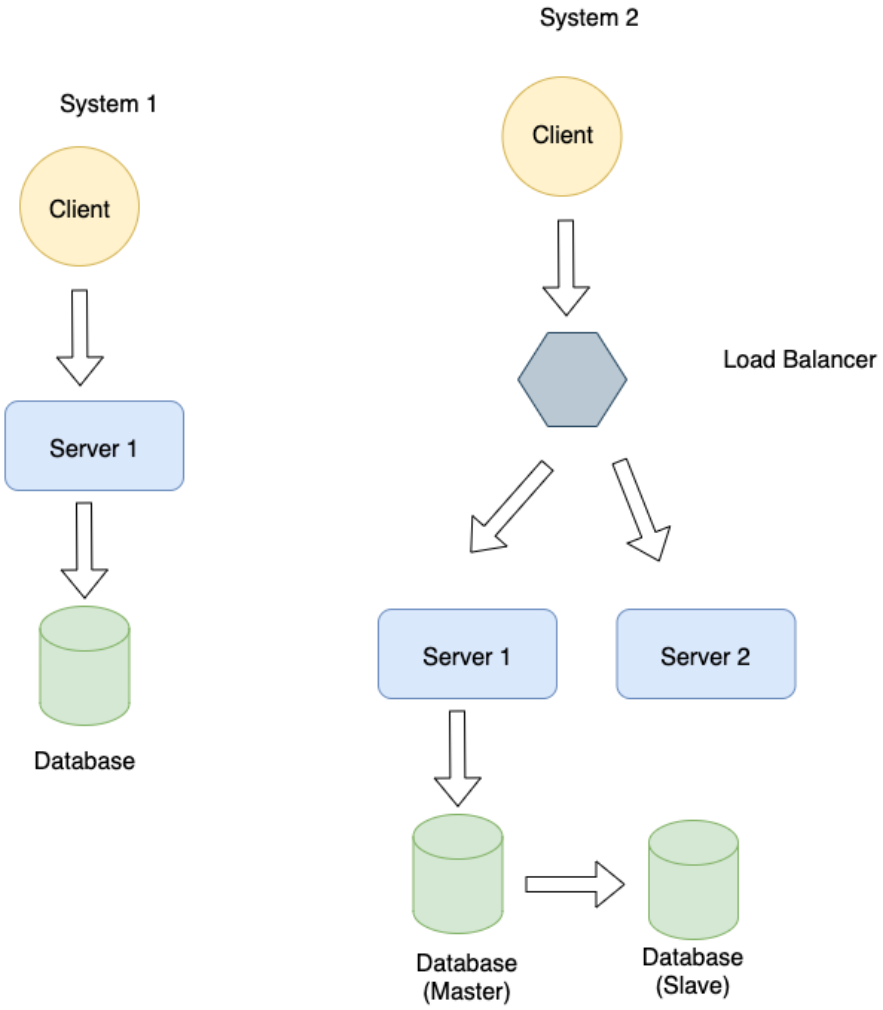
In this section, we discuss some of the techniques used to improve a system's availability. These are as follows:

Redundancy

One of the most common techniques to improve a system's availability is to introduce redundancy. Redundancy involves adding multiple copies of the same subsystem into the architecture. Statelessness and

load-balancing are often employed to make redundancy work well for the availability and scalability of the system. Let's take an example to explain this better. Consider the following scenarios. System 1 has a single client, a backend server connected to a single database replica. System 2 has a client, a load-balancer, two backend servers, and a master-slave replica.

The question then becomes which system is better. If we think about it, System-1 has multiple points of failure. Both the server and the Database engine can fail and make the system unavailable. On the other hand, System-2 can handle multiple failures. If server 1 fails, then server-2 can respond to the incoming requests. Similar is the case with the database engine. If the master fails, then the slave can handle requests from the upstream service. This makes System-2 much more robust than System-1.



Monitoring

Besides improving the service through redundancy, monitoring is another technique used to ensure that services remain available. In this post, we will use the term "*Observability*" interchangeably with *Monitoring*. In general, there are three layers on which monitoring is done. These are *network*, *machine*, and *application*. The most critical layer includes application metrics. It is the hardest as well.

Four Pillars Of Observability

There are four pillars of the Observability Engineering team's charter:

- Monitoring
- Alerting/Visualization
- Distributed systems tracing infrastructure
- Log aggregations/analytics

References

1. CAP Theorem and NoSQL Databases. What is the CAP theorem? | by Barmanand Kumar
2. System Design for Beginners -CAP Theorem | by Bibhu Pala
3. CAP Theorem. CAP theorem also known as Brewer's... | by Vivek Kumar Singh | System Design Blog
4. Towards Robust Distributed Systems
5. Spanner, TrueTime & The CAP Theorem
6. Object Storage Classes – Amazon S3
7. Prolonged AWS outage takes down a big chunk of the internet
8. Facebook outage: Lost ad revenue, advertisers could seek refunds
9. Measuring Availability of a Service, Meaningfully | by Mahendra Kariya
10. Meaningful Availability
11. What is the CAP Theorem? - India
12. Monitoring and Observability. During lunch with a few friends in late... | by Cindy Sridharan | Medium
13. Measure Anything, Measure Everything

Chapter 4: Designing An Auto-Complete Engine From Scratch

Functional Requirements

The system should be able to satisfy the following functional requirements:

1. Search for strings from a collection of predefined strings.
2. Provide completions from the set of strings indexed. The type of matches we will need are prefix matches only.

To understand the requirement, let's assume that you have a set of all the cities listed. Let's assume that you are building a travel application, and so you need to return to the cities that are traveled to most often. Here is a list:

- London, England.
- Paris, France.
- New York City, United States.
- Moscow, Russia.
- Dubai, United Arab Emirates.
- Tokyo, Japan.

- Singapore.
- Los Angeles, United States.
- Barcelona, Spain.
- Madrid, Spain.

Now let's say the user types in the letter, D. The engine should suggest the following:

1. Dubai,
2. Delhi,
3. Damascus
4. .....

Note: This list is ranked in order of its popularity.

If the user types in Dubai, the engine should recognize that it is the city of Dubai.

Non-Functional Requirements

1. **Latency**: The engine should return results within *10 ms*.
2. **Scalability**: The engine should be able to index *10 Billion strings*.
3. **Throughput**: The engine should support a *QPS of 1000 requests/hour.*
4. **Availability**: The engine should experience zero downtime.
5. **Customizability**: The engine should be customizable.
6. **Memory** + **Disk Availability**: We can assume an infinite amount of memory and disk available.

System Metrics Every Engineer Should Know

Before we begin solving the problem, it is essential to understand the most common system metrics. These are as follows:

1. L1 Cache Reference: 0.5 ns
2. Latency of access to main memory: 100 ns
3. Latency of access to a solid-state drive: 100 us

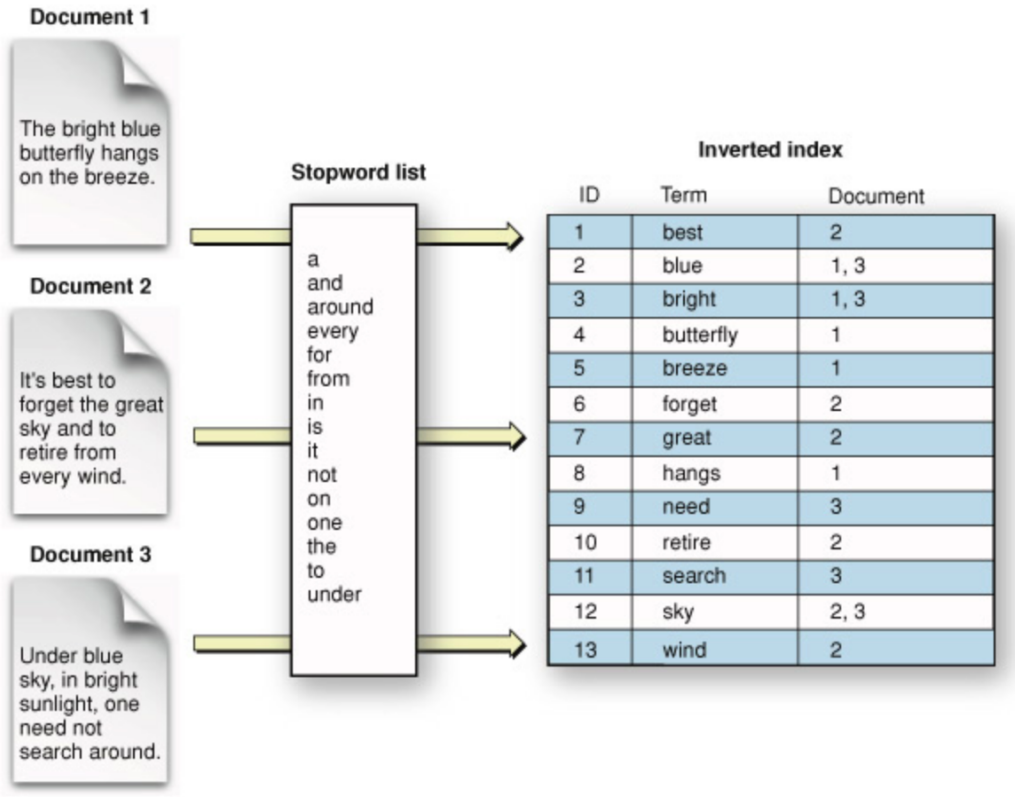
4. Latency of access to hard disk: 10 ms
5. Round trip within the same data center: 500 us
6. Maximum memory in 1 web server: 256 GB

Data Structure Design

Let's first start with the design of the data structure for the search index. Typically, there are the following two different data structures that one can use for searching information:

Inverted Index: Inverted Index, also known as the postings list or inverted file, is an index storing scheme mapping content, such as words or numbers, to its location in a table. Let's take the following example. In the image below, we can see a list of words derived from their respective documents. Each word (also called a term) has an associated ID and the Document number associated with it.

While Inverted Indices are good for searching complete terms, they are not amenable to prefix, suffix matches, etc.



Binary Search Tree: A Binary Search Tree is a well-understood data structure. It has the following properties:

1. Each node has two children.
2. The left subtree of a node contains only nodes with keys lesser than the node's key.
3. The right subtree of a node contains only nodes with keys greater than the node's key.

The comparison can be defined using lexicographic comparison for strings.

The search complexity in a well-balanced binary search tree will be $M * log N$, where M is the length of the longest string and N is the number of keys in the tree.

Trie: A Trie is an efficient information retrieval data structure. Tries are built on top of the same idea as a tree but are optimized for storing information for faster retrieval. The search complexity can be brought down to the optimal limit (i.e., key size). A trie makes a trade-off on the time-complexity (i.e., reduces the time-complexity) but requires a higher amount of space (increases the space-complexity).

When comparing the data structures, tree-like data structures support prefixes and exact search semantics much better than a Hash-Table. When we reach a tree with a trie, a trie becomes efficient in terms of time complexity.

We, therefore, will go with a trie-like data structure.

Memory vs. On Disk Structure

Regarding the system design, the next question to answer is whether we should keep this data structure all in-memory vs. keep it on disk vs. memory.

Let's do the following calculations to understand this better:

Memory Requirements

Let's assume that each node represents a city. A city will have some metadata associated with it. It may be the city's name, synonyms, country, geo-location, population, popularity, etc. Let's assume that we have 100 *Bytes* of data. Considering *10 Billion* strings, this would come out to be 1 TB data. We can safely assume that we cannot keep this in a single off-the-shelf web server. We are assuming that a typical off-the-shelf web server has *256 GB* of memory capacity.

Data Layout

1. **All Data On Disk:** In this design, we will assume that all the data is on disk (for argument's sake). Let's assume that a typical access pattern would look like at least two different disk seeks. I am assuming that we are retrieving ten results. And, optimistically, they are spread over two separate pages. This would generally take 20 ms if the pages are not in the memory buffer and are non-contiguous. In general, the tail-latency would be high if the disk seeks to follow a very random pattern. Given our requirement of 10 ms lookup latency, this seems like a No-Go.
2. **Data In Hybrid Store**: This design assumes that 20% of the data is kept in memory, and 80% resides on a disk. Considering that most of the hot data (i.e., frequently accessed) can be cached in memory, this technique can be cost-efficient and perform better. The tail latencies will still involve access to the disk, which may incur 10+ ms of access latencies.

All The Data In Memory: In this design, we will assume that all the data is in-memory. The cost of access will be low, i.e., 10 - 100 ns per memory reference. The bottleneck, therefore, shifts away from data access to

compute (< 10ms). Given that we have an infinite amount of memory available (we will discuss how to achieve this), we should keep all the data in memory.

Data Structure Design

Let's now design the core data structure as follows:

```
struct NonLeafNode {
  vector<Node *> children;
  String value;
};


struct LeafNode {
  vector<Node *> children;
  String value;
  // Metadata
```

```
    double popularity;

    String country;

    double population;

    .....

};
```

System Architecture

Scalability

Let's next discuss how the system should scale. For the application to scale to 1 TB of in-memory, we need to support scalability. The approach we will take is horizontal scaling over vertical scaling (Read more about scaling here).

Vertical Scaling:

Pros:

- Ease of implementation.

Cons:

- Limitation on the scale.
- Hardware becomes complex and expensive as the size increases.

Horizontal Scaling:

Pros:

- Provides infinite scaling (through elasticity).
- Provides more fault tolerance (through redundancy).
- Cheaper (can be implemented with off-the-shelf hardware).
- No need to rely on complex, specialized hardware.

Cons:

- Introduce the complexity of implementation.

With horizontal scaling, we will divide the data into multiple servers. Let's assume that we need to store 1 TB of data and each server has 256 GB of memory. Assume that the overhead of trie is 50%. We need nearly 2 TB of memory. This amounts to 8 web servers.

Sharding

To divide the data between the eight nodes, we will go with sharding (Read more about sharding here). There are several ways to shard the data. We will discuss the pros and cons of each in the following section:

Sharding the data alphabetically: The most basic scheme we can use is sharding the data using an alphabetic scheme. The diagram below shows the distribution of data, sharded across the nodes. The advantage of this scheme is its simplicity of implementation. The drawback of this scheme is that it can cause excessive pressure on specific nodes. The nodes that hold data with A, R, or S will have many strings indexed. This is due to a skew of data distribution in the English language.

Sharding Data With Alphabetic Distribution

Sharding the data randomly: In this approach, we uniformly sample each node and store the data using a method similar to random trees. The main idea is to distribute the data randomly (using a pre-defined

hash function) and ensure that the load is evenly distributed. Even though the load remains evenly distributed, there are certain drawbacks to this approach. In case of a server failure, all the data needs to be re-distributed, which can cause upstream disruption.

Sharding the data using Consistent Hashing: Consistent hashing (blog, paper) is a technique that uses the idea of distribution using random trees but takes a slightly different approach concerning data distribution. Like other hashing schemes, consistent hashing assigns a set of items to buckets so that each bin receives roughly the same number of items. Unlike standard hashing schemes, though, a small change in buckets gives only a slight change in the assignment of items to buckets. Consistent hashing achieves this by building "*ranged hash functions.*" A ranged hash function ensures *smooth distribution*, a *low spread of keys*, and a *low load on each bucket*. (*Note: We will cover the intuition behind consistent hashing in a different blog*).

Consistent Hashing is the most robust and performant scheme among the schemes mentioned here and is used commonly.

Availability

One of the problems with keeping all the memory is that a part of the data may become unavailable for a significant period if the server goes down. This is because reading the data from the data source (from the disk) and building the index in memory is a slow process. The user will not access the part of the data that is currently not in memory.

Introducing Redundancy In the Design

To ensure high availability, we introduce redundancy in the system. Each shard is now present on two different nodes rather than on a single node.

System Architecture With Redundant Shards

Let's look at the image above. We can see that each shard (e.g., A-D, X- Z) is now present on two nodes instead of one.

Pros:

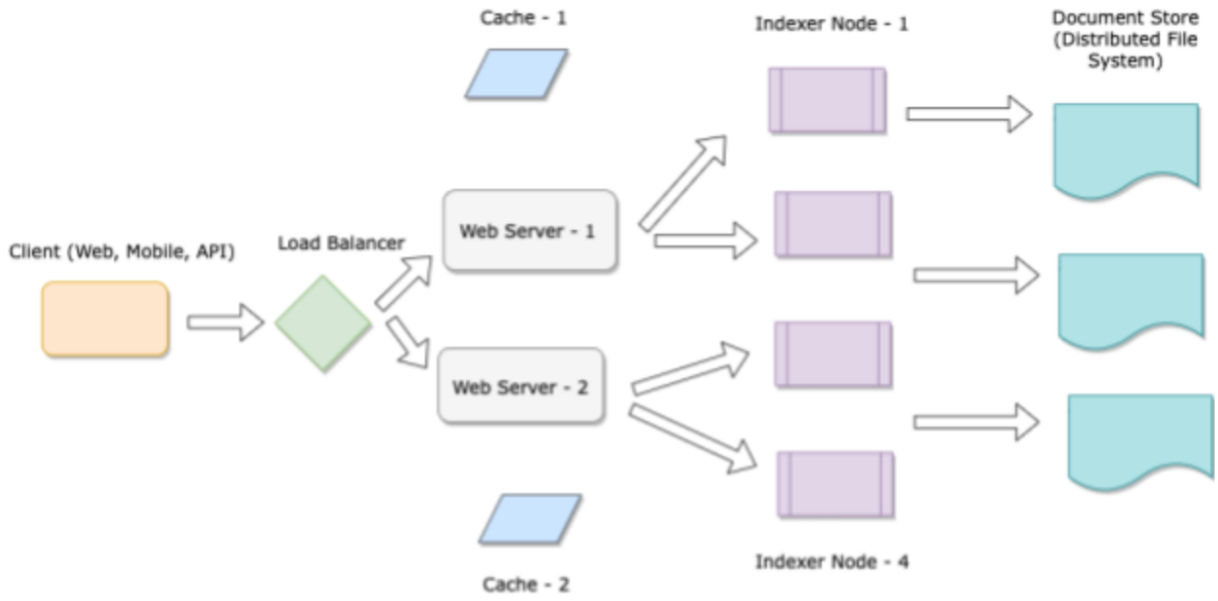
- This design ensures that even if one of the shards is unavailable, the other shard can serve the requests.
- In case of high pressure on one node, requests can be redirected to the same shard.

Cons:

- This increases the memory requirement of the system.
- This introduces extra complexity of implementation (shard-routing, synchronization of shard data, etc.).

High-Level Diagram And Data Flow

Here we discuss the high-level view of the search engine and the data flow diagram.



High Level Diagram And Data Flow View

The major components of the system are as follows:

1. **Client:** The client consists of a Web App, Mobile App, or API end-point that hits the backend services.
2. **Load Balancer**: The load-balancer is used to route requests to the webserver and balance the load between the two services.
3. **Web Server**: The web servers encode the business logic. In the auto-complete engine, the web-server will do the following:
    1. Validate the request.

    2. Check with the caches if the results for the request are already cached. If not, hit the indexer nodes.
    3. Find the indexer node on which the request should be routed.
    4. Send a request to the correct set of nodes (using the Consistent Hashing Scheme).
        1. If the request is for an exact match, then only one node needs to be looked up.
        2. If the request is for a prefix match, one or two nodes may be looked up.
    5. Aggregate the results from the indexer nodes, rank them by personalizing them to the user, and send the results back to the client.
4. **Caches**: The caches store intermediate results from the indexer nodes intermittently. We may have in-memory caches (such as *Memcached*) or support persistence on-disk caches (*Redis*).
5. **Indexer Nodes**: The indexer nodes store the distributed trie data structure. Our design would have 16 nodes that hold 1 TB of data

(with a redundancy factor of 2). The Indexer Nodes are backed up with a distributed file store. If an indexer node goes down, the cluster manager (not shown in the figure) will restart the node. On a restart, the node will then re-index all the data from the distributed file store.

6. **Distributed File Storage**: Stores the actual data (i.e., 10 Billion strings). This may be built using Amazon's S3.

For the scope of an interview, this is a good enough discussion to have with the interviewer. There are a few topics that need to be covered to complete the scope of the interview. These are as follows:

I. API Interface between the client and the web-server, and the web-server and the indexer-nodes.

II. Solving for challenges such as getting top 10 matches, auto-completion for empty strings, handling failover between nodes, etc.

We will cover a blog on advanced design for *Search Engines* where we will cover (I) and (II).

References

1. Latency Numbers Every Programmer Should Know · GitHub
2. Trie | (Insert and Search)
3. Binary Search Tree
4. Database Sharding. Database sharding is the process of... | by Vivek Kumar Singh | System Design Blog
5. System Design: Challenges In Distributed Systems (Availability)
6. System Design: Fundamentals

Chapter 5 Designing A Scalable Notification Service

Introduction

Let us first start with a definition of a notification service. A "notification service" is a general-purpose service that is used to send notifications to its users. There are several use cases of notification service. The primary use case tends to be setting up an alert based on a specific event in the system. Let us take a look at a few concrete use cases.

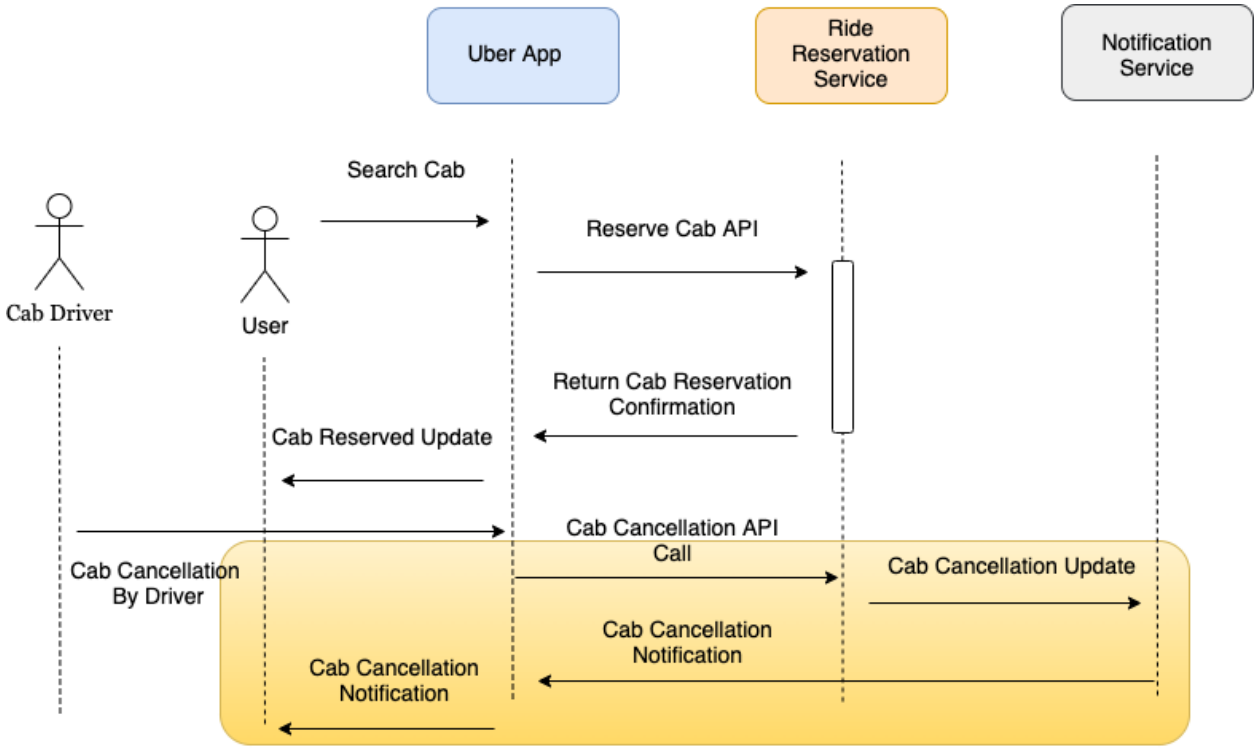
Use Cases

- **E-Commerce**:
    - Notifying the user when an item becomes available.
    - Notifying the user when an item's price increases or decreases significantly.
    - Notifying the user when an item has a discount running on it.
- **Cab-rides**:
    - Notifying the user when a cab ride is booked or canceled.

    - Notifying the user when a credit card transaction goes through or is declined.
- **Software development**:
    - Notifying the user when a service is down. Typically used by APMs (such as AppDynamics, NewRelic).

Problem Statement

For the scope of this blog, let us take the example of an application like Uber, where the user is updated when a cab ride gets booked. Here is the high-level view of the data flow of the application:



A Basic Notification Service

The workflow is as follows:

- The user sends a request to book a cab using his or her application.
- The API then hits the *Ride Reservation Service*. The Service then reserves a cab from the list of available cabs and sends back a confirmation to the app.
- The application relays the confirmation back to the user.

If the cab driver cancels the cab, then the following workflow is triggered.

- A cancellation request is made to the *Ride Reservation Service*.
- The Ride Reservation Service relays the notification to the Notification Service.
- The Notification Service then relays triggers a cancellation to the Uber App.
- The Uber App then relays the notification back to the user.

Functional Requirements

- The "*notification service*" should send an email or a push notification to users if there is a cancellation by the cab driver or the rider.
- The "*notification*" should also contain some metadata, i.e., the reason for cancellation and any additional charges levied.

The "*notification service*" should be agnostic of the client, i.e., it should be able to send the *notification* whether the client is a Web App, Mobile App, or an API end-point.

Non Functional Requirements

- The "*notification service*" should be independently *scalable*, *deployable*, *durable*, *available, secure,* and *maintainable*. Let us look at each in detail:
    - *Scalable*: Should support sending tens of million messages per hour.
    - *Available*: Should survive hardware/network failures. It should not have a single point of failure.
    - *Performant*: Should keep end-to-end latency as low as possible. The message should get delivered within five seconds to the user.
    - *Durable*: Should not lose the message, and the message should be delivered at least once to the user.

- *Maintainable*: It should be easy to debug and look at metrics such as how many notifications were delivered, maximum latency of delivery, the maximum load on the service, etc.
- *Deployable*: Upgrading the notification service should not disrupt any other service.
- *Secure*: Should not send updates of one rider to a rider. The same holds for a driver as well.

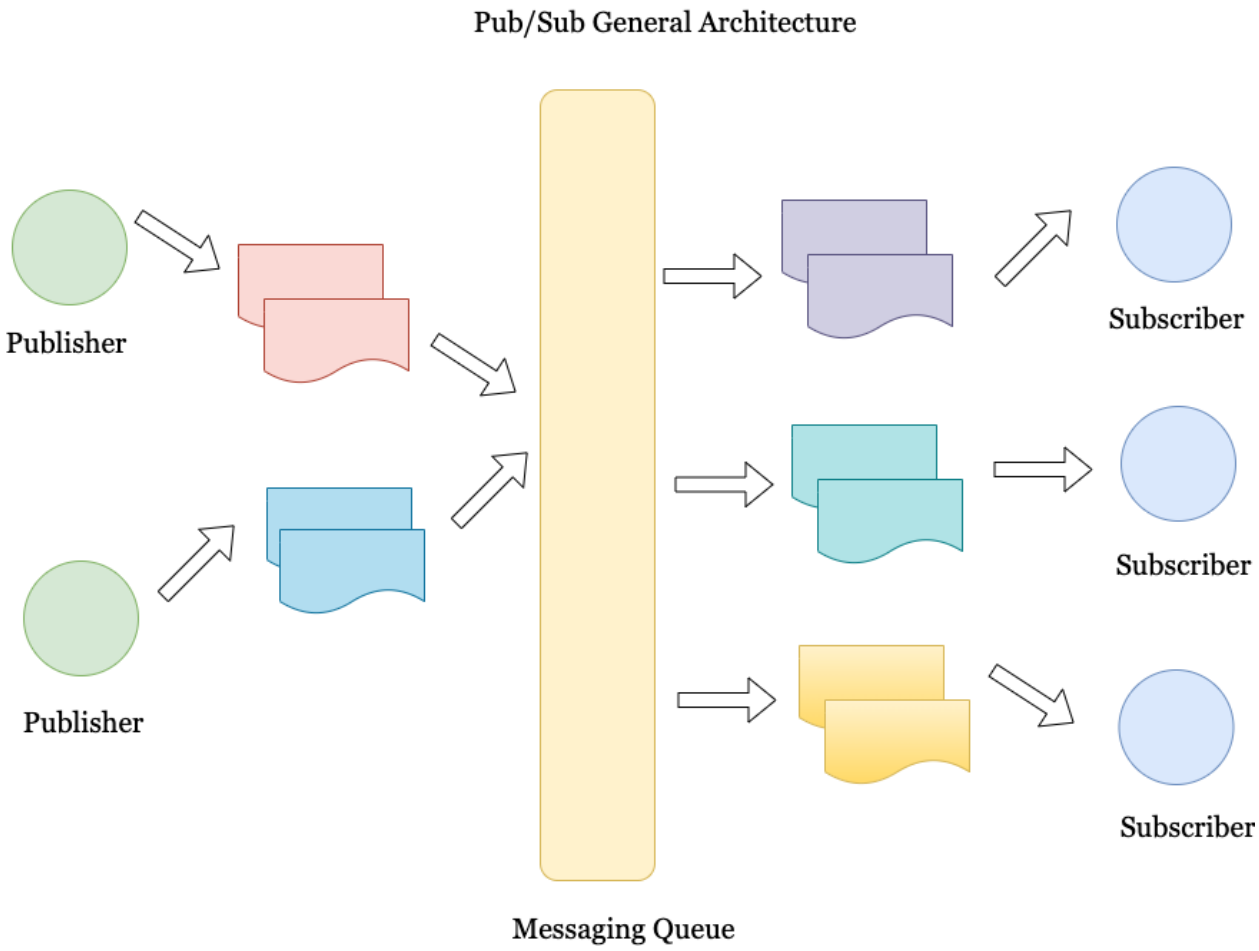
Basic Components

Before we get into the High-Level Architecture design, let us discuss some of the fundamental ideas required to understand the system.

Publisher-Subscriber Model

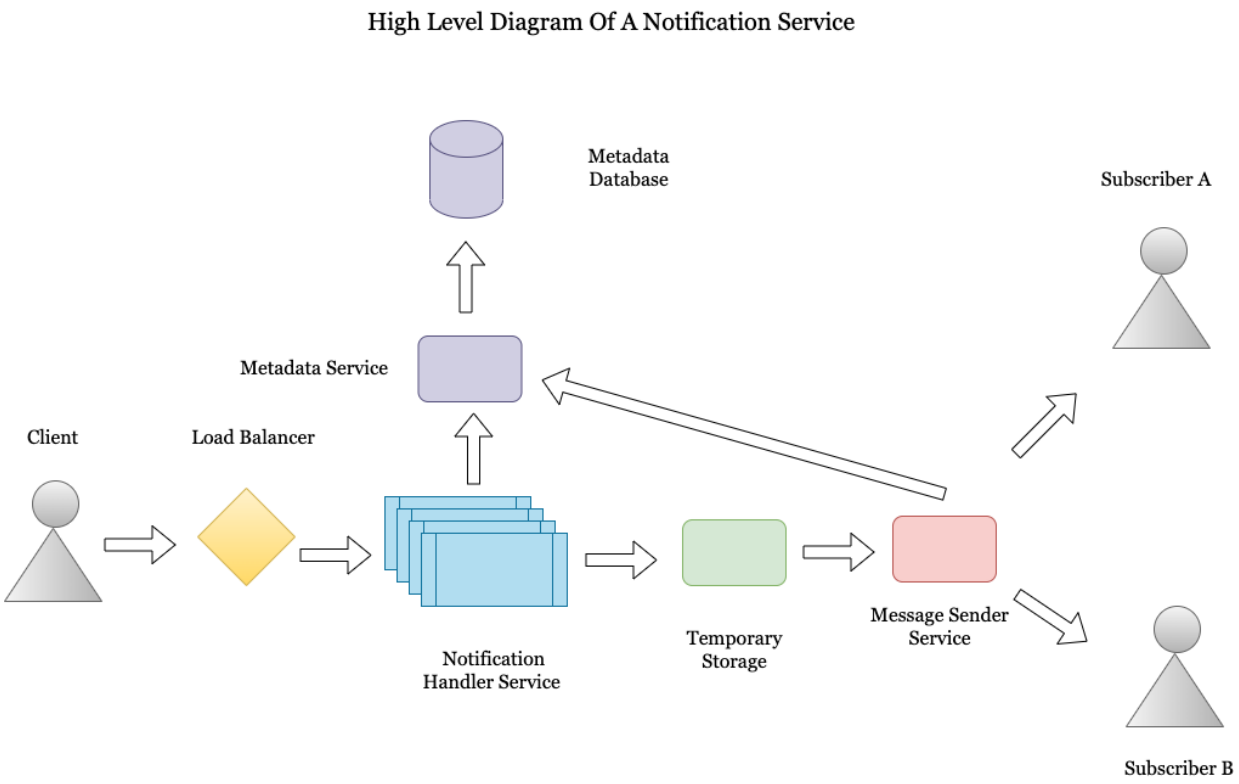
In software design, publish-subscribe is a messaging pattern where senders, called publishers, do not program the messages to be sent directly to specific receivers called subscribers. Instead, they categorize

the messages into groups (topics) into intermediate messaging subsystems (e.g., a messaging queue). The messaging queue is agnostic of the semantics of the messages. Examples of messaging queues are Rabbit MQ, Apache Kafka, etc.



High-Level Architecture

We will now discuss the components of the design in detail.



Load Balancer

A load balancer sits in front of the notification handler service to balance the load on the downstream notification service and handle routing in case of downstream failures.

Notification Handler Service

The Notification Handler Service is a lightweight web service. It is stateless and can be deployed across several data centers. Some of its functions are:

- Request validation
- Authentication and authorization
- SSL Termination
- Caching
- Rate Limiting & Throttling
- Request Dispatching
- Metric Collection
- Encryption of responses

The frontend service host consists of the following sub-components:

1. Reverse Proxy: The reverse proxy handles the incoming request, terminates the HTTPS connection, and forwards them to the Notification Web-Server. Some of the popular reverse proxy servers are NGINX and HAProxy.

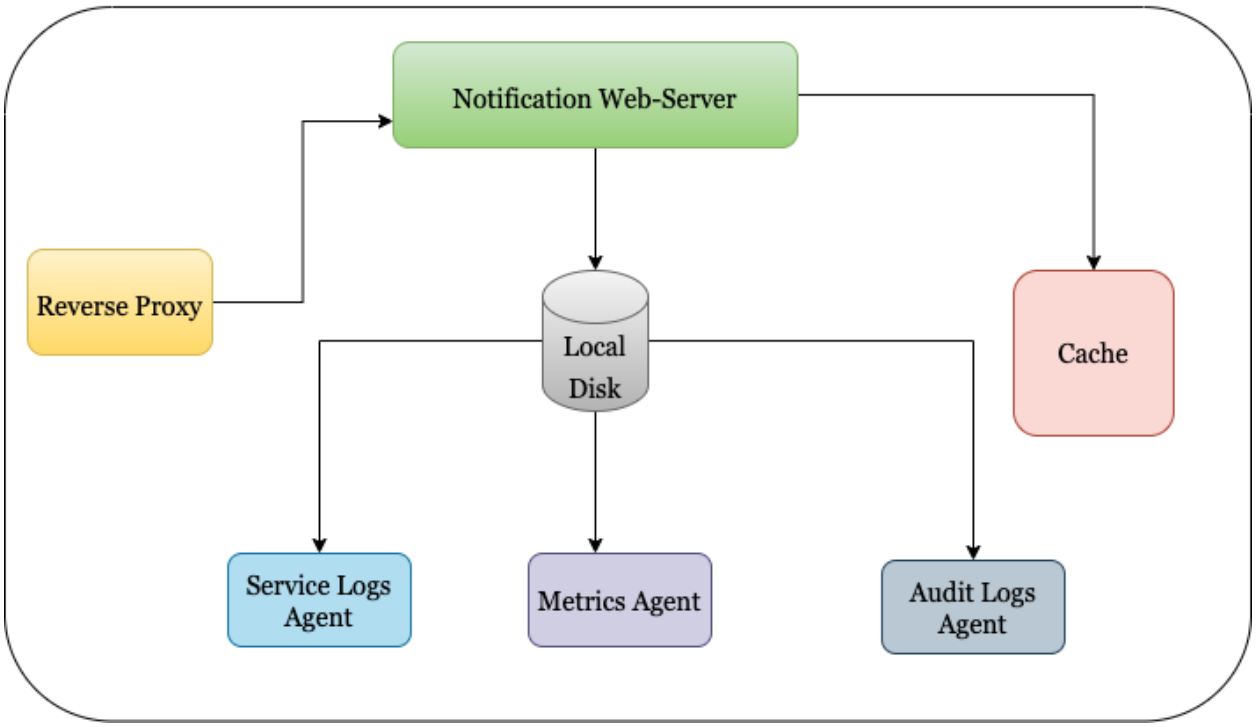
2. Web Service: The Web Service encodes the logic to handle the incoming request, fetch the topic for the message from the Metadata Service and then pass it on the temporary storage. It also performs request validation, auth checks to ensure that the request has the necessary privileges.
3. Local Cache: The Web Service may also keep a local cache so that it does not have to visit the Metadata Service for every call to get the topic of the message. It can be an in-memory cache and built using Guava or Memcached (as a separate process).
4. Service Logs Agent: The webservice also generates many logs. These logs consist of exceptions, warnings, metrics, and audit trails. To capture these logs, usually, we have an agent injected into the service. These agents collect logs and send them to a central location such as Splunk and ELK. The log search platforms make it easy for developers to find issues and debug them.
5. Metrics Agent: A set of metrics are injected into them as well to monitor services. Some of these metrics include the following:

1. The number of notification requests to the web server
2. The throughput of notification requests
3. The number of valid/invalid notification requests
4. Latency distribution of the notification requests.

A metric agent will capture these metrics and send them to an APM (e.g., AppDynamics, and NewRelic).

Audit Logs Agent: Similar to the metrics, logs are also collected to audit the system. These may be collected and stored in a Database using an Audit Log Agent.

Notification Handler Service Implementation



Metadata Service

The Metadata Service stores a mapping between a topic and its corresponding owner. Primarily, it acts as a caching layer between the Notification WebServer and the Database.

System Requirements:

The cache should be able to support high reads, low writes.

Solution:

There are multiple ways to implement it. We can use any scalable cache such as Redis, Memcached, and DynamoDB like the Metadata Service.

Metadata Database

The Metadata Service is backed by a relational database that stores information corresponding to each topic ID. Some of the metadata that has to be stored with each topic is the list of subscribers for each topic, configurations such as rate-limit per topic, SLAs, etc., defined for each topic. The data has a relational structure to it, and therefore, a relational DB would suffice.

Temporary Storage

The most critical piece of the system is a temporary storage layer that sits between the Notification Service and the Subscribers.

System Requirements:

- The temporary storage should support the durable storage of messages for a small period.

- The temporary storage should be scalable as the number of messages to be read and written will be very high.

There are several options to consider. Let us discuss each of the following possibilities in detail.

Databases: We can go with either a SQL or a NoSQL database. System requirements:

- Large scale.
- Do not require ACID semantics or transactional semantics.
- The database must be highly scalable for reads and writes, highly available, and tolerate network partitions.
- Do not require highly complex analytical queries.
- Do not require data to be stored for analysis or data warehousing.

A No-SQL database such as a key-value store or columnar data store is better. Examples of such databases include Cassandra or Dynamo DB.

In-memory caches:

Another option to store the intermediate messages is keeping them in in-memory caches such as Redis and Memcached. Performance of reads and writes is an advantage of in-memory caches. The disadvantage tends to be durability and correctness semantics. Disk-backed caches such as Redis do solve the problem of durability. However, the notification service will have to implement some of the correct features as message delivery guarantees, only once semantics, etc.

Messaging Queues:

The last option we will discuss is messaging queues. Some of the standard messaging queues are Apache Kafka, Amazon's SQS, and RabbitMQ. Typically, a messaging queue provides the following guarantees:

1. Permanent Storage
2. High Availability
3. High Throughput
4. Scalability

Apache Kafka:

Apache Kafka is designed on an abstraction of a distributed commit log. It is a distributed event stream processing platform and can handle trillions of events in a day. It supports the "*Publish-Subscribe*" model. The underlying design is an immutable commit log, from where publishers can write messages and the subscribers can and read messages from. Some good resources to go deeper into are Kafka's original paper and blog about how LinkedIn uses Kafka for 7 trillion messages per day.

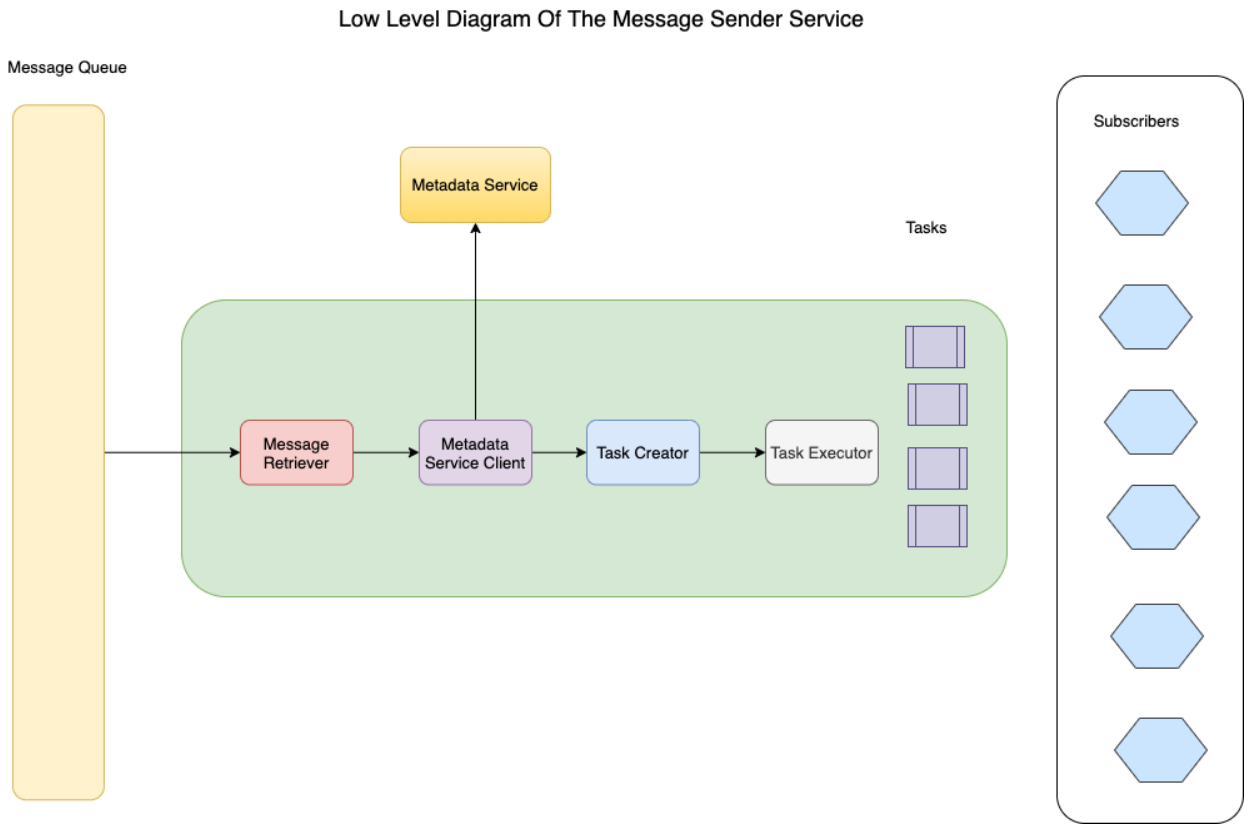
Compared to RabbitMQ, Kafka has a much more performant platform, has a more significant community around it, and is much more widely adopted.

In comparison to Amazon's SQS, Kafka being cloud-agnostic, has a much higher value in being used.

Given that a message queue satisfies most of our requirements and plays well with our pub-sub model, it is good to go with the best message queue platform out there, i.e., Apache Kafka.

Message Sender Service

The primary role of the message sender service is to read messages from the shared message queue and send them to the downstream subscribers. Let us look at the design in more detail.



Message Retriever

The Message Retriever module gets messages from the message queue. Once the message is read from the message queue, it is passed on to the

task creator to dispatch the message downstream to the list of subscribers that consume the message.

Apart from reading the messages, it also can throttle the number of messages being read so the Sender Service may not get overwhelmed by the sender service.

Metadata Service Client

The Metadata Service Client gets the metadata information that is associated with each message. We make a separate call instead of passing the metadata information along with the message itself. The rationale for this design is to reduce the size of the payload that the service has to pass around with the message.

Task Creator + Task Executor

Once the client receives the message and its metadata, the task creator will translate it into a task. The task will contain information about the message, a unique identifier, a list of all the subscribers, the total number of subscribers, timestamp, etc. The task is a wrapper defining the

message. A thread pool such as the task executor is then used to send the message to subscribers. Java provides a pre-defined implementation of a ThreadPool. The push to subscribers may be using an email, SMS, IM (e.g., WhatsApp), API end-point, and a Phone Call.

Non-Functional Requirements

Is the system architecture scalable? ✅

Explanation: Each of the components of the architecture can independently scale. It includes Notification Handler, Metadata Service, Temporary Storage, and Message Sender Service. This architecture will scale horizontally.

Is the system architecture highly available? ✅

Explanation: Each of the services has redundancy built-in and can be replicated across data centers. None of the services expose a single-point-of-failure.

Is the system architecture performant? ✅

Explanation: Each of the services can scale horizontally. Besides, the caches (Metadata Service and Temporary Storage) are built keeping in mind performance. The Metadata Service is built on top of Redis, Memcached (which are performant services), and the Temporary Storage is built using Apache Kafka, which is highly scalable.

Is the system architecture performant? ✅

Explanation: The messages in the message queue are persisted on disk and are durable. Similarly, the Metadata Service built on Redis is durable and backed by a Database, making data loss very low.

Challenges To Consider

We have designed a functional, scalable, available, and performant system. There remain several practical challenges that we will need to consider. We will consider these in a follow-up blog.

1. *Thresholding Spammers*
2. *Duplicate messages*
3. *Message Delivery*
4. *Monitoring (Alerting)*
5. *Message Ordering*
6. *Security*

We will discuss these challenges in a follow-up blog, an Advanced Form Essay On Notification Service Design. Furthermore, we will also discuss the design of a scalable message queue (e.g., Apache Kafka).

References

1. System Design Interview - Notification Service
2. Architecting a Scalable Notification Service | by Ardy Gallego Dedase | The Startup
3. Publish–subscribe pattern
4. What is Apache Kafka?
5. Messaging that just works — RabbitMQ
6. Apache Cassandra | Apache Cassandra Documentation

7. Fast NoSQL Key-Value Database – Amazon DynamoDB – Amazon Web Services
8. How LinkedIn customizes Apache Kafka for 7 trillion messages per day
9. Kafka: a Distributed Messaging System for Log Processing
10. Introduction to Thread Pools in Java
11. New Relic | Monitor, Debug, and Improve Your Entire Stack | New Relic
12. AppDynamics: The world's # 1 APM solution
13. Redis

Chapter 6 Lessons From Netflix's Notification Service Design

Photo by Thibault Penin on Unsplash

Netflix, the widely popular video-streaming platform, released its platform for managing and scaling its notification service. This chapter will discuss the architecture of the system's design and the lessons that they learned having built and scaled the system. They named their system ***Zuul.*** Here are a couple of useful links:

- Link to the YouTube talk
- Link to the Github Repository

Alright. Let's get started, then!

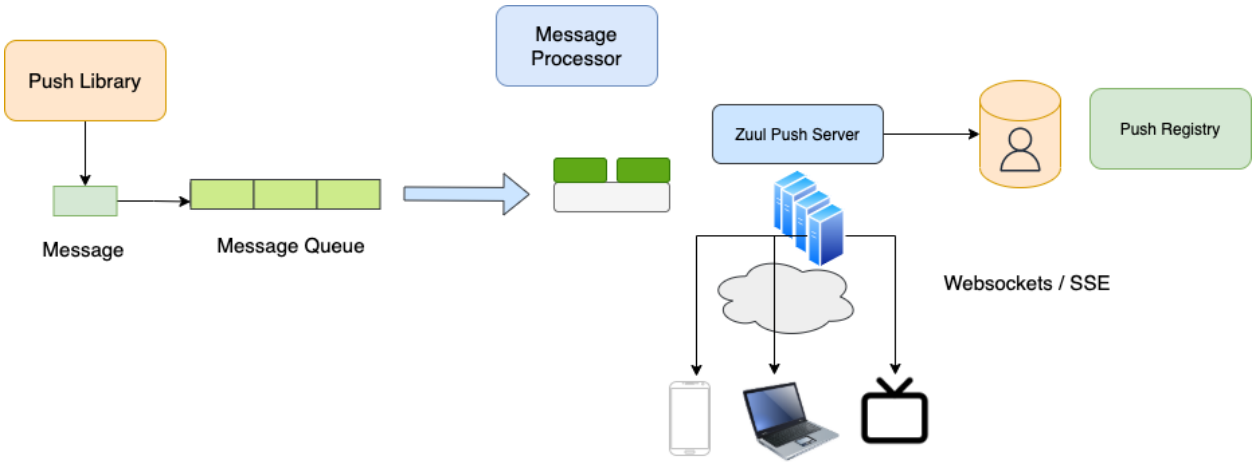
High-Level Architecture

The basic components of the Zuul Push Architecture are as follows:

1. Zuul Push Servers
2. Push Registry
3. Message Processor

4. Push Message Queue
5. Push Library
6. WebSockets/SSE

Zuul is not a single service but a complete infrastructural platform made of multiple components. *Zuul Push Servers* sit on the network edge and accept client connections.



The clients connect to servers using either *WebSockets or SSE*. The Websocket connections are persistent and async in nature. Websockets enable the clients to push updates from the edge to the Zuul Servers. Each of the clients connects to a Zuul server. A *Push Registry* is built to keep track of which client is connected to which server. Senders use the

Push Library to send messages to clients. The call that they use is a single, asynchronous send message call. The message is sent to a push message queue. A *Message Processor* handles the messages, finds the server to which a message must be delivered, and sends the message to the Push Server.

What are WebSockets?

WebSockets, developed by Michael Carter and Ian Hickson, essentially is a thin transport layer built on top of a device's TCP/IP stack. They provide a TCP communication layer to web applications that are as close to raw as possible, without the developer dealing with raw sockets. They enable a two-way interactive communication session between a client (on the user's browser, mobile, or Television, etc.) and a server. With this API, a client can send messages to a server and receive event-driven responses without polling the server for a reply. This is a good article on the differences between WebSockets and the HTTP Protocol.

Use Case Of WebSockets: The emergence of WebSockets was a need to support duplex communication between the client and the server. Let's

take the example of Netflix. You are watching a movie, and the movie starts to crash in between. With a traditional HTTP connection, the server will need to poll the client to check for regular updates. It makes the entire process resource-intensive. The client establishes a connection with the server, creates a connection, and then closes the connection. The process, if done repeatedly for thousands of concurrent users, can consume a large set of resources.

Component Diagram In Detail

This section discusses the design and implementation of the different components with the Zuul Notification System.

Zuul Cluster

The Zuul Cluster can handle an aggregate of 10M concurrent connections at peak. It is based on the Zuul API Gateway. The Zuul API Gateway can support more than 1 Million Requests Per Second. "**The Cloud Gateway team at Netflix runs and operates more than 80 clusters of Zuul 2, sending traffic to about 100 (and growing) backend**

service clusters which amount to more than 1 million requests per second. " The underlying design is non-blocking and supports asynchronous connections. This blog discusses the design of the Zuul API Gateway in detail. Before we go into the detailed design, let's discuss why the system designers preferred an asynchronous model.

Disadvantages With A Synchronous Approach

In the traditional synchronous model, the server would allocate a different thread per connection. This design would quickly exhaust all the resources (CPU and Memory) and wouldn't be able to scale to meet even 10,000 concurrent connections. In this thread, each server maintains its thread stack that consumes memory. Besides, the 10,000 different threads that run will consume most of the CPU, albeit idle, and waste a lot of the resources.

Zuul 2 uses Netty as the framework for the development of its client-server applications. It is an open-source, asynchronous event-driven network application framework for the rapid development of maintainable high-performance protocol servers & clients. Netty is

used by Cassandra and Hadoop platforms and has been well tested in the wild.

Push Registry

The push registry is a data store that stores mapping from each client to the server to which it is connected. Any specific data store can be plugged into it. The characteristics of the required data store are:

- Low read latency; the write latency can be high
- Auto-record expiry - prevents phantom registration records
    - Each connection in the Zuul server creates an entry in the registry when a new client connects
    - If the connection drops and fails to clean up the registry, we end up with what are called "*phantom registries.*"
    - One of the ways to solve this problem is to have a system for automatic expiration in place. This can be done by associating a TTL with each entry in the Push Registry.
- The data store should support sharding for improvement of performance and replication for fault-tolerance.

- Redis, Cassandra, Amazon's Dynamo DB all satisfy these requirements.
- The Netflix team eventually went on to build a system called *Dynomite* over Redis. Dynomite supports auto-sharding, a quorum for reads and writes, and cross-region replication for fault tolerance.

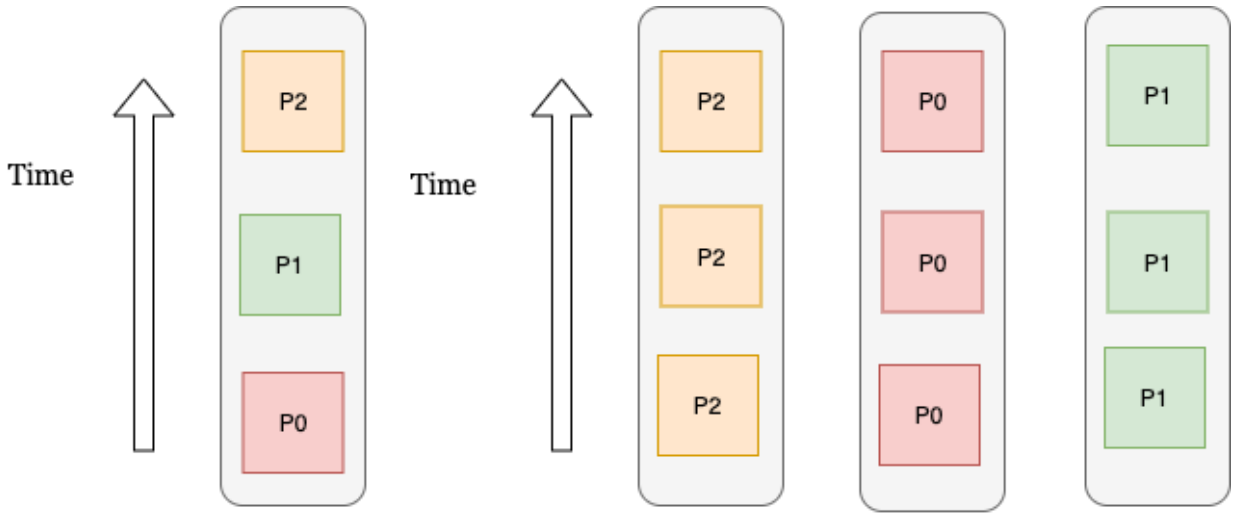
Message Processing

The Netflix team decided to go with Apache's Kafka platform for message queuing. Refer to our discussion on the tradeoffs of using Apache Kafka over other services such as RabbitMQ and Amazon's SQS in a previous blog. The message senders use the message bus in two different ways.

Fire & Forget: Some of the senders adopt the "*fire and forget*" approach. In this approach, the senders send the message on the queue and do not check back on whether the message has been delivered to the client. This could use cases where a weak consistency model may work well instead of gaining some performance. An example could be a request for collecting likes on a video. Eventual consistency will work.

Subscription Based: Some of the senders adopt a stricter model, where they subscribe to updates from the Push Registry to ensure that the message is delivered and received by the client. The sender can then do a re-try in case the client does not receive the message. This is a strict consistency model. Examples of use-cases can be sending updates (such as software upgrades, pushing new content to the clients, etc.). The status is maintained in the "*Status Queue*." The "*message queue*" is replicated across regions to ensure that the queue remains fault-tolerant.

To prevent priority inversion (video, blog), a separate queue is maintained for each priority. In the image below, we can see how having individual threads for each priority removes the problem of priority inversion.



Example of Priority Inversion

Separate Queues For Each Priority

To scale up the message processor service, multiple instances of the service are run in parallel.

Operational Lessons

The most important lessons for any system are learned from running the system in an actual production setting. This tends to be the most exciting and relevant part of the process of a system's design.

Persistent Connections Make Zuul Push Server Stateful

In Zuul, the clients make persistent connections with the Zuul servers. These tend to be long-lived and stable connections. While they are great for a system's overall efficiency, they tend to be terrible for operations. Let's take a few examples:

Rollbacks are painful: Let's take the example of a bug being pushed in production. Let's imagine that the fix requires a new software push and resetting connections from the clients. It could be a change in the TCP stack and old connections may still be running with the bug. Now, in order to achieve this, all the old connections will need to be reset. In order to achieve that a manual process must be triggered to reset connections from the servers. In addition, all the push registries need to be updated and can cause bugs in the system

Migrations are painful: Migration of servers is very, very tricky. In addition to operational issues (mentioned above), it can lead to the problem of the *thundering herd problem*. Let's understand this better. Let's say there are 10,000 active connections on server 1. Now, we need to migrate these to server 2 because we want to reset server 1. If we do it in

one go, server 2 will get 10,000 requests over a short period of time, the server will get overwhelmed and the clients will see a very sluggish performance.

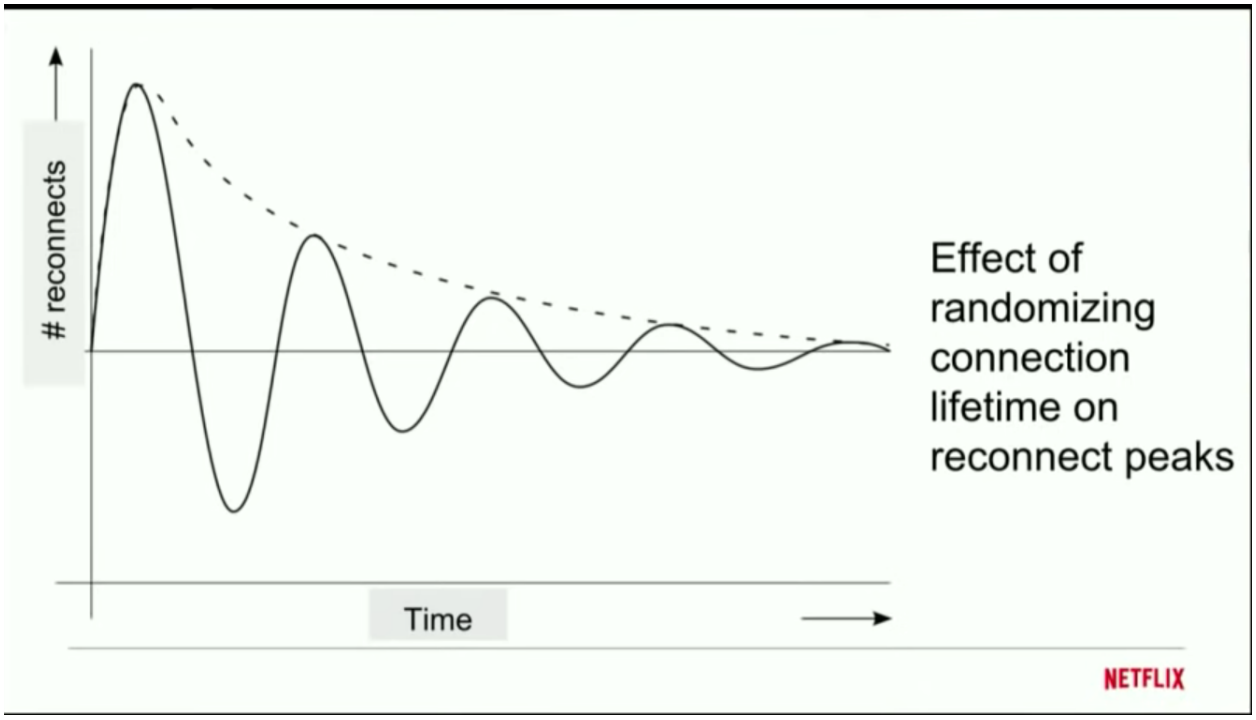
I think we now get a sense of how the performance of the system can degrade :)

How did they solve the problem?

Here comes the exciting part. There are two interesting engineering tricks that the team used. These are as follows:

Terminate connections periodically: By terminating connections periodically, they ensure that a client doesn't stick to one single server. It is pretty much at the root of the *thundering herd* problem. Now, when the client disconnects and reconnects, it will most probably connect to a new server. If the server is buggy or not very performant, the load balancer can route requests away from the server and balance the load incrementally and not in a single go.

Randomize each connection's lifetime: In addition to terminating connections periodically, they also ensured that each client's connections have a different lifetime. The impact was dampening the number of reconnects per unit time. Instead of each of the 10,000 connections hitting and reconnecting through the load-balancer simultaneously, the connections would dampen out and reconnect at different timestamps.



Lessons: To avoid the problem of *thundering herds*:

- Terminate connections periodically.
- Randomize the lifetime of each connection.

Optimizing Push Server

Another observation that the team had was that most connections to the servers remained idle over the lifetime of the connections. The seemingly optimal approach was to keep a single beefy server with all the connections crammed to it. Why the approach? They believed that fewer servers would make the operational cost low as the IT team would have to maintain fewer servers. However, the design fails in case of server failures. A failure of a single beefy server would lead to many reconnects. It gives rise to a thundering herd in case of a server failure and can cause a stampede in your server. Solution: *Goldilocks strategy*. "*The Goldilocks principle is named by analogy to the children's story "The Three Bears", in which a young girl named Goldilocks tastes three different bowls of porridge and finds she prefers porridge that is neither too hot nor too cold but has just the right temperature*." The team decided to go with a strategy where the server size was neither too large, nor too small. They ended up using Amazon's *m4 large*.

Lesson: Keeping costs the same, a high number of small servers is better than a low number of big servers.

Auto-Scale Server

Typically, systems with built-in elasticity scale well. Elasticity is the ability of a system to scale on resources dynamically and adapt to the incoming traffic load. In the case of Netflix's system, they needed to take a decision on which metrics to auto-scale the system on? Should they go with RPS (i.e. requests per second), or CPU usage? Both the metrics, in this case, seemed not to be useful. Given that the threads remain idle for most of the time, CPU is not a great metric. Similarly, given that the connections remain persistent, requests/second isn't useful as well. They went with the number of open connections as the metric to optimize bind elasticity.

References

1. The WebSocket API (WebSockets) - Web APIs | MDN
2. Scaling Push Messaging for Millions of Devices @Netflix

3. Zuul Github Repository

Chapter 7 Interview Prep: Designing A Distributed Cache

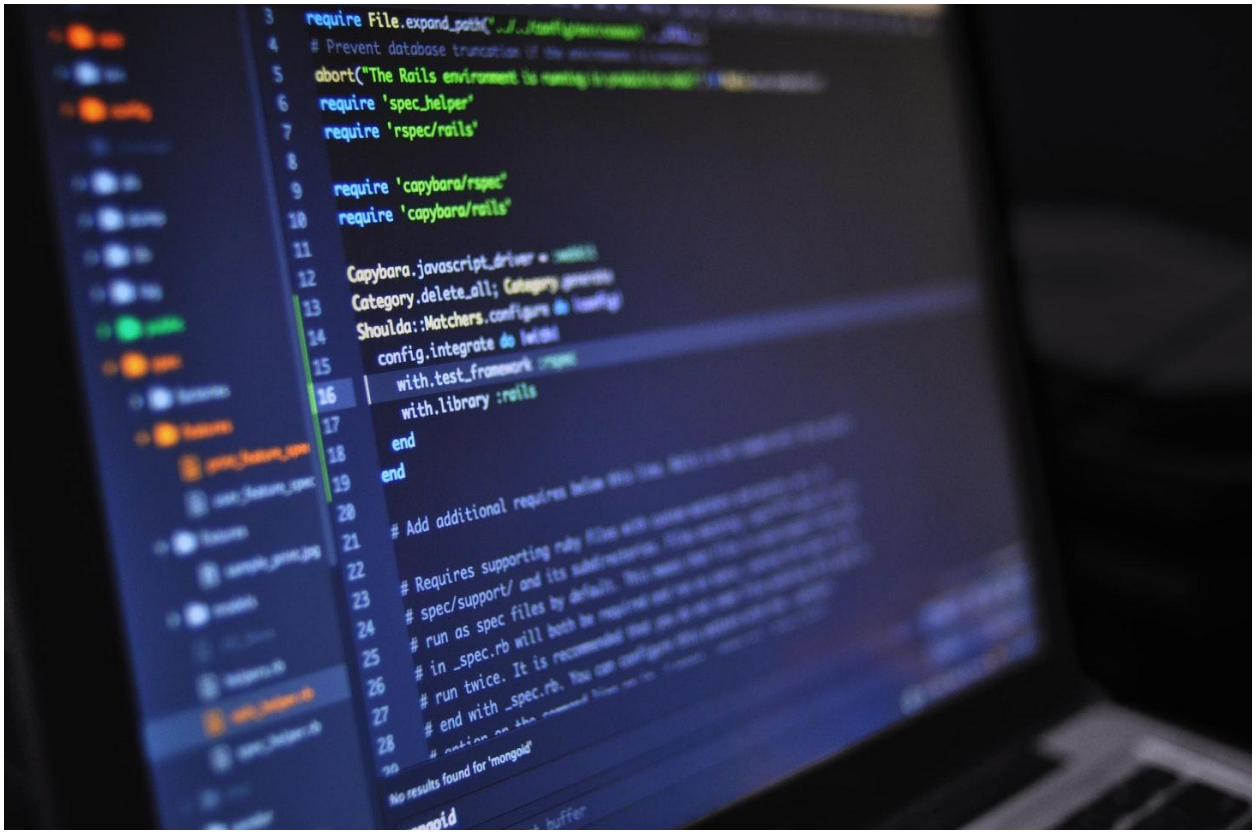


Photo by **luis gomes** from **Pexels**

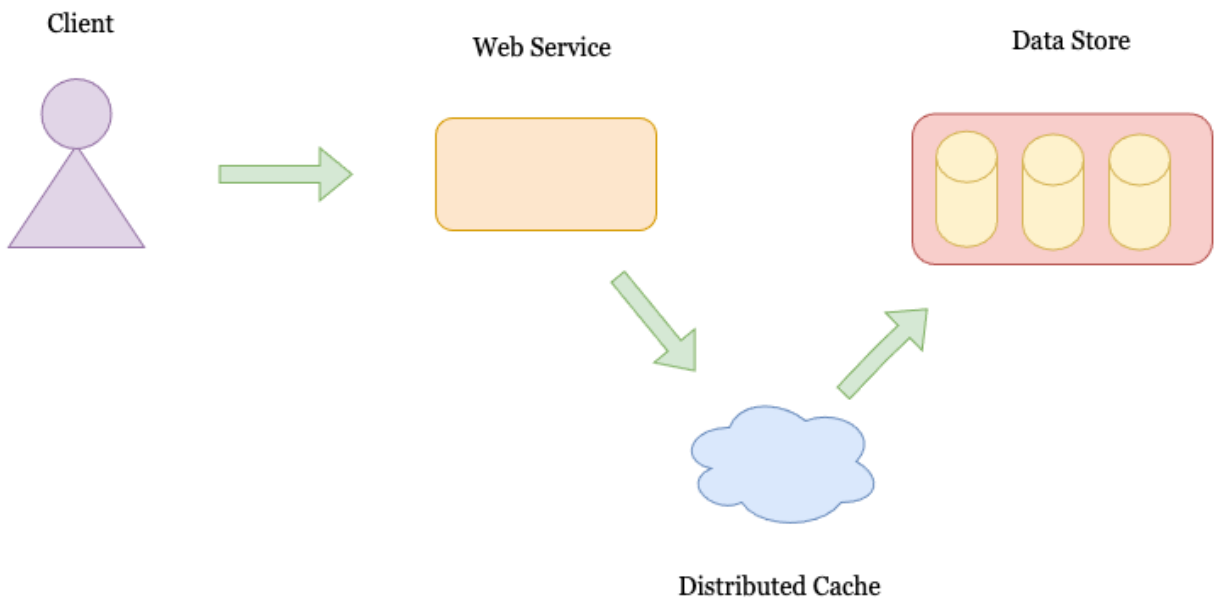
A Generic WebService Setup

Generally, a webservice has the following setup. There is a client that connects to a webservice. The client may be a web application, mobile application, or API client. The client then connects to a web service. The web service fetches the required data from an underlying data store. The

data store may be a relational store (MySQL, Postgres SQL), a NoSQL store (Cassandra, MongoDB), or a file store like Amazon's S3.

As the system scales, the performance of the data store (both in terms of latency and throughput) starts to degrade. The degradation can be because of a high number of requests, high parallelism, expensive queries, etc. To alleviate the problem, web services employ a cache to keep a pre-recorded set of queries. In general, applications have a Zipfian distribution pattern. 80% of the requests are served by 20% repeated queries. Therefore, Caches are a powerful element when scaling an application.

This blog discusses the design and implementation of caches in further detail.



Web Application With A Distributed Cache

Caching: Definition, Types & Use Cases

Cache: Definition

A cache is a short-term memory used to store data items from databases, servers, or other secondary storage layers. It is typically faster than the source and keeps data for a smaller period. Normally, a cache helps speed up the application, reduce the load on the backend servers, and help scale the system. A cache may sit on a single node or multiple

nodes. For the course of this chapter, we will focus on the design and implementation of a distributed cache and discuss the advantages and challenges of maintaining one.

Distributed Caching

A distributed cache is a cache system that sits on several nodes (a) within a cluster, (b) across multiple nodes in different clusters, (c) across several data centers located in the world. Why do we not use a simple one-node cache? The answer to this is simple. As a system scales up, they need to store hundreds of GB of data in the cache. It becomes impossible to store that much amount of data in a single node's memory. Therefore, the requirement of building a distributed cache becomes evident.

Why do we need distributed caching?

There are several advantages that a distributed caching system provides. Here are the requirements for a distributed caching system.

Functional Requirements

- Put (key, value): The first requirement of a cache should support a put operation on a key-value pair. The value is mapped to the key and can be retrieved using the key uniquely.
- Get (key) - The second requirement of a cache is that it should support a get operation. The get operation takes a key as the input parameter and emits out the value. In case the key is not present in the cache, the get operation returns null.

Furthermore, most caches also treat a Get, and Put as legitimate access to the item and maintain metrics around the access patterns. We will discuss this in detail in this chapter.

Non-Functional Requirements

- ✅ **Scalability**: The cache cluster should scale-out with an increasing number of requests and data. The constraint remains that the performance of the application shouldn't degrade as the application scales 10x in the data it stores or the number of users using the application.
- ✅ **Highly Performant**: The gets and puts should be fast. This is the primary requirement of a cache cluster. The performance in

terms of latency of look-ups shouldn't degrade as the number of requests and data scales on the cluster.

- ✅ **Highly Available**: The cache cluster should service hardware and network failures. In case a node within the cluster fails, the cache cluster should remain alive and the requests should be served from other nodes. It should be noted that a lot of applications have a requirement of *eventual consistency*. Take an example of caching the likes on Facebook Photos. For a user, it may be fine for the application to show 100 likes instead of 110 likes. For such applications being available is a requirement from the cache clusters.

Use-Cases

Distributed caches have several use-cases. This section discusses some use-cases of a distributed cache.

Session Store

A cache server (such as *Memcached*, *Redis*) can be used to store information related to a user's session. Webservices use the session information to authenticate access to each API by a user.

Database Caching

Databases store 100 GBs to terabytes of data. Access to data is expensive and induces a lot of load on the server that is running the database. To alleviate this load, caches are deployed as the first layer of access to a database to ensure that the data stores can scale up.

API Caching

Typical web servers can handle 10-100k QPS depending upon the complexity of the API being triggered. However, as the APIs become complex and the number of users increases, it gets hard to scale up webservers. Caches help alleviate this by caching responses to the most common and recent API accesses.

Local Cache

We will start with a local cache and then move to the design of a distributed cache.

1. **Data Structure**

    1. Hash Table: A hash table is used to locate the position of the value within a doubly linked list.
    2. Queue: A queue is used to maintain a list of all the items in the cache. It supports constant time for add, update, and delete operations. We use a doubly linked list and implement an LRU cache implementation.

2. **Algorithm**
    1. **Retrieval Algorithm**
        1. Step-I: Check if the item is in the cache.
        2. Step-II: If the item is not in the cache, return null.
        3. Step-III: If the item is in the cache, return the item and move the item to the head of the list.

           Note:
            1. The item at the front of the list is the most recently used.
            2. The item at the tail of the list is the least recently used.
    2. **Update Algorithm**

1. Step-I: Check if the item is in the cache.
2. Step-II: If the item is in the cache, update the value of the key and bring it to the front of the queue.
3. Step-III: If the item is not in the cache, check if the hash table is at its capacity.
    1. If the hash table is not at its capacity, add the item to the front of the queue.
    2. If the hash table is at its capacity, evict one item (*based on the eviction policy*) and then insert the item to the front of the cache. The item is deleted from the tail of the list.

3. **Eviction Policy**
    1. **Least Recently Used**: LRU is the most commonly used policy for cache eviction. Many applications have temporal locality while accessing items. Items that have been recently used have a higher probability of being accessed.

2. **Least Frequently Used**: LFU is a policy in which the frequency of access is maintained with each item. The underlying hypothesis is that items accessed more often have a higher probability of being accessed than those that are rarely accessed. Furthermore, the cost of a cache miss on a rarely used item is much lower.
3. **FIFO** (**First In First Out**): FIFO is the most straightforward policy that says that items that came into the cache earlier should be evicted earlier. The items that are old now (came in earlier) will have a lower probability of being accessed.

Elements of a Cache System

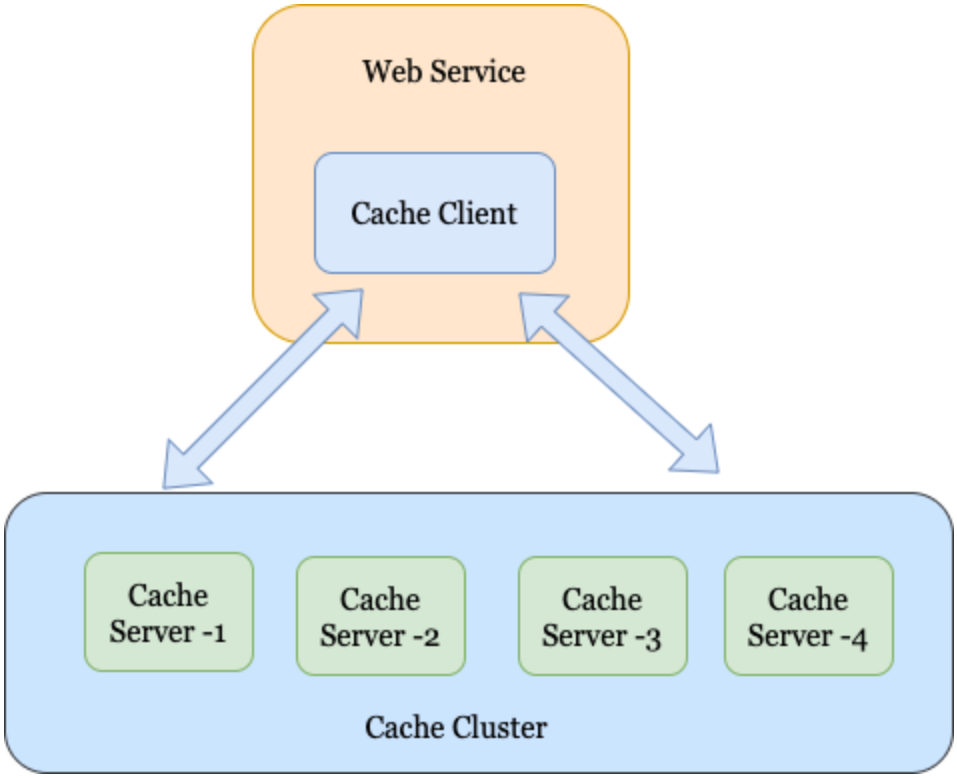
This section discusses some of the basic elements that make up a cache system They are as follows:

Cache Client

A cache client is a library that resides on the client-side. Typically, it knows about all the cache servers. It implements a search algorithm to identify which server should the request be routed to. The algorithm may be a simple hash-based search or may be somewhat more sophisticated, such as a binary search for implementing *Consistent Hashing*. We discuss Consistent Hashing subsequently in this chapter. It uses TCP or UDP protocol to talk to servers. If the server is not available the client assumes that it was a cache miss.

Cache Server

A Cache Server stores the data that the client requires from the cache. Typical caching servers are single nodes or distributed in nature. Some of the common cache servers are *Varnish*, *Memcached*, *Redis*, *Amazon's DAX for Dynamo DB,* and *ElasticCache*. The cache server is also connected to the underlying data store to retrieve the data in case the data is not present in the cache.



Elements Of A Cache Cluster

Designing A Distributed Cache

Co-Located vs. Remotely Hosted:

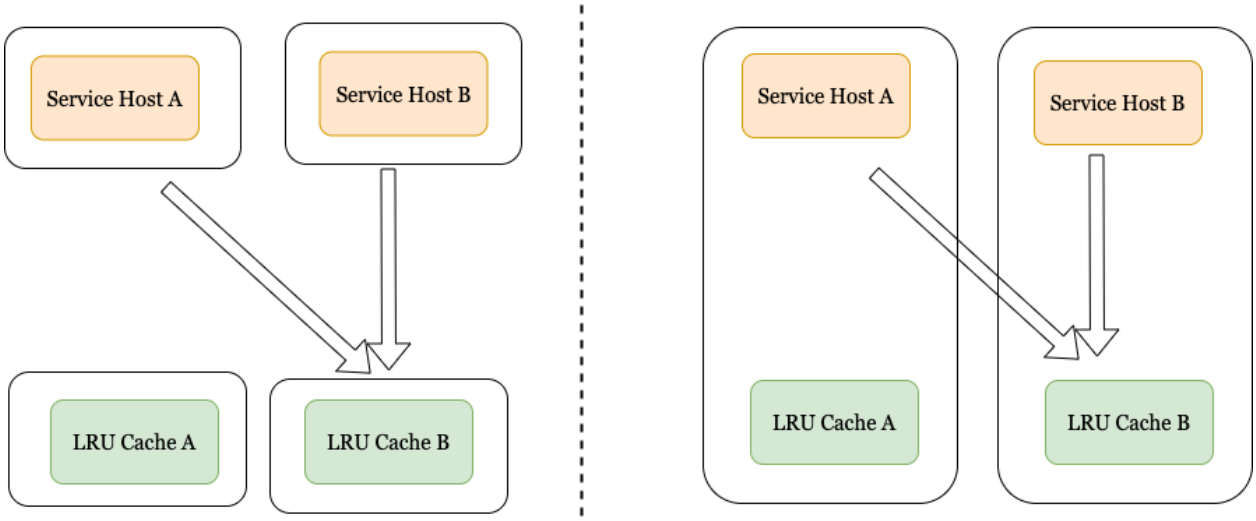
The cache server may be located on the same host as the WebServer or co-located with the Webserver. The tradeoffs for the two choices are as follows:

Advantages of co-located cache:

1. The co-located cache will scale as the web-service scales
2. Extra hardware and operational cost are low.
3. Co-location can help reduce lookup latency

Advantages of remote caches:

1. Isolation of resources between the web server and the cache. So, even if the machine on which the webserver is deployed goes down, the cache server still remains up.
2. Flexibility in choosing different hardware for the webserver and the cache server.



Hashing Algorithms

To deduce which bucket the item should be stored in the cache client needs to rely on a search algorithm. In this section, we will skim through two commonly used hashing techniques.

Mod-based hash: In this technique, the client maps the item using a simple mod-based function. An example of this could be as follows:

Server-id = f(item's key) mod N

where *the function f* translates the item's key into an *integral space,* and *N* is the number of servers in the system.

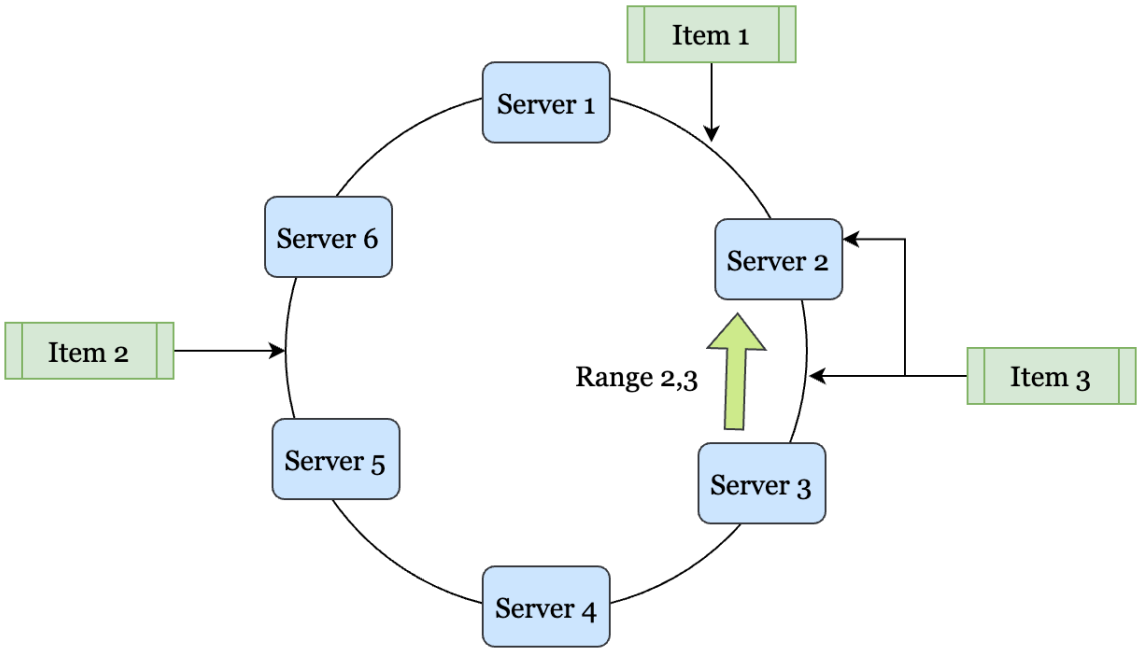
Tradeoffs

1. Advantages: It is a function that is easy to implement and understand

2. Disadvantages: The technique is not resilient to server failures. If a server goes down, all the keys will have to be re-hashed. It makes the overall approach sub-optimal.

Consistent Hashing

Consistent Hashing is a technique wherein the items are distributed amongst servers using a family of hash functions. The image below simplifies the hashing scheme. Each of the servers is hashed using a key. This can be using their IP address in a value space (which is circular). The items are hashed in the same space. All the values between two servers (say Server-1 and Server-2) go to one of the servers. For argument's sake, let us assume that these values go to Server-1. Extending the design,, all the values that lie between Server-2 and Server-3 go to Server-2 and so on.



Schematic Diagram Of Consistent Hashing

If Server-2 dies, all the keys that ownsServer-2 owns are then copied over and owned by Server-1. This way, all the other keys remain stable, and only two servers are distributed. The design is much more stable than using a Mod Function for the distribution of the keys. *Consistent Hashing* is therefore used much more heavily as compared to other hashing schemes.

Achieving High Performance

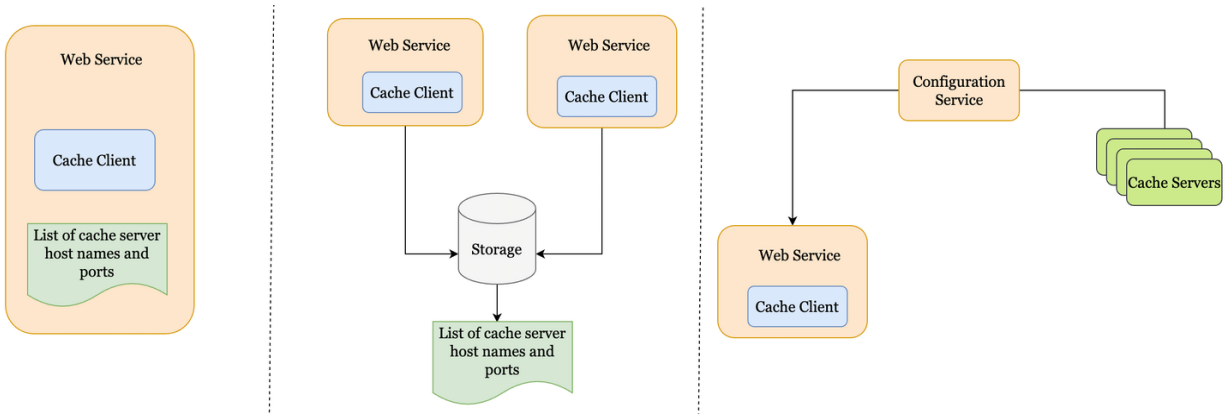
- The LRU algorithm can fetch an item in O(1) time-complexity and is fast. ✅
- Caches keep hot-keys in memory (Memcached, Redis) and lookups are fast. ✅
- Consistent Hashing can find a node in (log N) time through a binary search. ✅
- Cache client accesses data from the server using TCP, UDP. These protocols are fast. ✅

Achieving High Availability

What happens in the case when one of the cache servers goes down? The list of cache server hostnames and ports can be maintained in one of the following ways:

- Single file on each cache client.
- Single file on a shared storage server (E.g., in S3).

- Accessible through a central configuration service such as ZooKeeper.



Different Setups For Cache Client To Access List Of Servers

Trade-offs:

1. The disadvantage of maintaining the list in a single file is that the file needs to be manually updated and the server restarted in case of a server failure.
2. In order to avoid, web server restarts we can maintain the list of cache servers in a separate location within shared storage such as S3. The file needs to be updated manually, still.
3. The optimal approach is to maintain the servers in a configuration service such as Zookeeper and have the client use

the configuration service to update the list of cache servers it can connect to for getting the cache data.

Achieving High Scalability + Availability

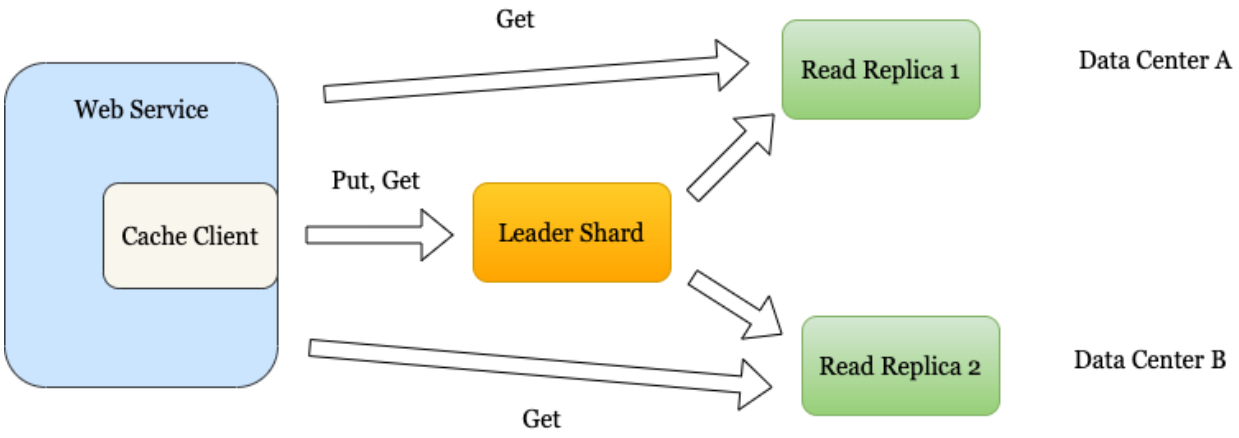
To ensure that the system scales and remains highly available, cache clusters take the approach of replication through data sharding. Sharding ensures that the data space is divided into many subspaces. Each subspace is then distributed to a different node in the cluster.

We can go with a master-slave replication model. Each shard has a leader and reads replicas. The writes go to the leader shard. The reads can go to either the leader or the read replica.

To deal with "*hot shards*", more read replicas can be added to the cache cluster.

In case the leader shard goes down, a leader election process triggers. This can be done through a configuration service such as Zookeeper

which assigns one of the replicas as the leader. The cache cluster remains available throughout the process.



Cache Cluster With Replication

References

1. Distributed Cache Design By RAJESH KUMAR
2. System Design Interview - Distributed Cache
3. LRU Cache Implementation - Geeks For Geeks

Top K Problem (aka Heavy Hitters)

In this chapter, I will discuss the problem of finding the top k hits within a given period. To relate to the problem better, let us imagine that you are building a Google-like search engine, and you need to be able to find the top-k queries within the last one day. [1]

Understanding The Real World UseCase

Figure 1 depicts some of the use cases in the real-world that require the system to solve the problem of finding the top-K heavy hitters. Let us say that you go and login to your Instagram account. The homepage shows you the top 100 trending pictures or accounts on Instagram. The problem is solved in the background using a similar system (that we will end up designing as a part of this chapter). Other examples include the most retweeted tweets in the last twenty-four hours, the most played song on Spotify or the most viewed video on YouTube.

[1] I would call out that I have used this YouTube video for writing most of this blogpost. , Iliked how The System Design Interview channel is built - although it has been dormant for some time now.

Understanding Scale

The most important principle of solving a system design problem is to understand the constraints of the system. The solution one can design depends heavily on the scale of the inputs, requirements of the outputs, and the resources available to build the system. For the scope of this problem we should assume the following:

- The system can contain billions of items;
- Each item may have a size of 10 MBs;
- The output is required within tens of seconds;
- The system can deploy thousands of compute nodes;[2] and
- The system can have infinite storage space.

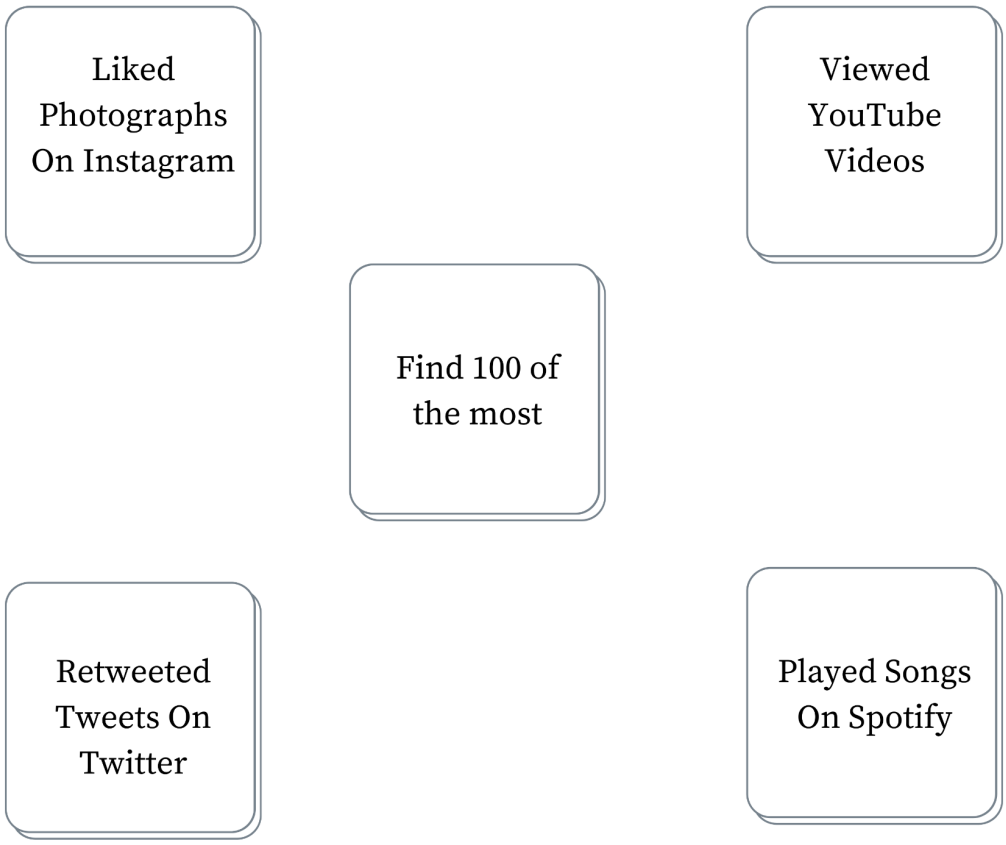
Can We Use MapReduce?

It may be apparent that this problem needs to be solved in real-time. You may end up discussing this with the interview to get a sense of whether they want to solve this in real-time or not. If the requirement is to get the search results within a few seconds then it requires a real time solution. Solutions involving MapReduce may be out of the window as

[2] Each compute node may have 32 GB of memory, 8 VCPUs, and GiB ethernet.

they tend to batch updates and require a longer timeframe. MapReduce persists intermediate data on disks and therefore may not be able to return results back within a small bounded time-interval and thus may not qualify as real-time.

Problem Statement



Functional Requirements

We want our system to return the list of K most frequent items over a predefined time interval.

- topk (K, startTime, endTime)

Non-Functional Requirements

As discussed in our previous chapters, the system should adhere to specific non-functional requirements. Some of these requirements are as follows:

- Scalable - The system should scale with an increasing amount of data ingested.
- Available - The system should survive hardware and software failures, i.e., no single point of failure.
- Performant - The system should be able to return the top 100 list of items within hundreds of milliseconds.
- Accurate - The system may return an approximate solution. With data sampling we can calculate an approximate list. This may be sufficient for certain use-cases such as approximate analysis. However, for the sake for this discussion, let us assume that we

would need to design a system that can return the exact topK results.

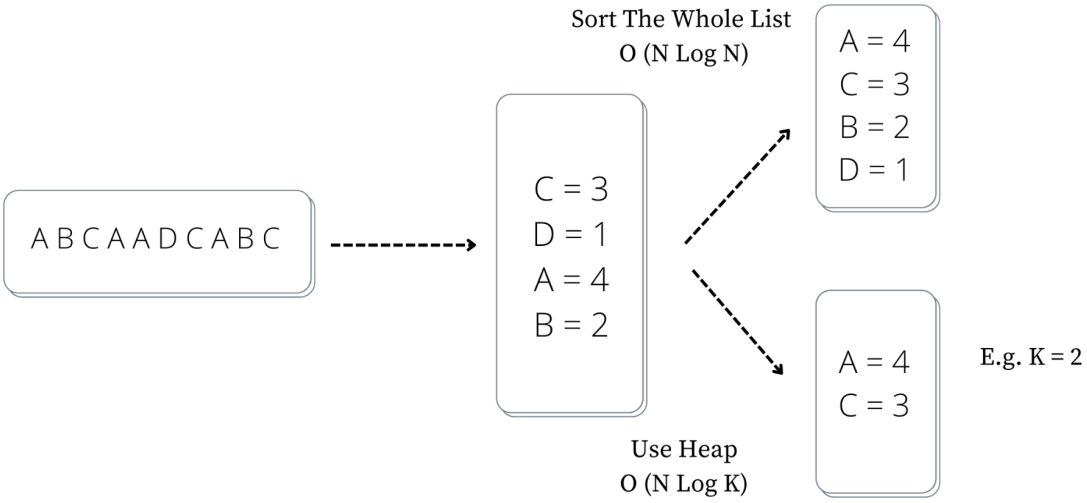
Exploring The Solution Space

Let us first start with the basics using a first principles approach and then work towards the more complex solutions. In general, it is good to start with simple design and then work upwards towards more complex designs. Why does this matter? It helps the interviewer understand your thought process, and shows to them that you are a very systematic thinker. I adopt this approach whenever I am interviewing or discussing a problem on System Design.

Solution #I: Hash table, single host

For the sake of this solution, let us make the following assumptions:

Hash Table, Single Host



Assumption #1:

Let us also assume that all the data can fit on a single server memory.

Assumption #2:

Let us assume that a collection of video events where each of the letters in the figure below represents a video. So, the letter “A” represents Video 1, “B” represents Video B, and so on. We can imagine this as a video streaming service such as YouTube or Netflix.

We can sort the list or use a heap. The heap approach is faster. The time-complexity of sorting is O(N Log N) and that with the heap is O(N Log K), where N is the total number of unique videos and K represents the total number of videos to be returned.

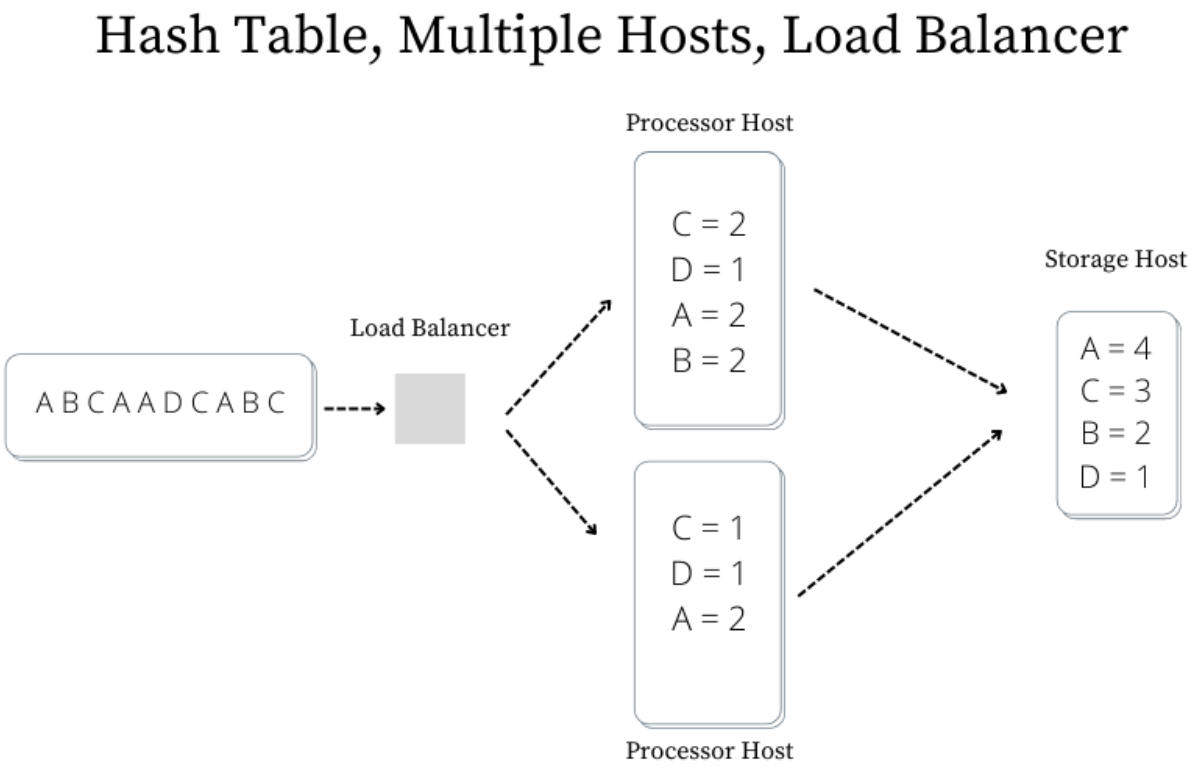
Drawbacks:

The solution is not scalable. If the rate at which the elements hit the single host increases then the host may quickly become bottlenecked on the CPU. Let us explore an improvement to this solution by introducing a load balancer.

Solution #2: Hash Table, Multiple Hosts, Load-Balancing

Figure 3 shows the updated design with a load-balancer and the processing now distributed between two different hosts. The solution works a bit better now. The total throughput of the system has increased now. However, the system can still get bound by memory. Typically, YouTube may end up having billions of videos. Storing these into a single host machine may not be feasible as the server may start to run

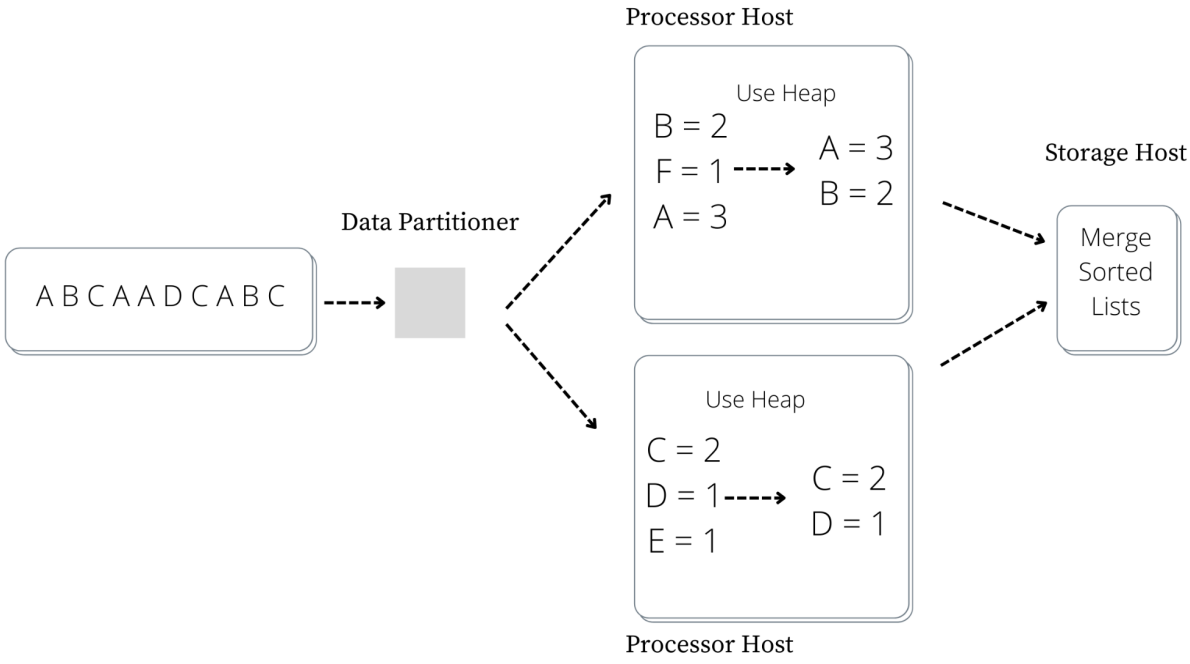
into OOM (Out-Of-Memory) issues.



Solution #3: Hash Table, Multiple Hosts, Partitioning

Figure 4 shows a data partitioner that sits in front of the processor hosts. Each processor contains a subset of the videos. Now each of the processor hosts contains its own list of top k heavy hitters. To get the final list we will need to merge the two sorted lists. It is important to note that we cannot pass the entire hash table list to the storage host.

Hash Table, Multiple Hosts, Partioning



Complexities Involved

While the design with data partitioning makes the system extremely scalable, we still need to deal with certain complexities in building the architecture. Let us go through these one-by-one and discuss them in detail.

Complexity #1: Data Replication

The first problem that the system needs to deal with is to ensure that the data is replicated across the different nodes. Let us go through and revise what data replication is first. (I covered some of the fundamentals of

system design in this blog post, in case you are interested in revisiting it).

Data replication is a method of copying data to ensure that all information stays identical in real-time between all data resources. Think of database replication as the net that catches your information and keeps it falling through the cracks and getting lost. [3] Why is data replication required? Data replication is required to ensure that the system does not lose data in case one of the processor hosts shuts down during the course of the system's lifetime. Generally, a replication factor of 3 is used in real-world systems. This means that any dataset is kept at a primary host and copied to two more hosts on the network. It is worth noting that the probability of simultaneous failure of three different nodes is close to zero.

Complexity #2: Rebalancing

Computer machines tend to be faulty and can often crash. What should the ideal response of a system be in case one of the machines crashes? What if we want to add more nodes to the cluster so that we can get a

[3] https://redis.com/blog/what-is-data-replication/

higher level of parallelism from the system? The answer to those questions is building a system that can rebalance itself. Let us discuss the two scenarios where a node is added to the system and when a node is deleted from the system in the next subsection.

Case #1: Addition of a new node.

In case a new node is added to the cluster, we need to ensure that data from existing nodes is divided and moved to the new node. This is achieved by keeping the metadata for the cluster in the Data Partitioner. What does the metadata contain? Ideally, the metadata contains the following:

- Identifier for each processor host in the cluster;
- The range of data contained on each of the processor hosts;
- The replicated copies of the data and its location on the different processor hosts.

When a new node is added to the system, the metadata is updated to reflect the movement of data to the new node.

Case #2: Deletion of a node from a cluster.

In case a node is deleted from the cluster, the data that resides on the node needs to move to other nodes. The partitioning node updates the metadata and sends a request to the new nodes so that they query the old nodes and receive the data from them.

Now, we may wonder what complexity underlines the aforementioned situations. There are many tricy edge-cases and situations which may arise. Some of the situations that can cause the system to fail are as follows:

- Network partition in case data was being moved during rebalancing.

- A node goes out of memory or disk and therefore cannot receive the extra dataset.
- The partitioner node goes down and the metadata becomes inconsistent with the cluster state.

Complexity #3: Dealing With Hot Partitioning

The last of the complexities that I want to talk about is the problem of hot partitioning. The distribution of access usually follows a Zipfian curve or a heavy-tailed distribution curve. Take for example this tweet by Ellen Degneres that was retweeted more than 3 million times or the video stream of an IPL final on Hotstar which had 18.6 Million concurrent connections. The record was set during the Indian Premier League (IPL) final match between the Chennai Super Kings and Mumbai Indians. Some amazing numbers! However, the issue with such access patterns tends to be that one of the nodes on the system would become heavily loaded and would become a laggard in the system. The overall

performance of the system is dependent on the slowest link in the overall chain. There are several mechanisms to deal with hot-partitioning.

Merge N Sorted Lists Algorithm

Once the data has been partitioned on each of the individual processor nodes, it is sorted using a heap and then merged in the Storage Processor. An n-way merge algorithm is used to merge the sorted lists. We have kept the scope of the algorithm out of this work.

Drawbacks:

The solution that we have works for a set of videos considering a bounded time interval. However, in a real world scenario keeping track of all the videos visited within a day on YouTube may not be entirely feasible.

Data Structure

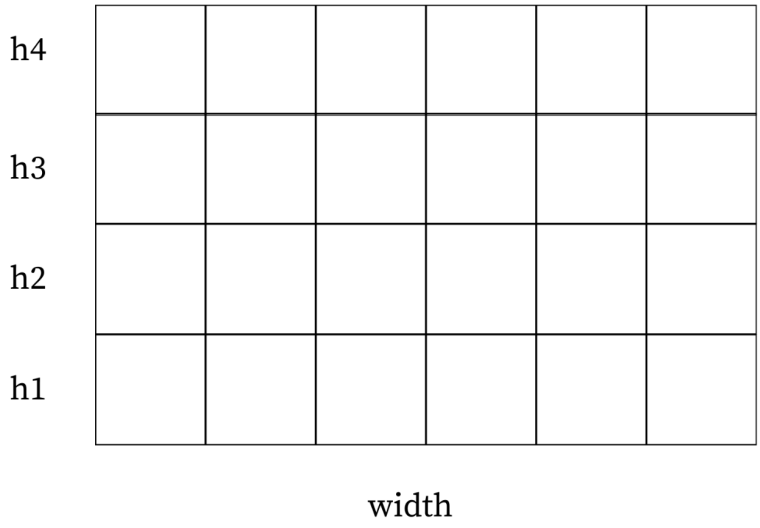
Before we go further, let us first think through the possible data structures that can help us achieve keeping an approximate counter within a bounded memory space.

Count–min Sketch

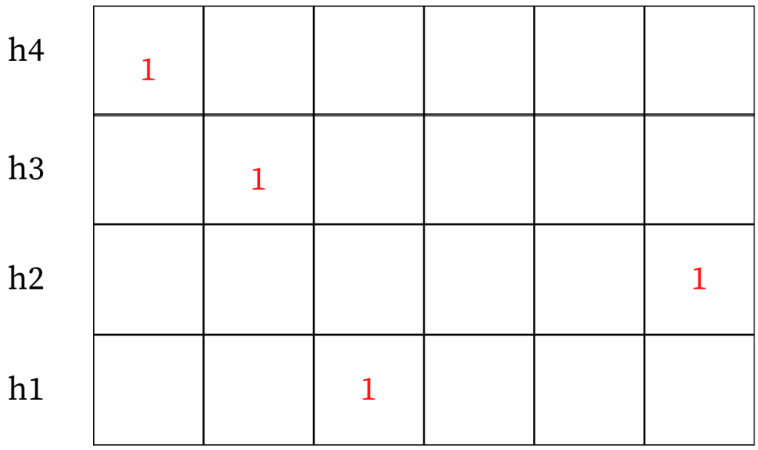
Count-min Sketch is a probabilistic data structure meaning that the frequencies computed may not be accurate but will have some bounded errors. The goal of the basic version of the count–min sketch is to consume a stream of events, one at a time, and count the frequency of the different types of events in the stream. At any time, the sketch can be queried for the frequency of a particular event type I from a universe of event types U, and will return an estimate of this frequency that is within a certain distance of the true frequency, with a certain probability. Let us first take a deep-dive into the design of the count-min sketch data structure.

We can imagine it as a 2-D array. It has a width (W) and a height (H). Figure …. represents a representation of an empty count-min sketch data structure.

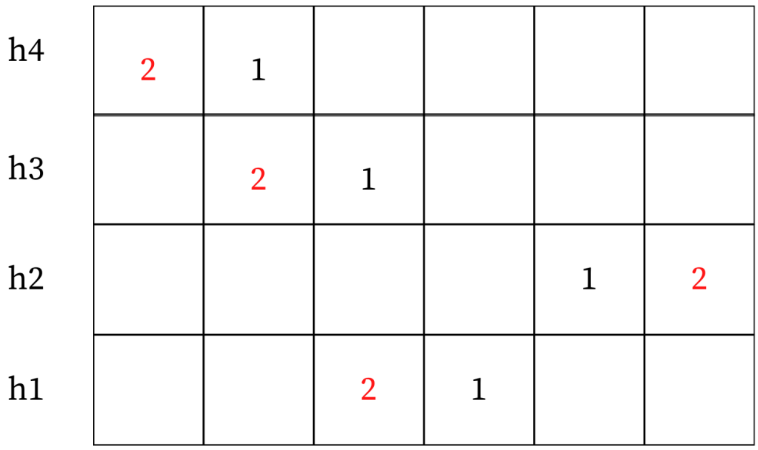
Let us go through an example to understand the data structure. Let us say that we need to store the counts of the following inputs: “A A B C”. We go through the figures XXX. to XXX. to understand how the data structure is updated and then how do we retrieve the count corresponding to each of the elements (i.e. A, B, and C). Note, the counts corresponding to each of the elements remain approximate when we use this data-structure. The figures below show a simple diagram as to how the elements are stored in the data structure.



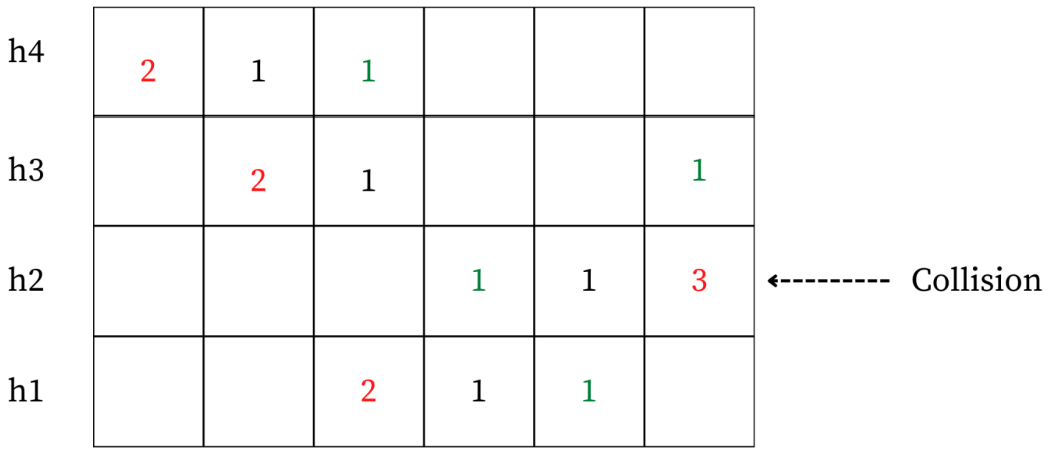
A



A A B


h4
2
1
h3
2
1
h2
1
2
h1
2
1
width

A A B C


h4
2
1
1
h3
2
1
1
h2
1
1
3
Collision
h1
2
1
1
width

High-level Architecture

Before we go through each of the components in detail, let us first discuss the higher level data flow for the system. We will discuss the design of the system with two design goals in mind.

Design Goal #1: A system that is fast (gets results within seconds) — the results are approximate.

Design Goal #2: A system that is relatively slower (gets results within minutes to hours) — the results are accurate.

It is worth noting to discuss this question with the interviewer and understand whether they would be interested in design goal #1 or design goal #2 or both. You can then discuss designs based on what the interviewer is looking for.

Higher-level Data Flow

Before we go any further, let us discuss the higher-level data flow and then we can discuss the details of each of the components.

API Gateway

The most important function of an API Gateway is to route the request from the client to the backend services. In the modern tech stacks, most of the distributed systems use API gateways in the distributed systems. For our use-case we are interested in using the API Gateway to look at the request logs. The request logs contain information as to which video was viewed by a user. The API Gateway is designed as follows:

- Single entry point for all clients.
- Aggregates data on the fly or via a background process the processes logs. Data is flushed based on either time or size. A bounded-sized memory buffer is used in the API Gateway to map each of the requests to a count. It can be implemented using a Hash-Table as discussed before in this chapter. Once the buffer becomes full, the data is flushed onto a disk.
- Last but not the least, the data is serialized in a compact binary format (e.g. Apache Avro) to bring efficiency. This is done to ensure

that the size of the data that is transferred over the network is compacted and bandwidth utilization is low.

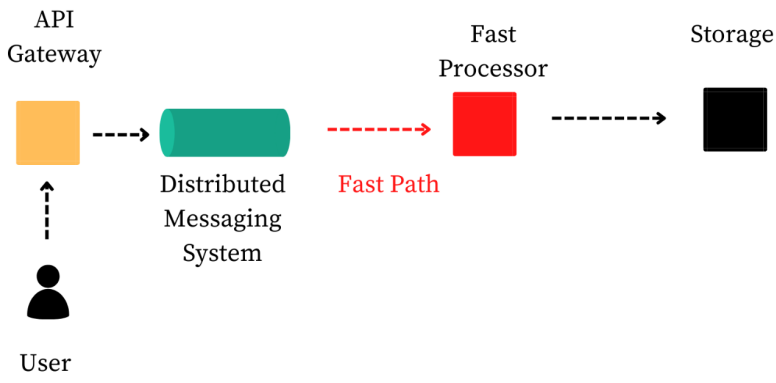
Distributed Messaging System

The data is then sent to a distributed messaging queue such as Apache Kafka, or AWS Kinesis. Internally, For the scope of this problem, we do not have any specific preference for a partitioning scheme. A random partitioning scheme will serve the purpose of distributing data across different nodes within the cluster.

Fast Path

On the fast path we will calculate an approximate list of heavy hitters and we will ensure that the result is available within seconds.

High-Level Architecture (Fast Path)



Fast Processor

The first component on the fast path is a service called the fast processor. It creates a count-min sketch and aggregates data for a short period of time (seconds). Because memory is no longer a problem, no need to partition. In order to ensure that the system remains highly available (one of the requirements of the system's design), we need to ensure that the data is replicated. Otherwise, the system may not remain available when one of the processor nodes shuts down. However, given we have the Slow Path as a backup to the Fast Path, it may be a fine

tradeoff to make. It is here where it is good to discuss this as a tradeoff with the interviewer and understand the requirements clearly.

Storage

Every several seconds the fast processor flushes data to a secondary storage layer. It is good to note that the count-min sketch data structure has a predefined size in memory. This means that the memory requirements do not grow unbounded within the fast processor. So, we could keep the count of the videos views for a longer period of time in-memory in the fast processor. However, the drawbacks of keeping the data in-memory for a longer time are as follows:

1. The higher the time-window for which the data is not persisted on the secondary storage, the higher the probability of it being lost.
2. The longer the data remains in-memory the longer it takes for the final results to be calculated.

The storage layer can be built using SQL or NoSQL database schemes. In my opinion, both could work. It builds the final count-min sketch and

stores a list of top k elements for a period of time. Given this layer ensures that the data remains durable and highly available data replication is required.

Slow Path

On the slow path we will calculate a precise list of heavy hitters and the result will be

available within minutes or hours depending on the data volume. Given that our objective is to calculate an accurate answer to the problem, we will dump the data on the disk and then use one of the following options to calculate the topK heavy hitters. Let us go through the options and explore their pros and cons.

Option #1: Use MapReduce

In order to implement MapReduce, we can dump the data onto a distributed storage layer such as HDFS, or S3. We can then run two different MapReduce jobs.

Job #1: Count the frequency of each item (view of a YouTube video).

Job #2: Calculate the TopK hitters.

Option #2: Use Data Partitioner And Map Reduce

Step 1#: Using A Data Partitioner

- The data partitioner reads batches of events from the Distributed Messaging Queue and partitions them into individual events. The partitioner will use a key derived from the video identifier and time window to hash the keys into different partitions.
- Each of the partitions are then sent to different partitions of a messaging queue. The messaging queue can be Apache Kafka, AWS Kinesis or any other queuing system that supports distributed processing.
- It is also worth noting that the data partitioner should take care of hot partitions. In case a certain set of videos are being watched heavily, the partitioner will need to rebalance the data flowing into the stream for the specific data partition. This may be done by

creating a duplicate partition for the hot set of keys or by using a different hashing scheme.

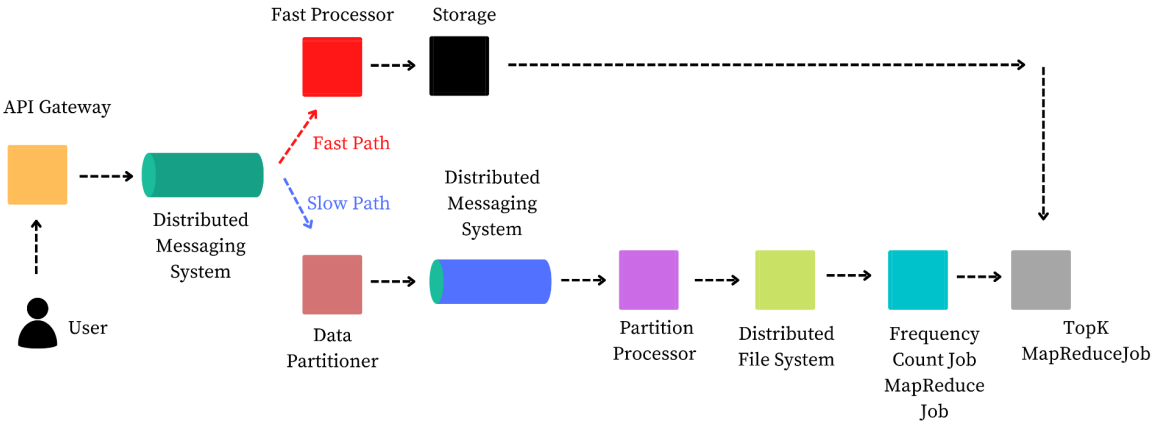
Step #2: Using A Distributed Messaging Queue

Each partition sent to the messaging queue will be sharded in Kafka or Kinesis – depending on the platform we end up using. In order to ensure availability, both Kafka and Kinesis take care of data replication — in case one of the shards becomes unavailable.

Step #3: Partition Processor

Next, we need a component that will read each partition and aggregate it further. Let us call this component a partition processor. The main job of the partition processor is to aggregate the partitioned data in-memory over the course of several minutes and generate the data in predefined file sizes and store it in the distributed file system.

High-Level Architecture



To get a weekly newsletter with System Design Problems, join more than 2,000 subscribers at:

[Ravi's System Design Newsletter]