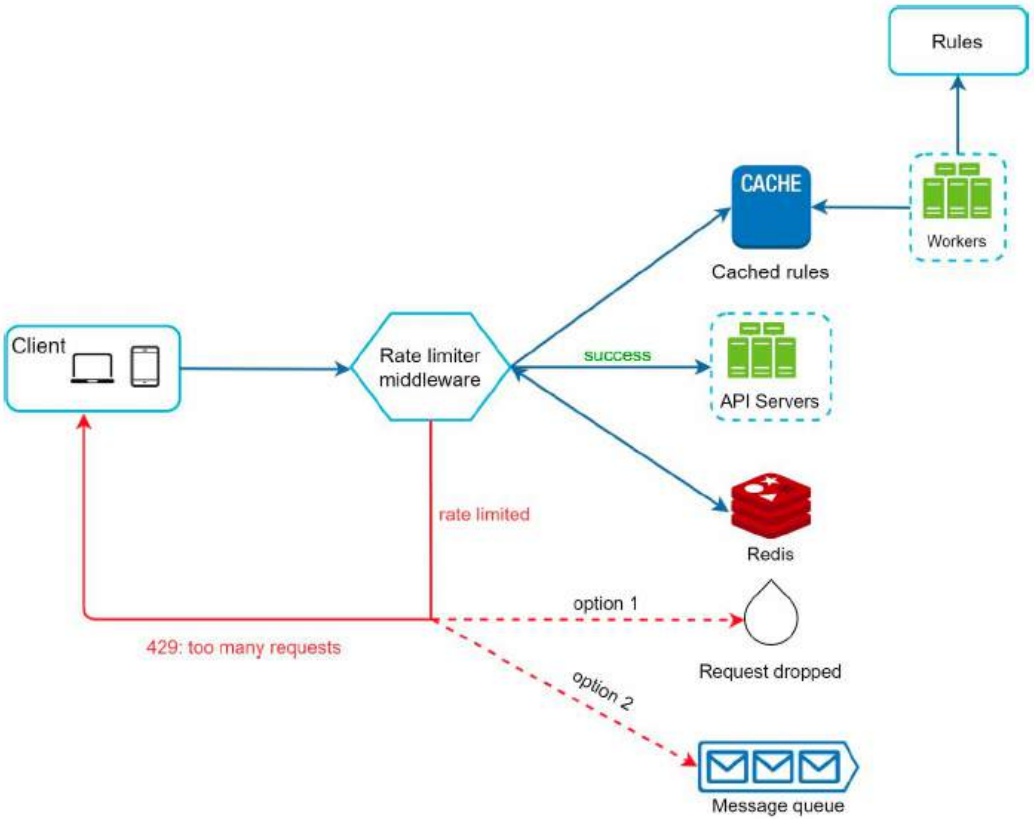
# 1 - RATE LIMITER

- **FINAL DESIGN**




  - When a client sends a request to the server, the request is sent to the rate limiter middleware first.
  - It fetches counters and last request timestamp from Redis cache.
  - Rules are stored on the disk. Workers frequently pull rules from the disk and store them in the cache.
  - Rate limiter middleware loads rules from the cache. Based on the response, the rate limiter decides:
    - if the request is not rate limited, it is forwarded to API servers.

- if the request is rate limited, the rate limiter returns 429 too many requests error to the client. In the meantime, the request is either dropped or forwarded to the queue.

- 

***Requirement gathering & Analysis***

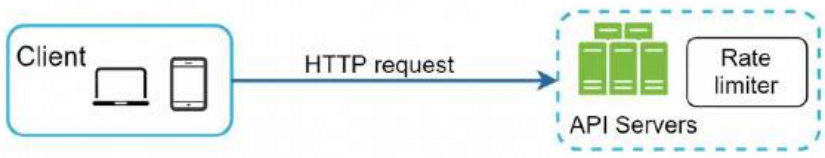
Accurately limit the requests. Reason why we use the rate limiter is to avoid any kind of DDOS attacks. IT could also increase the cost of operation if at all we are using third party APIs.

Distributed rate limiting. The rate limiter can be shared across multiple servers or processes

429 code to consumer. Logging on our end to analyze

***High-Level Design***

* You can implement a rate limiter at either the client or server-side but client is an unreliable place to enforce rate limiting because client requests can easily be forged by malicious actors. Moreover, we might not have control over the client implementation. Following shows a rate limiter that is placed on the server side.



* Besides the client and server-side implementations, there is an alternative way. Instead of putting a rate limiter at the API servers, we create a rate limiter middleware, which throttles requests to your APIs as shown below. It is something like API gateway

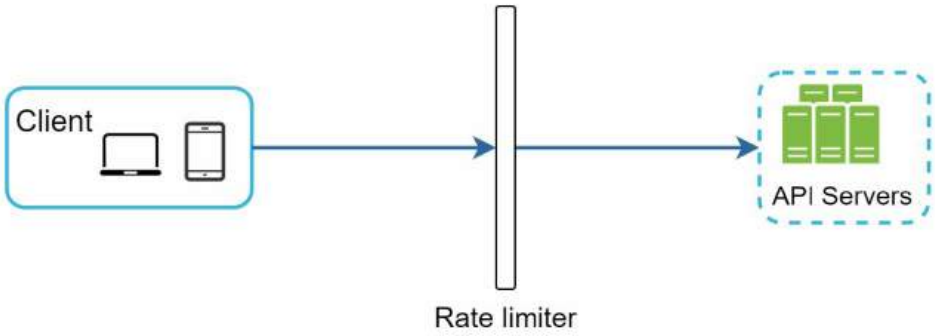



Figure 4-2

**Algorithms**

* Various algorithms can be used to implement the Rate Limiter such as Token Bucket, Fixed Window, Leaking Bucket etc

* Although, it's not totally necessary to go into details of these algorithms in a design interview, it's always nice to have knowledge on these subjects

**** Token bucket***

- A token bucket is a container that has pre-defined capacity. Tokens are put in the bucket at preset rates periodically. Once the bucket is full, no more tokens are added
-

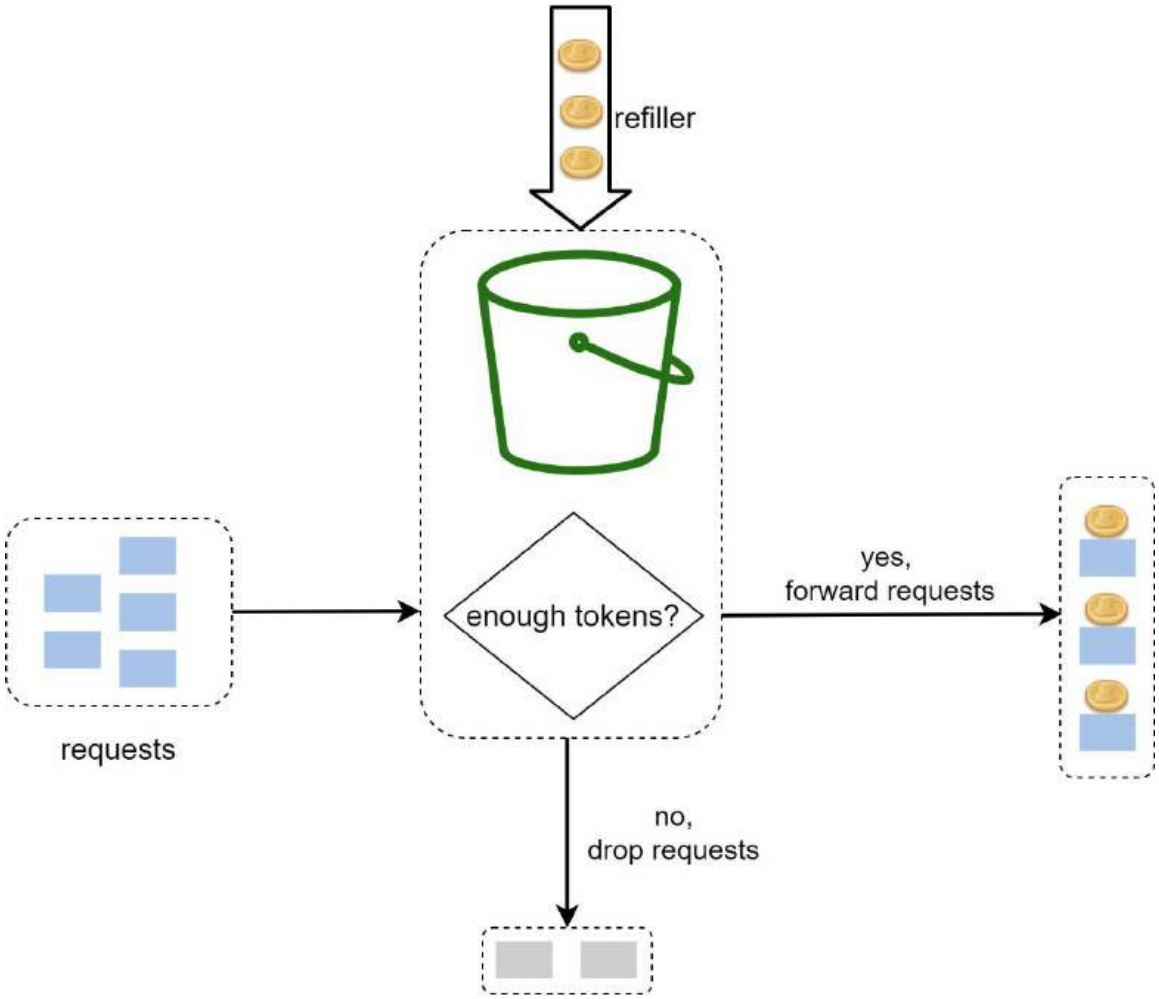



Figure 4-5

The token bucket algorithm takes two parameters:

- Bucket size: the maximum number of tokens allowed in the bucket
- Refill rate: number of tokens put into the bucket every second

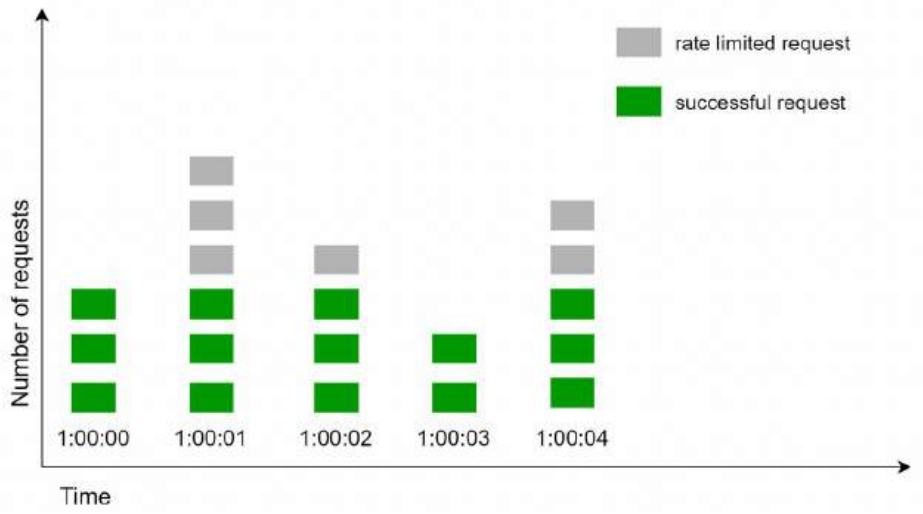
How many buckets do we need? This varies, and it depends on the rate-limiting rules. Here are a few examples.

- It is usually necessary to have different buckets for different API endpoints. For instance, if a user is allowed to make 1 post per second, add 150 friends per day, and like 5 posts per second, 3 buckets are required for each user.
- If we need to throttle requests based on IP addresses, each IP address requires a bucket.
- If the system allows a maximum of 10,000 requests per second, it makes sense to have a global bucket shared by all requests.

**** Fixed Window***

- The algorithm divides the timeline into fix-sized time windows and assign a counter for each window.
- Each request increments the counter by one.
- Once the counter reaches the pre-defined threshold, new requests are dropped until a new time window starts.

- Let us use a concrete example to see how it works. In Figure 4-8, the time unit is 1 second and the system allows a maximum of 3 requests per second. In each second window, if more than 3 requests are received, extra requests are dropped as shown in Figure 4-8.




- A major problem with this algorithm is that a burst of traffic at the edges of time windows could cause more requests than allowed quota to go through.

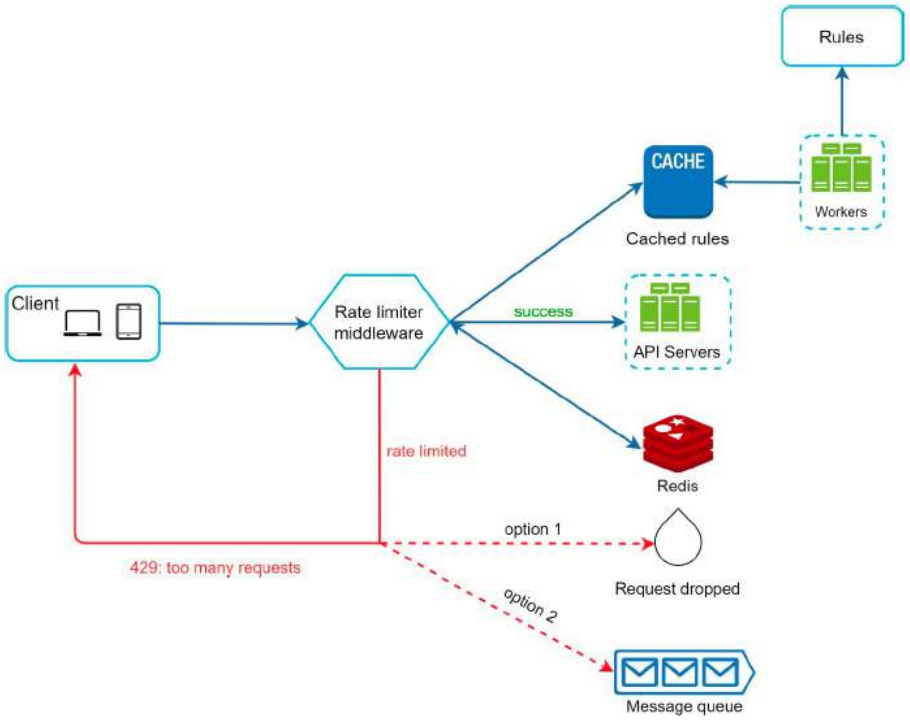
***[High-Level Design]***

* At the high-level, we need a counter to keep track of how many requests are sent from the same user, IP address, etc.

* Where to store the counters ??

    - DB is not a good idea due to disk slowness
    - Cache like Redis is the best solution
- * Rate limiting rules need to be created and stored in cache

**Explanation**




    - Rules are stored on the disk. Workers frequently pull rules from the disk and store them in the cache.
    - When a client sends a request to the server, the request is sent to the rate limiter middleware first.
    - Rate limiter middleware loads rules from the cache. It fetches counters and last request timestamp from Redis cache. Based on the response, the rate limiter decides:
        - if the request is not rate limited, it is forwarded to API servers.

        - if the request is rate limited, the rate limiter returns 429 too many requests error to the client. In the meantime, the request is either dropped or forwarded to the queue.

- * In a distributed env, how will you handle the synchronization as multiple clients could use diff Rate limiters ??

    - Centralized data store Redis cache. All of them can retrieve the info from cache

- In case a request is rate limited, APIs return a HTTP response code 429 (too many requests) to the client.

  * How does a client know whether it is being throttled? And how does a client know the number of allowed remaining requests before being throttled?

    - Answer lies in the Headers. In the headers, data like X-Ratelimit-Remaining, X-Ratelimit-Retry-After etc can be sent

***Monitoring***

After the rate limiter is put in place, it is important to gather analytics data to check whether the rate limiter is effective. Primarily, we want to make sure:

- The rate limiting algorithm is effective
- The rate limiting rules are effective.

# 2 - CONSISTENT HASHING

It's the concept which is primarily used when we need to load the balance among multiple servers. When data is distributed among n servers, we can hope to has the key and then % n in order to distribute it among the servers available. But, this design has serious drawbacks. Because, when a server is lost the value of n changes although the original hash of the key remains the same and hence the hash(key) % n value changes as well and the distribution cannot be expected to be even. This is where Consistent Hashing comes into play

The basic steps in consistent hashing algorithm are

- Map servers and keys on to the ring using a uniformly distributed hash function.
- To find out which server a key is mapped to, go clockwise from the key position until the first server on the ring is found.
  Hence, All the keys are directed to the server that is present first in the clock wise direction.

To determine which server a key is stored on, we go clockwise from the key position on the ring until a server is found. Figure 5-7 explains this process. Going clockwise, *key0* is stored on *server 0*; *key1* is stored on *server 1*; *key2* is stored on *server 2* and *key3* is stored on *server 3*.

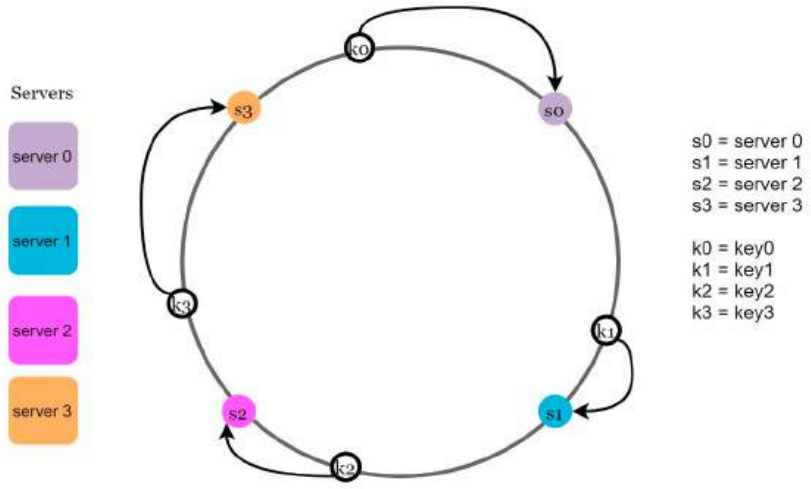
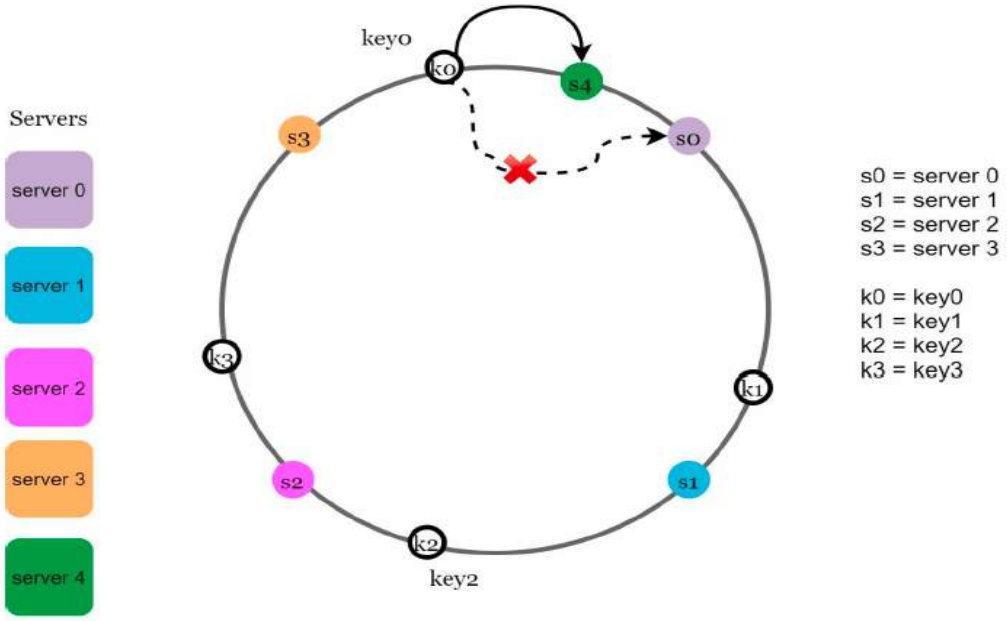



Figure 5-7

- In the figure below, after a new server 4 is added, only key0 needs to be redistributed. k1, k2, and k3 remain on the same servers. Let us take a close look at the logic. Before server 4 is added, key0 is stored on server 0. Now, key0 will be stored on server 4 because server 4 is the first server it encounters by going clockwise from key0's position on the ring. The other keys are not redistributed based on consistent hashing algorithm.




Although the solution is great, it still has a couple of issues.

- First, it is impossible to keep the same size of partitions on the ring for all servers considering a server can be added or removed. A partition is the hash space between adjacent servers. It is possible that the size of the partitions on the ring assigned to each server is very small or fairly large
- Second, it is possible to have a non-uniform key distribution on the ring.

Both are depicted below

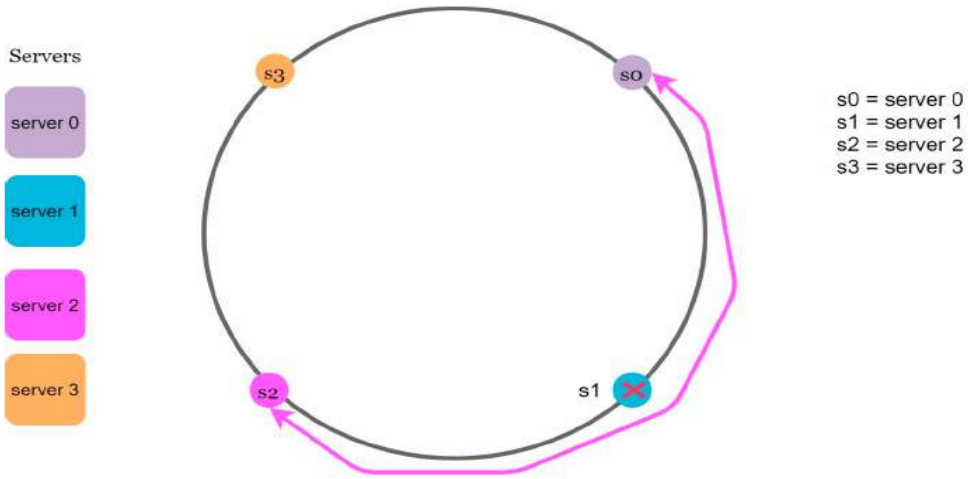



Figure 5-10

Second, it is possible to have a non-uniform key distribution on the ring. For instance, if servers are mapped to positions listed in Figure 5-11, most of the keys are stored on *server 2*. However, *server 1* and *server 3* have no data.

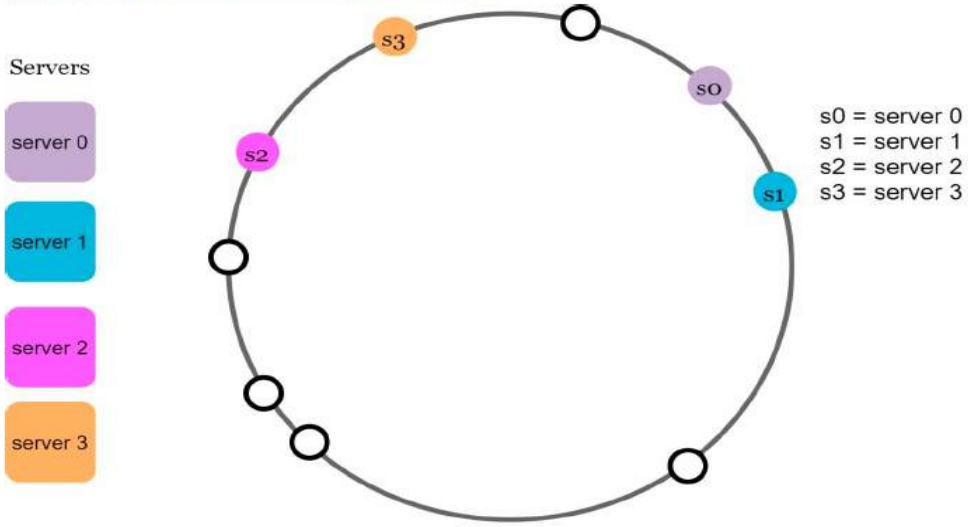



Figure 5-11

- And this is solved using virtual nodes. This doesn't mean we add more servers but it means that we create virtual copies of the existing servers. That is by hashing the servers using different hash function again. In that case, we can have the servers located at different positions on the ring and the key values can be expected to be distributed evenly enough instead of having all the keys mapped in skewed format

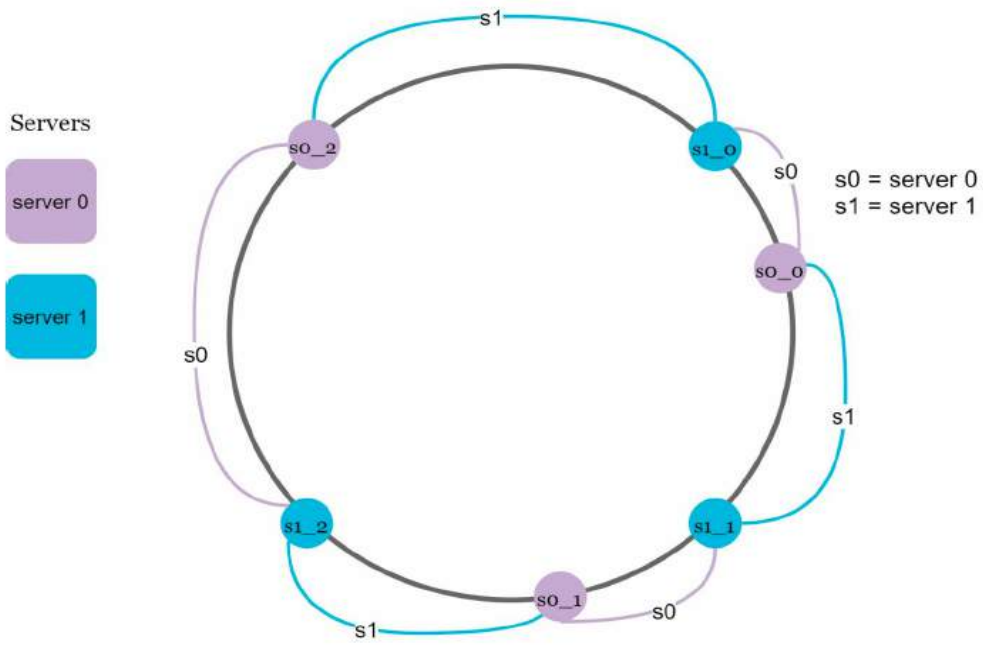
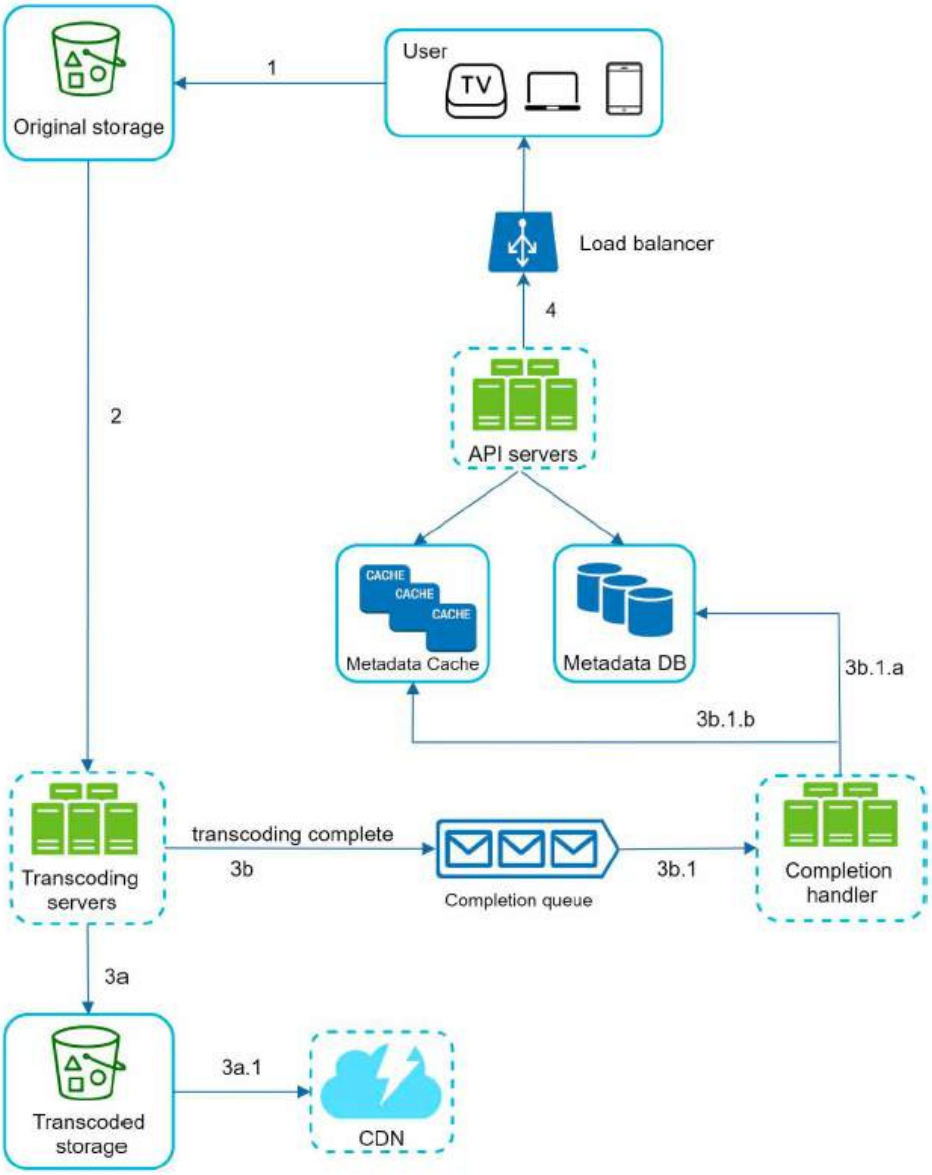
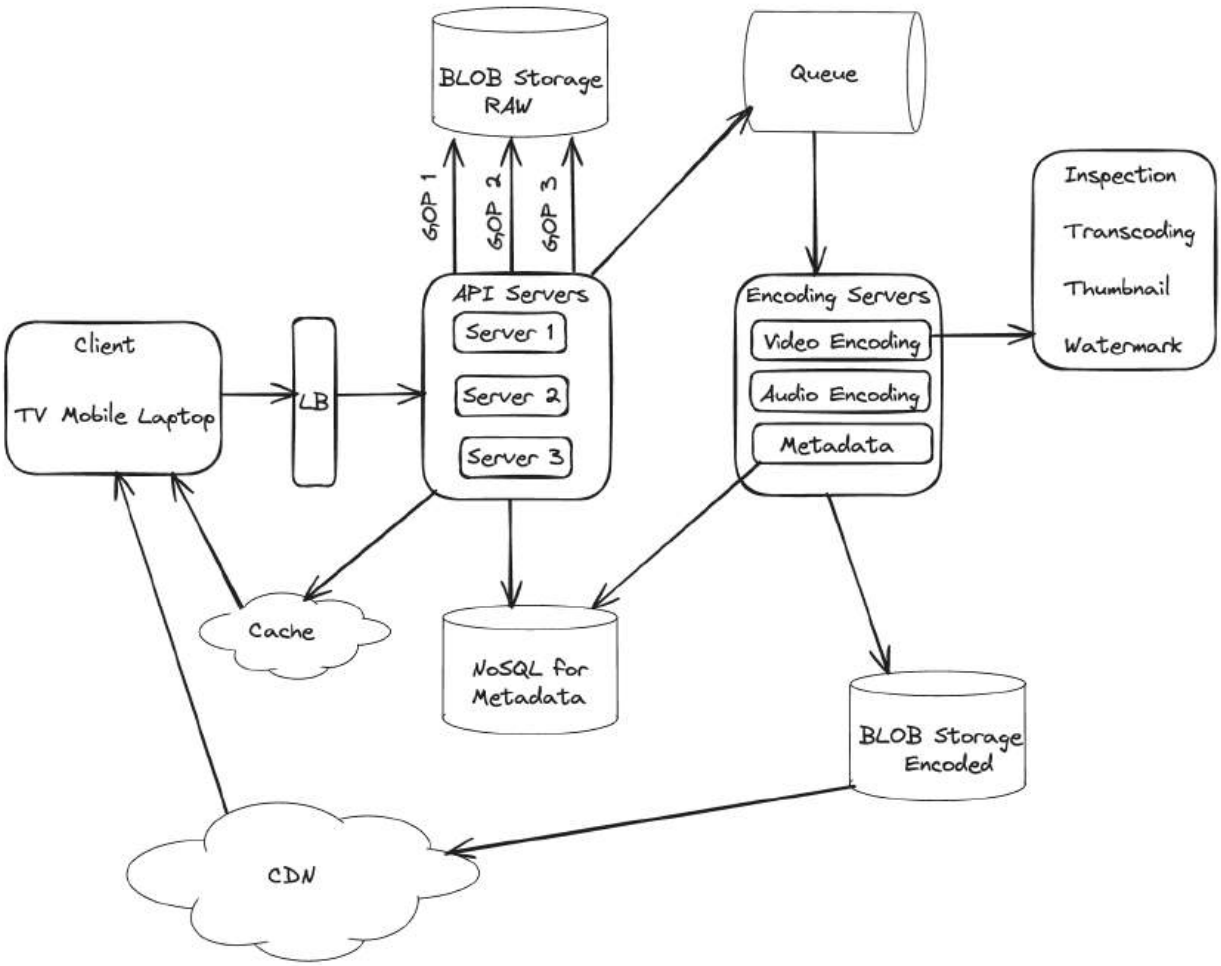

Servers
server 0
server 1
s1
s0
s0
s1
s0
s1
s0
s1
s0_2
s1_0
s0_0
s1_1
s0_1
s1_2
s0 = server 0
s1 = server 1

Figure 5-12

# 3 - NETFLIX / YOUTUBE






## STEP 1: UNDERSTANDING & QUESTIONS

- What all features?
- What all clients are we supporting?
- What is the approximate daily users count?
- Any file size requirement for the videos?
    - Should be under 1 GB
- Supporting all kinds of resolutions?
    - Yes
- What is the Read to Write ratio?

## STEP 2: CAPACITY ESTIMATION

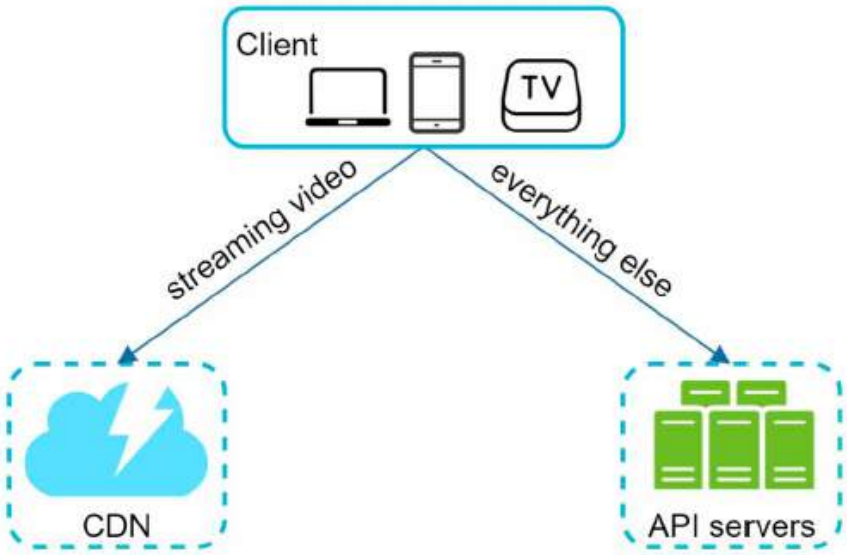
- 5 million Daily Active Users
- Each user watches 5 videos per day

- Only 10% of the users upload video
- **Avg video size can be approximated to 300 MB**
- Total daily storage needed
    - **5 million * 10% * 300 MB = 150 TB**

## STEP 3: HIGH-LEVEL DESIGN

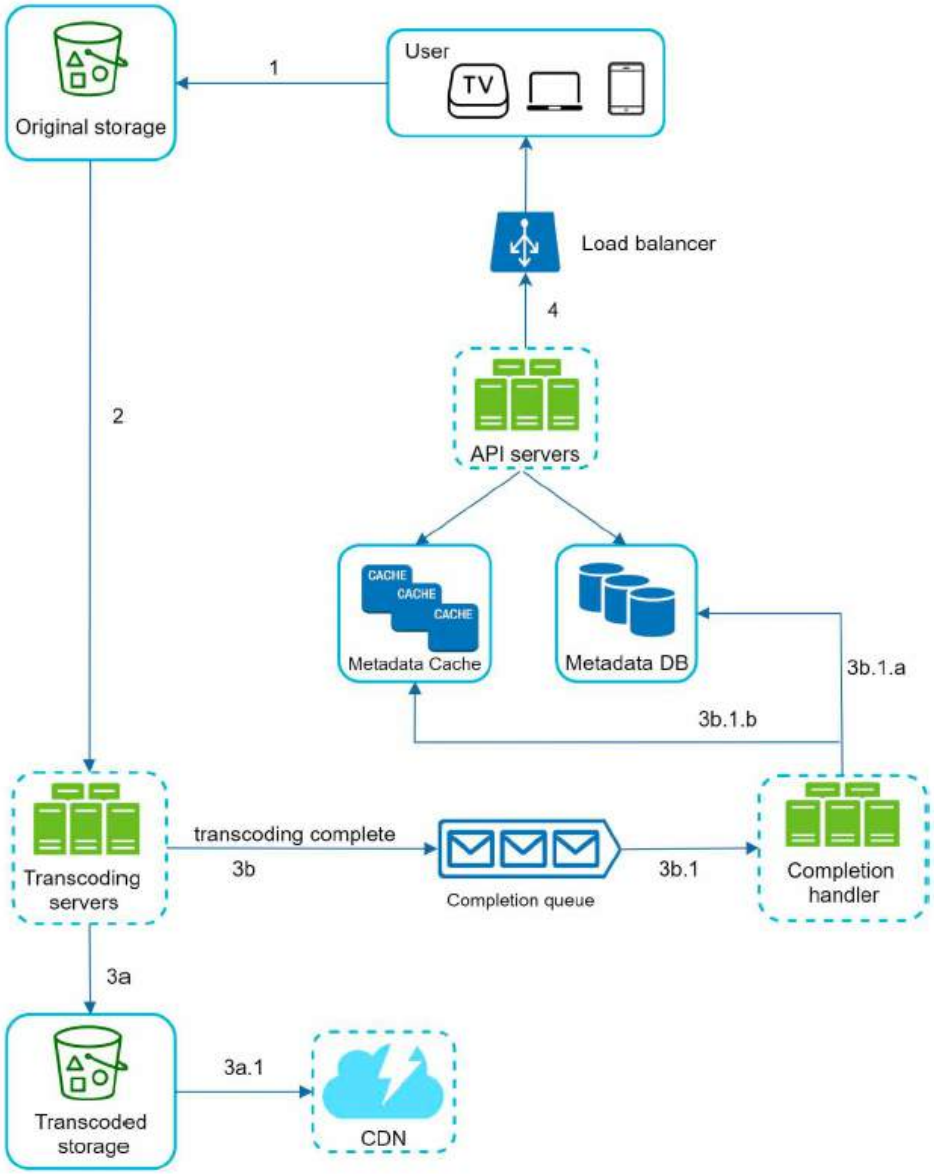
- **POST**
    - **uploadVideo(stream videoData, String videoTitle, String desc, String category, string[] tags)**
- **GET**
    - **viewVideo(videoId, codec, videoResolution)**

- At the High level, we have 3 components
    - Client
    - CDN for streaming the videos. CDN because it's impossible and impractical to load the videos every time from the storage
    - API Servers which handle the video uploads

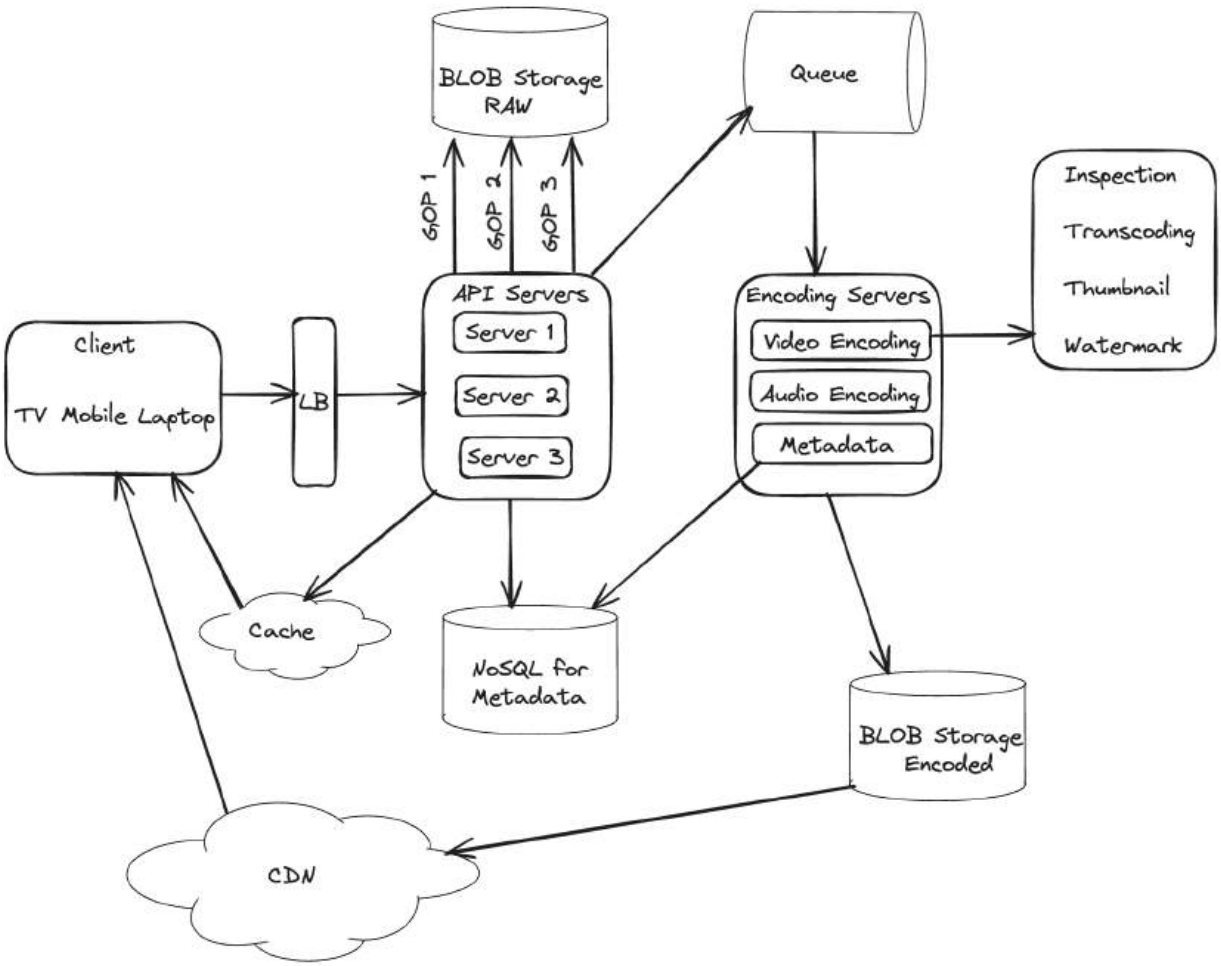



- So, there are 2 flows
    - Video Uploading Flow
    - Video Streaming Flow

## STEP 4: DESIGN DEEP DIVE

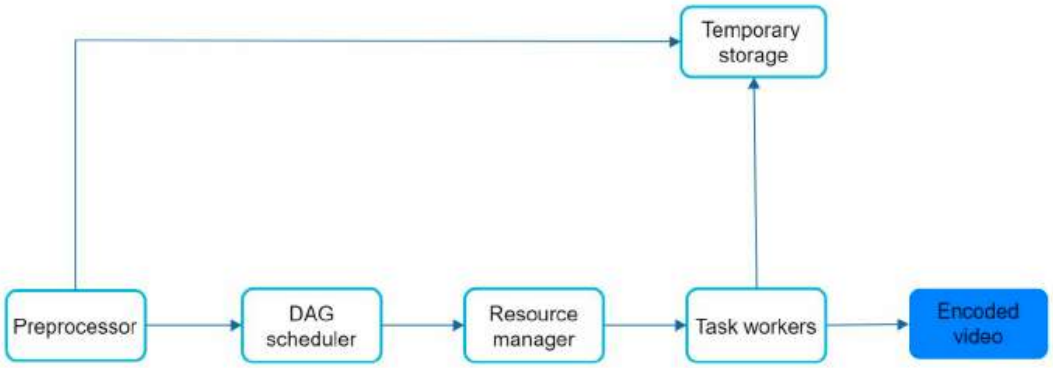



- Roughly, this is how the flow is going to look like.




- A client uploads a video to server through Load Balancer. Load Balancer also implements Rate Limiting which is kind of unwritten rule
- These videos can be uploaded in chunks to the BLOB Storage
- At the same time, Metadata can be stored in a NoSQL DB
- Once the video upload is complete, an event can be placed in the Queue
- Video Encoding is one of the important features. These workers poll the queue constantly and check for any events. Audio encoding happens as well.
- Video Encoding
    - Inspection
        - Checking for any malformed or unwanted data. Sanity check
    - Transcoding
        - Converting the video into multiple formats and also multiple qualities. In this way, we can send the higher quality video when there is higher bandwidth and otherwise

    - Thumbnail
        - We can generate thumbnails and store
    - Watermark
        - For copyright, we can have them watermarked
- Inside the encoding, again there could be multiple stages such as ordering the tasks in terms of priority, placing them in queues and then workers doing the jobs based on that etc




- Once the encoded video is formed, **it is stored in the blob storage and eventually pushed into CDN.** This is where the end user is served the videos from.

## OPTIMIZATIONS

- Parallelize Video Uploading
    - We parallelized the video uploading where the blocks of video is uploaded instead of whole video at once. That is efficient
- We can have Queues everywhere even inside the encoding servers to parallelize everything and increase the speed
- We can ensure only authorized users upload the videos. Authorization can be done in LB
- We can copyright the videos to avoid any copying
- CDNs are crucial when the videos are served to users. Usually only a few popular videos are accessed frequently but many others have few or no viewers. Hence we can only serve the most popular videos from CDN and other videos from our high capacity servers

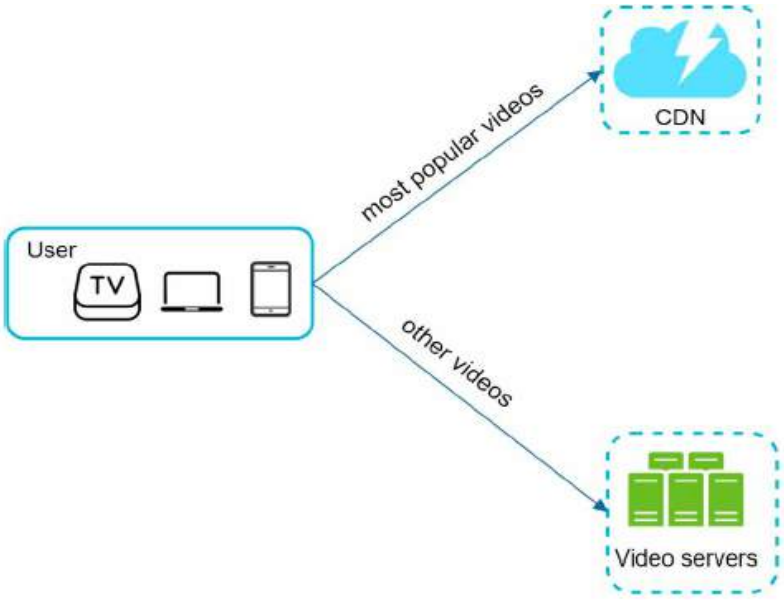


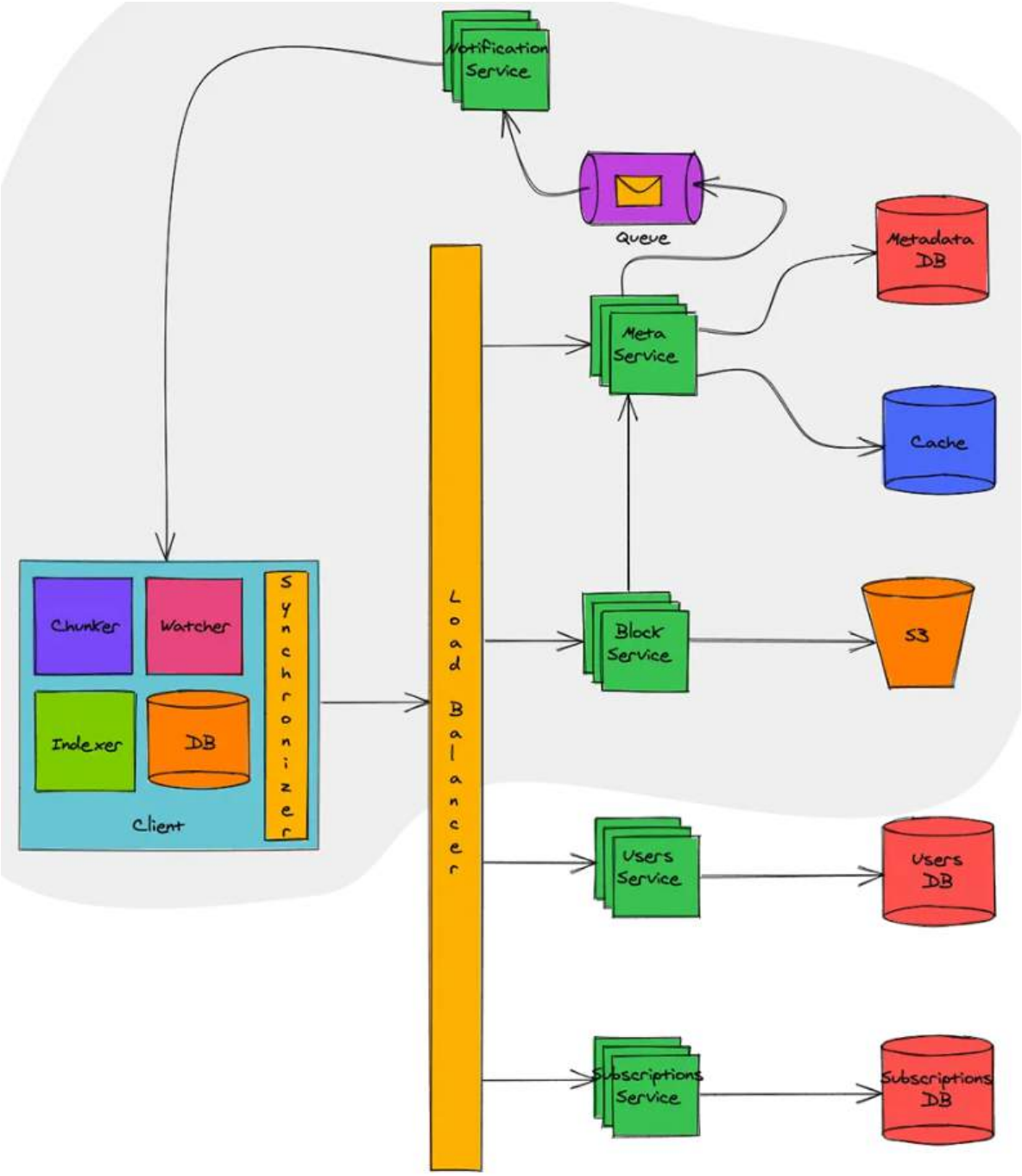

Figure 14-28

2. For less popular content, we may not need to store many encoded video versions. Short videos can be encoded on-demand.

3. Some videos are popular only in certain regions. There is no need to distribute these videos to other regions.

# 4 - GOOGLE DRIVE / DROP BOX

## FINAL DESIGN (CONCENTRATE THE PORTION IN GREY AREA)




## STEP 1: UNDERSTANDING & QUESTIONS

- Is this a web app or mobile app or both?
- What all features are needed as part of this design?
- What is the approx users count?
- What are the supported file types and do they need to be encrypted?

- File size limit??
- Reliable
- Fast sync between documents
- Scalability
- Availability

## Functional Requirements

1. Users should be able to sign up using their email address and subscribe to a plan. If they don't subscribe, they will get 1 GB of free storage.
2. Users should be able to upload and download their files from any device.
3. Users should be able to share files and folders with other users.
4. Users should be able to upload files up to 1 GB.
5. System should support automatic synchronization across the devices.
6. System should support offline editing. Users should be able to add/delete/modify files offline and once they come online, changes should be synchronized to remote server and other online devices.

## Non-functional requirements

1. System should be highly reliable. Any file uploaded should not be lost.
2. System should be highly available.

## Out of scope

1. Real-time collaboration on a file.
2. Versioning of the file.

## STEP 2: CAPACITY ESTIMATION

## Assumptions

1. Total number of users = 500 million
2. Total number of daily active users = 100 million
3. Average number of files stored by each user = 200
4. Average size of each file = 100 KB
5. Total number of active connections per minute = 1 million

## Storage Estimations

Total number of files = 500 million * 200 = 100 billion
Total storage required = 100 billion * 100 KB = 10 PB

STEP 3: APIs & Schema

# Download Chunk

This API would be used to download the chunk of a file.

Request:

```
GET /api/v1/chunks/:chunk_id
X-API-Key: api_key
Authorization: auth_token
```

### Upload Chunk

This API would be used to upload the chunk of a file.

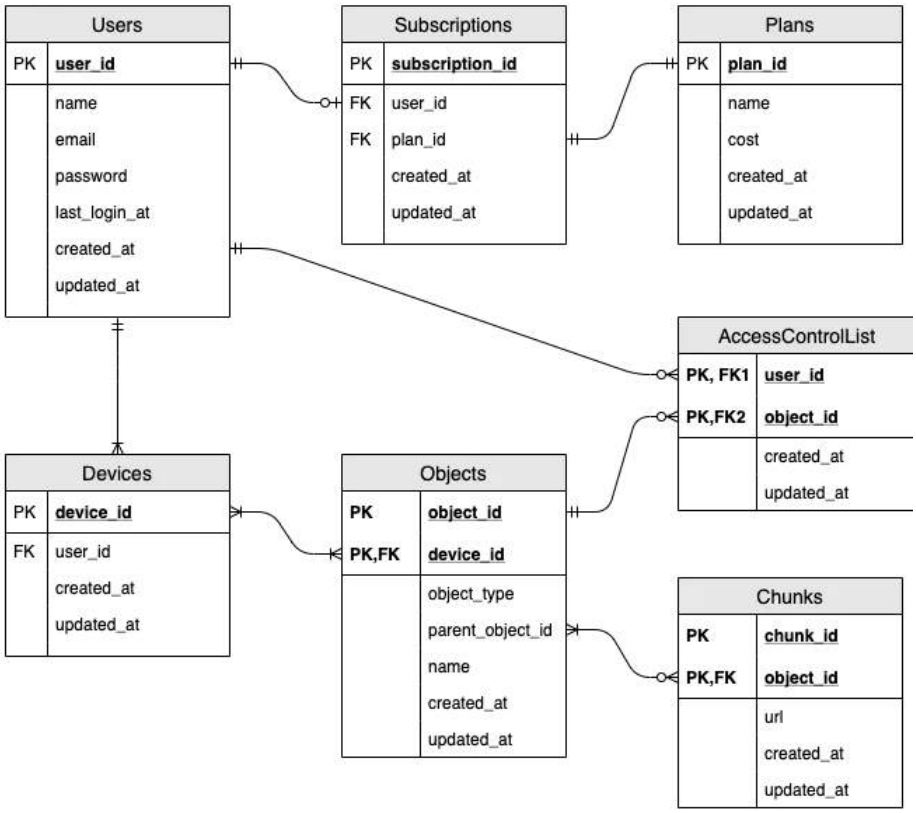
Request:

```
POST /api/v1/chunks/:chunk_id
X-API-Key: api_key
Authorization: auth_token
Content-Type: application/octet-stream
/path/to/chunk
```

`Content-Type` header is set as `application/octet-stream` to tell server that a binary date is being sent.

## Database Schema

The database schema containing most important tables is illustrated below:



## STEP 4: HIGH-LEVEL DESIGN

Keeping this in mind, let's look in to the different components of this optimized client:

**Client Metadata Database:** Client Metadata Database stores the information about different files in workspace, file chunks, chunk version and file location in the file system. This can be implemented using a lightweight database like SQLite.

**Chunker:** Chunker splits the big files in to chunks of 4 MB. This also reconstructs the original file from chunks.

**Watcher:** Watcher monitors for file changes in workspace like update, create, delete of files and folders. Watcher notifies Indexer about the changes.

**Indexer:** Indexer listens for the events from watcher and updates the Client Metadata Database with information about the chunks of modified file. It also notifies Synchronizer after committing changes to Client Metadata Database.

**Synchronizer:** Synchronizer listens for events from Indexer and communicates with Meta Service and Block service for updating meta data and modified chunks of file on remote server respectively. It also listens for changes broadcasted by Notification Service and downloads the modified chunks from remote server.

## Meta Service

Meta Service is responsible for synchronizing the file metadata from client to server. It's also responsible to figure out the change set for different clients and broadcast it to them using Notification Service.

When a client comes online, it pings Meta Service for an update. Meta Service determines the change set for that client by querying the Metadata DB and returns the change set.

If a client updates a file, Meta Service again determines the change set for other clients watching that file and broadcasts the change set via Notification Service.

Meta Service is backed by Metadata DB. This database contains the metadata of file like name, type (file or folder), sharing permissions, chunks information etc. This database should have strong ACID (atomicity, consistency, isolation, durability) properties. Hence a relational database, like MySQL or PostgreSQL, would be a good choice.

Since querying the database for every synchronization request is a costly operation, a in-memory cache is put in front of Metadata DB. Frequently queries data is cached in this cache thereby eliminating the need of database query. This cache can be implemented using Redis or Memcached and write-

- 

***Amazon S3***

## Block Service

Block Service interacts with block storage for uploading and downloading of files. Clients connect with Block Service to upload or download file chunks.

When a client finishes downloading file, Block Service notifies Meta Service to update the metadata. When a client uploads a file, Block Service on finishing the upload to block storage, notifies the Meta Service to update the metadata corresponding to this client and broadcast messages for other clients.

- 

## Notification Service

Notification Service broadcasts the file changes to connected clients making sure any change to file is reflected all watching clients instantly.

Notification Service can be implemented using HTTP Long Polling, Websockets or Server Sent Events. Websockets establish a persistent duplex connection between client and server. It is not a good choice in this scenario as there is no need of two way communication. We only need to broadcast the message from service to client and hence it's an overkill.

HTTP Long Polling is a better choice as server keeps the connection hanging till a data is available for client. Once data is available, server sends the data closing the connection. Once the connection is closed, client has to again establish a new connection. Generally, for each long poll request, there is a timeout associated with it and client must establish a new connection post timeout.

Notification Service before sending the data to clients, reads the message from a message queue. This queue can be implemented using RabbitMQ, Apache ActiveMQ or Kafka. Message queue provides a asynchronous medium of communication between Meta Service and Notification Service and thus Meta Service need not to wait till notification is sent to clients. Notification service can keep on consuming the messages at it's own pace without affecting the performance Meta Service. This decoupling also allows us to scale both services independently.

## Caching

We are using an in-memory caching to reduce the number of hits to Metadata DB. In-memory caches like Redis and Memcached cache the data from database in key-value pair.

From Meta Service servers, before hitting the Metadata DB, we are checking if data exists in cache or not. If it exists then we return the value from there itself bypassing a database trip. However if data is not present in cache, we hit the database, getting the data and populating the same in cache too. Hence subsequent requests won't hit the database and get the data from cache itself. This caching strategy is known as write-around caching.

Least Recently Used (LRU) eviction policy is used for caching data as it allows us to discard the keys which are least recently fetched.

SOURCES

- https://systemdesignprimer.com/dropbox-system-design/
- Alex Xu

# 5 - URL SHORTENER

## FINAL DESIGN

### URL redirecting deep dive

Figure 8-8 shows the detailed design of the URL redirecting. As there are more reads than writes, <*shortURL*, *longURL*> mapping is stored in a cache to improve performance.

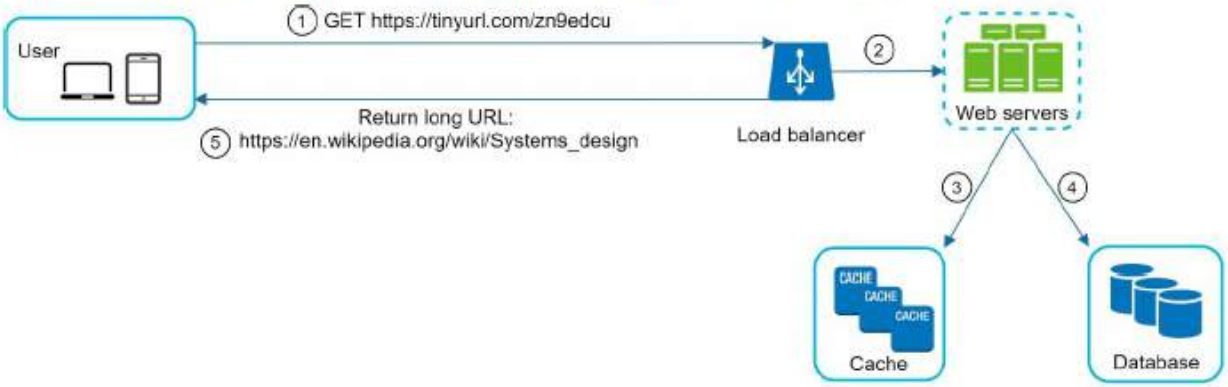



Figure 8-8

The flow of URL redirecting is summarized as follows:

1. A user clicks a short URL link: https://tinyurl.com/zn9edcu
2. The load balancer forwards the request to web servers.
3. If a shortURL is already in the cache, return the longURL directly.
4. If a shortURL is not in the cache, fetch the longURL from the database. If it is not in the database, it is likely a user entered an invalid shortURL.
5. The longURL is returned to the user.

## STEP 1: UNDERSTANDING & QUESTIONS

- What is the volume we are talking about

    - 100 Million URLs per day
- How short should the shortened URL be?
    - As short as possible
- What characters are allowed in Short URL?
    - 0-9 and a-z and A-Z
- Take a long URL and convert to Short URL
- When entered a Short URL, give the original Long URL

## STEP 2: CAPACITY ESTIMATION

- 100 M URLs per day
    - 100 M / 100000 seconds = **1000 writes per second**
- Read to write ratio 10 : 1
    - Reads = 1000 * 10 = **10000 reads per sec**
- Assuming the URL shortener service will run for 10 years
    - 100 million * 365 * 10 = 365 billion records.
    - Assume average URL length is **100 bytes**
    - **Storage requirement over 10 years: 365 billion * 100 bytes * 10 years = 365 TB**

## STEP 3: APIs & DATA MODEL

- **POST api/v1/data/shorten**
    - request parameter: {longUrl: longURLString}
    - return shortURL
- **GET api/v1/shortUrl**
    - Return longURL for HTTP redirection

One thing worth discussing here is 301 redirect vs 302 redirect.

**301 redirect.** A 301 redirect shows that the requested URL is "permanently" moved to the long URL. Since it is permanently redirected, the browser caches the response, and subsequent requests for the same URL will not be sent to the URL shortening service. Instead, requests are redirected to the long URL server directly.

**302 redirect.** A 302 redirect means that the URL is "temporarily" moved to the long URL, meaning that subsequent requests for the same URL will be sent to the URL shortening service first. Then, they are redirected to the long URL server.

- Store <shortURL, longURL> mapping in a relational DB

## STEP 4: DESIGN DEEP DIVE

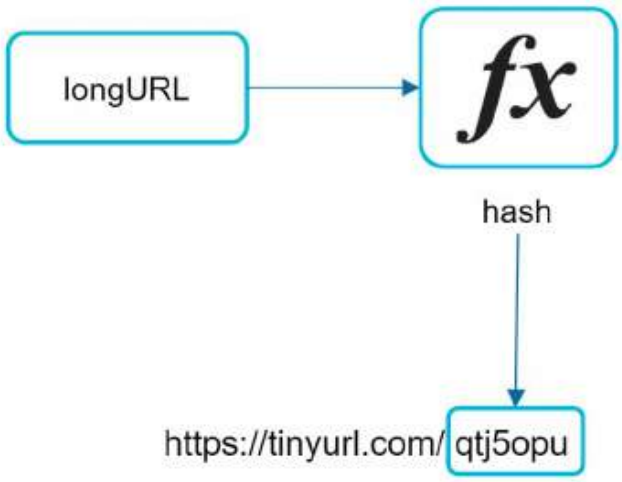



Figure 8-3

The hash function must satisfy the following requirements:

- Each *longURL* must be hashed to one *hashValue*.
- Each *hashValue* can be mapped back to the *longURL*.

- Hash value can contain 62 characters (0-9, a-z, A-Z)
- We must support 365 billion URLs as discussed early and we have 62 character set
    - **62 power of 7 = 3.5 trillion combinations**
    - **Hence, short url can be of 7 characters length**
- ***HASH FUNCTION***

- ***Approach 1 (INEFFICIENT)***
    - Use functions like MD5 etc. but length won't be 7 characters long
    - The first approach is to collect the first 7 characters of a hash value; however, this method can lead to hash collisions. To resolve hash collisions, we can recursively append a new predefined string until no more collision is discovered
    - DB querying is expensive. Not Efficient

- ***Approach 2 (EFFICIENT)***
    - Base 62 conversion
    - Generate a unique number and convert using base 62

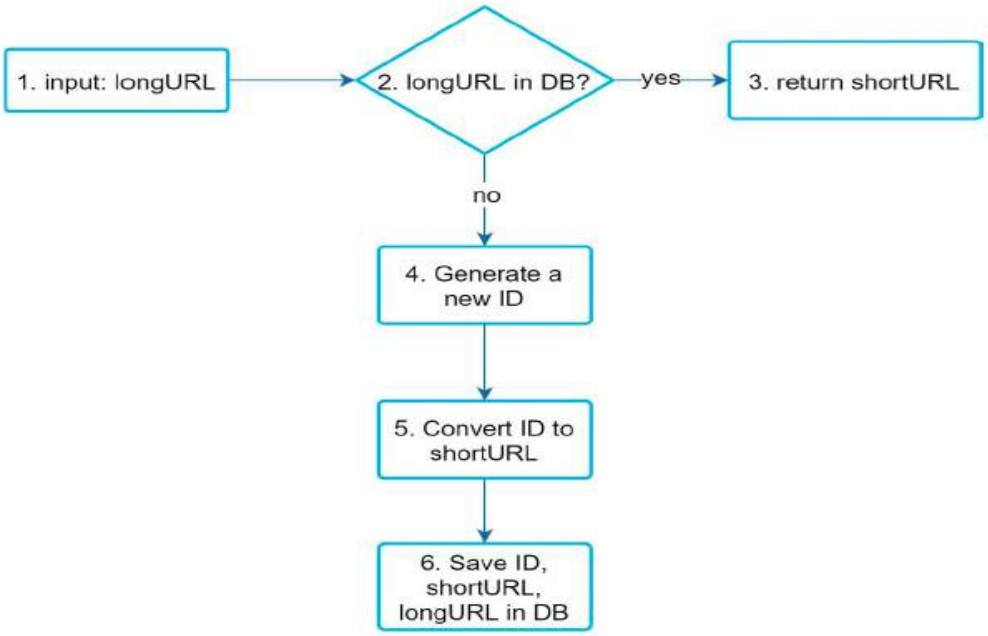



Figure 8-7

1. longURL is the input.
2. The system checks if the longURL is in the database.
3. If it is, it means the longURL was converted to shortURL before. In this case, fetch the shortURL from the database and return it to the client.
4. If not, the longURL is new. A new unique ID (primary key) Is generated by the unique ID generator.
5. Convert the ID to shortURL with base 62 conversion.
6. Create a new database row with the ID, shortURL, and longURL.

To make the flow easier to understand, let us look at a concrete example.

- Assuming the input longURL is: https://en.wikipedia.org/wiki/Systems_design
- Unique ID generator returns ID: 2009215674938.
- Convert the ID to shortURL using the base 62 conversion. ID (2009215674938) is converted to "zn9edcu".
- Save ID, shortURL, and longURL to the database as shown in Table 8-4.

| id | shortURL | longURL |
|---|---|---|
| 2009215674938 | zn9edcu | https://en.wikipedia.org/wiki/Systems_design |

- ***FINAL FLOW***

## URL redirecting deep dive

Figure 8-8 shows the detailed design of the URL redirecting. As there are more reads than writes, <*shortURL*, *longURL*> mapping is stored in a cache to improve performance.

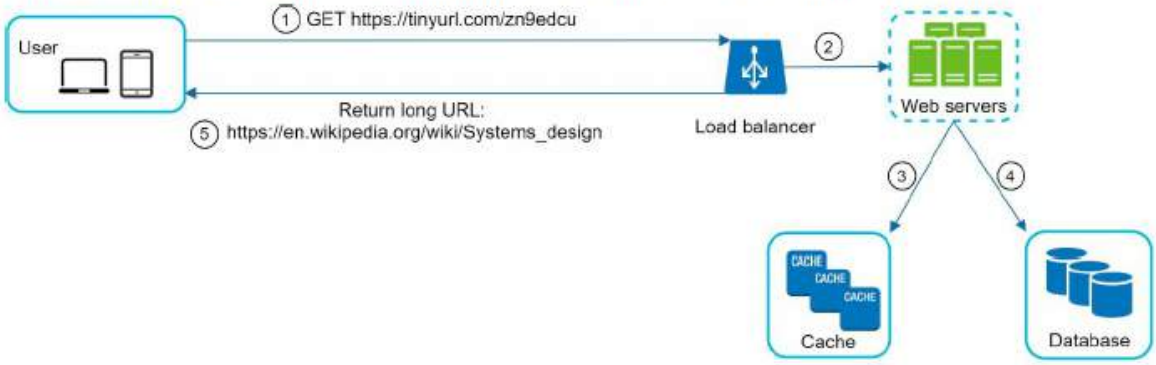


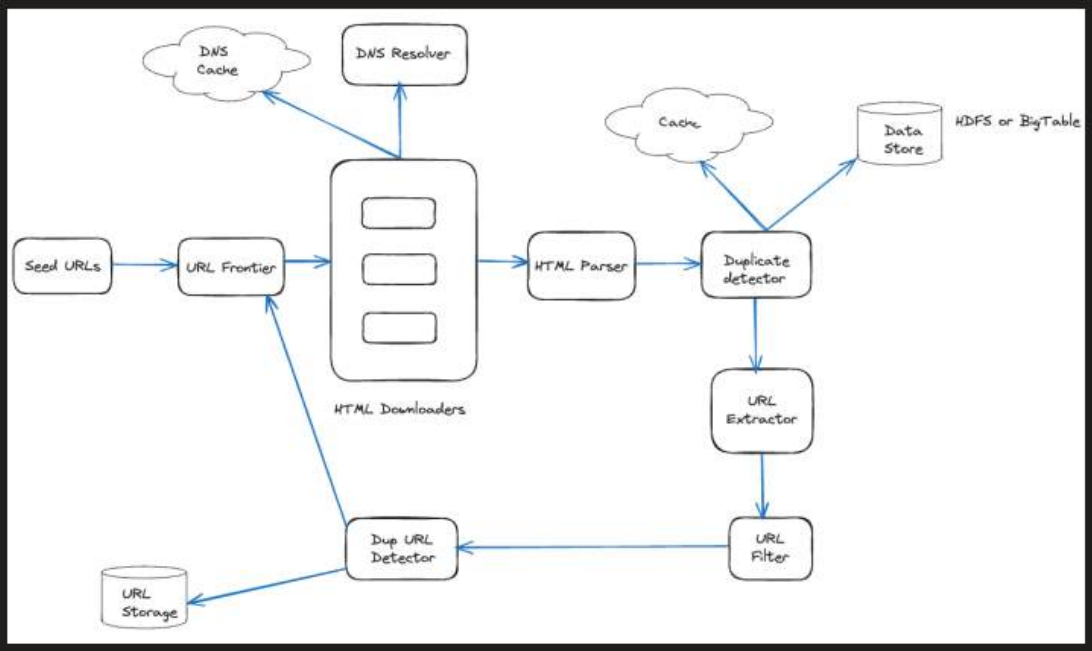

Figure 8-8

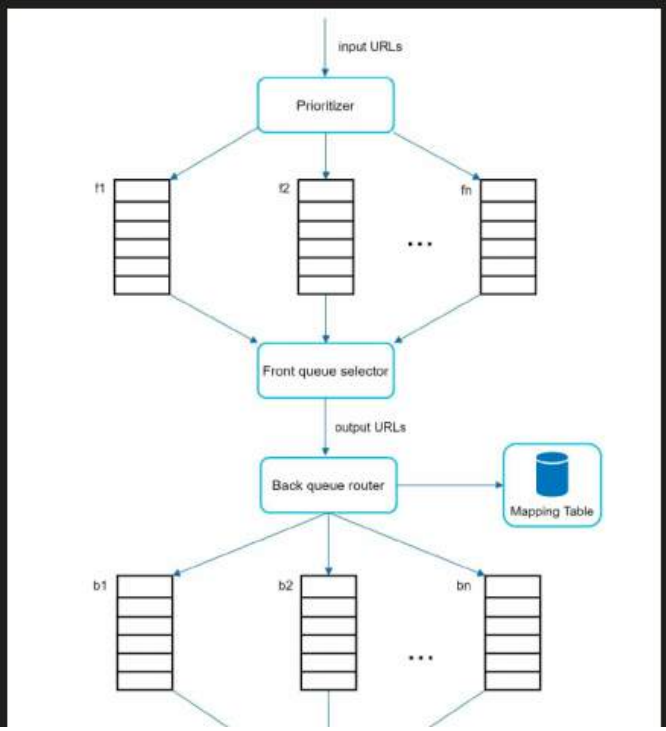
The flow of URL redirecting is summarized as follows:

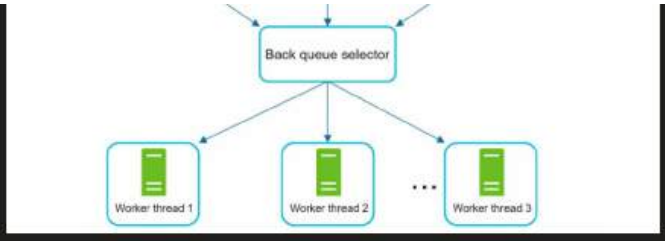
1. A user clicks a short URL link: https://tinyurl.com/zn9edcu
2. The load balancer forwards the request to web servers.
3. If a shortURL is already in the cache, return the longURL directly.
4. If a shortURL is not in the cache, fetch the longURL from the database. If it is not in the database, it is likely a user entered an invalid shortURL.
5. The longURL is returned to the user.

# 6 - WEB CRAWLER

## FINAL DESIGN






## STEP 1: UNDERSTANDING & QUESTIONS

- What is the purpose of the crawler? What are the kind of pages that we intend to crawl?
- How many pages should be crawled?
- What kind of content is included? Just the HTML or images and other content as well?
- Should the links to the web pages inside each of these URLs be included to crawl?
- Should these pages be stored? If yes, for how long ??
- What if there are any duplicate pages?

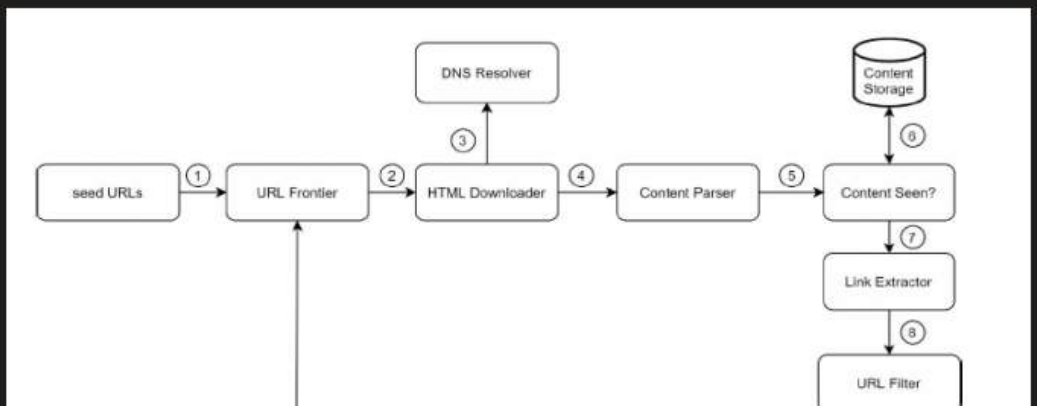
The system should be Robust, Scalable and Polite

## STEP 2: BACK OF THE ENVELOPE ESTIMATION

Assumptions:

**1 billion web pages are downloaded every month**
**QPS = 1 billion/(30 * 10 ^ 5 seconds) ~= 400 per second**
**Peak QPS = 2 * QPS = 800/sec**
**Each page is of size 500 KB**
**1 billion * 500 KB = 500 TB storage /month**
If each page is stored for 5 years, then total storage = 500 TB * 12 months * 5 years = 30 PB for 5 years

## STEP 3: HIGH-LEVEL DESIGN




High-Level design includes these components. Let's talk about each component

***Seed URLs***

They are starting points for the crawling process. We can have it divided into any number of smaller parts based on geography or on topics such as sports, news, shopping etc

***URL Frontier***

This module stores the URLs that need to be downloaded.

***HTML Downloader***

This component deals with downloading the web pages from the internet. The URLs to those pages are provided by the URL Frontier

***DNS Resolver***

To download a webpage, its URL needs to be converted to an IP Address. That job is done by the DNS Resolver. HTML Downloader calls this resolver to get the IP Address

***Content Parser***

After a web page is downloaded, it must be parsed and validated because malformed web pages could provoke problems and waste storage space. Implementing a content parser in a crawl server will slow down the crawling process. Thus, the content parser is a separate component.

***Content Seen?***

This is introduced to avoid any redundancy in the content we are working on. It helps to detect new content previously stored in the system. To compare two HTML documents, we can compare them character by character. However, this method is slow and time-consuming. An efficient way to accomplish this task is to compare the hash values of the two web pages

***Content Storage***

Storage system for storing HTML content. Most of the content is stored on disk because the data set is too big to fit in memory. Popular content is kept in memory to reduce latency.

***URL Extractor***

Parses and extracts links from the HTML pages.

***URL Filter***

It excludes certain content types, file extensions, error links and URLs in "blacklisted" sites.

***URL Seen?***

It is a data structure that keeps track of URLs that are visited before or already in the Frontier. "URL Seen?" helps to avoid adding the same URL multiple times as this can increase server load and cause potential infinite loops.

***URL Storage***

Stores already visited URLs

Step 1: Add seed URLs to the URL Frontier

Step 2: HTML Downloader fetches a list of URLs from URL Frontier.

Step 3: HTML Downloader gets IP addresses of URLs from DNS resolver and starts downloading.

Step 4: Content Parser parses HTML pages and checks if pages are malformed.

Step 5: After content is parsed and validated, it is passed to the "Content Seen?" component.

Step 6: "Content Seen" component checks if a HTML page is already in the storage.

- If it is in the storage, this means the same content in a different URL has already been processed. In this case, the HTML page is discarded.
- If it is not in the storage, the system has not processed the same content before. The content is passed to Link Extractor.

Step 7: Link extractor extracts links from HTML pages.

Step 8: Extracted links are passed to the URL filter.

Step 9: After links are filtered, they are passed to the "URL Seen?" component.

Step 10: "URL Seen" component checks if a URL is already in the storage, if yes, it is processed before, and nothing needs to be done.

Step 11: If a URL has not been processed before, it is added to the URL Frontier.

## STEP 4: DESIGN DEEP DIVE

1. ***DFS vs BFS***

We can think of web as a directed graph and these pages as nodes and the hyperlinks as edges. Now, we have 2 famous traversal algorithms that are DFS and BFS. But, in this use case DFS might not be the best option as we are dealing with the web world and the depth of a page could be too deep. In that case, we will never process the other data. Hence, BFS is the right option for this approach and is implemented using FIFO approach.

There are 2 problems with BFS which are

- Most of the links from the same web page are linked to the same host making the crawler process URLs from same host which is kinda "impolite"

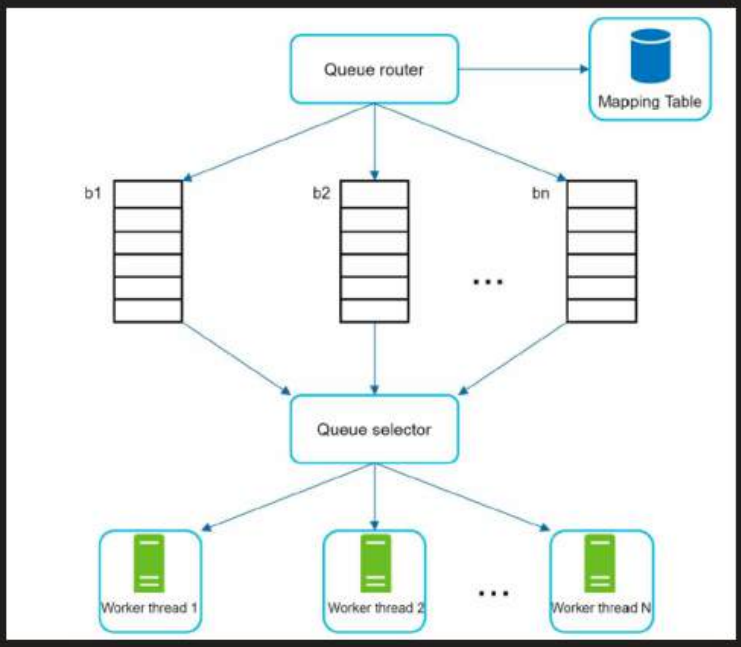
- Standard BFS does not take the priority of a URL into consideration. The web is large and not every page has the same level of quality and importance. Therefore, we may want to prioritize URLs according to their page ranks, web traffic, update frequency, etc.

## 2. ***URL Frontier***

* URL Frontier addresses these issues as it ensures politeness, URL prioritization.

- Politeness

  * The general idea of enforcing politeness is to download one page at a time from the same host. A delay can be added between two download tasks. The politeness constraint is implemented by maintain a mapping from website hostnames to download (worker) threads. Each downloader thread has a separate FIFO queue and only downloads URLs obtained from that queue




• Queue router: It ensures that each queue (b1, b2, … bn) only contains URLs from the same host.
• Mapping table: It maps each host to a queue.

| Host | Queue |
|---|---|
| wikipedia.com | b1 |
| apple.com | b2 |
| ... | ... |
| nike.com | bn |

Table 9-1

- FIFO queues b1, b2 to bn: Each queue contains URLs from the same host.
- Queue selector: Each worker thread is mapped to a FIFO queue, and it only downloads URLs from that queue. The queue selection logic is done by the Queue selector.
- Worker thread 1 to N. A worker thread downloads web pages one by one from the same host. A delay can be added between two download tasks.

- Prioritization
  - We prioritize URLs based on usefulness, which can be measured by PageRank [10], website traffic, update frequency, etc. "Prioritizer" is the component that handles URL prioritization.

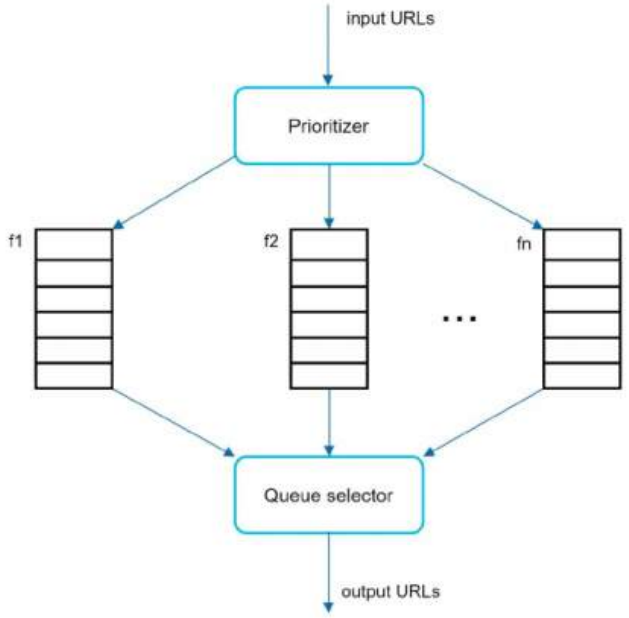
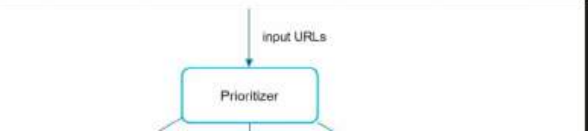
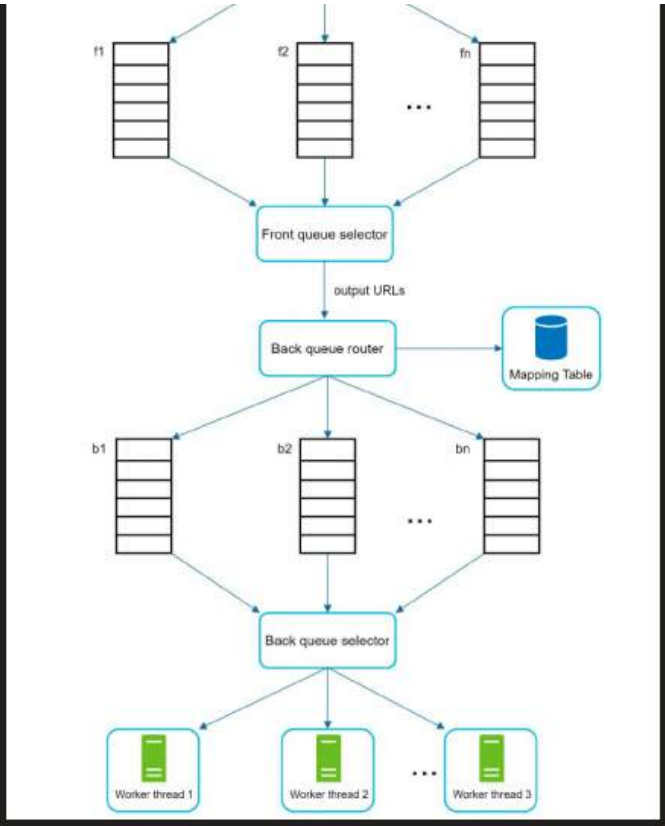



Figure 9-7

- Prioritizer: It takes URLs as input and computes the priorities.
- Queue f1 to fn: Each queue has an assigned priority. Queues with high priority are selected with higher probability.
- Queue selector: Randomly choose a queue with a bias towards queues with higher priority.

* Overall, this is how it's going to look like in the URL Frontier. Front queues take care of the prioritization and the backend queues take care of the politeness






3. ***HTML Downloader***

It downloads the web pages using HTTP protocol. Specifically, it uses the Robots Exclusion Protocol first

**Robots.txt**

Robots.txt, called Robots Exclusion Protocol, is a standard used by websites to communicate with crawlers. It specifies what pages crawlers are allowed to download. Before attempting to crawl a web site, a crawler should check its corresponding robots.txt first and follow its rules.

To avoid repeat downloads of robots.txt file, we cache the results of the file. The file is downloaded and saved to cache periodically. Here is a piece of robots.txt file taken from https://www.amazon.com/robots.txt. Some of the directories like creatorhub are disallowed for Google bot.

4. ***Optimization***

* To achieve high performance, crawl jobs are distributed into multiple servers, and each server runs multiple threads. The URL space is partitioned into smaller pieces; so, each downloader is responsible for a subset of the URLs.

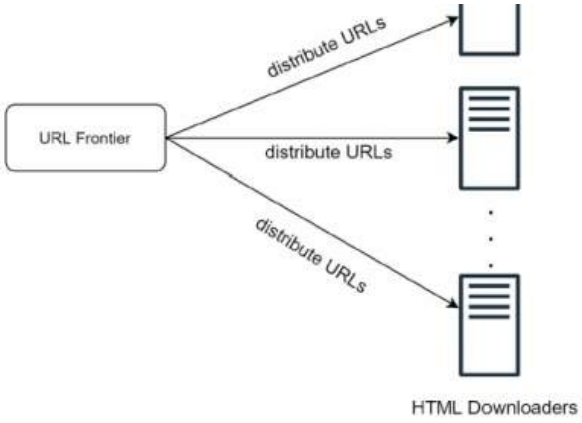



Figure 9-9

**2. Cache DNS Resolver**

DNS Resolver is a bottleneck for crawlers because DNS requests might take time due to the synchronous nature of many DNS interfaces. DNS response time ranges from 10ms to 200ms. Once a request to DNS is carried out by a crawler thread, other threads are blocked until the first request is completed. Maintaining our DNS cache to avoid calling DNS frequently is an effective technique for speed optimization. Our DNS cache keeps the domain name to IP address mapping and is updated periodically by cron jobs.

**3. Locality**

Distribute crawl servers geographically. When crawl servers are closer to website hosts, crawlers experience faster download time. Design locality applies to most of the system components: crawl servers, cache, queue, storage, etc.

**4. Short timeout**

Some web servers respond slowly or may not respond at all. To avoid long wait time, a maximal wait time is specified. If a host does not respond within a predefined time, the crawler will stop the job and crawl some other pages.

## STEP 5: FINAL OPTIMIZATIONS & POTENTIAL Q/A

- Nearly 30% of the web pages are duplicates. Hashes or checksums help to detect duplication
- Spider trap is a web page that causes a crawler in an infinite loop. For instance, an infinite deep directory structure is listed as follows:
  www.spidertrapexample.com/foo/bar/foo/bar/foo/bar/...
  Such spider traps can be avoided by setting a maximal length for URLs.
- With finite storage capacity and crawl resources, an anti-spam component is beneficial in filtering out low quality and spam pages.
- For large scale crawl, hundreds or even thousands of servers are needed to perform download tasks. The key is to keep servers stateless.

SOURCES


- Alex Xu
- https://www.enjoyalgorithms.com/blog/web-crawler

# 7 - NOTIFICATION SYSTEM

- **FINAL DESIGN DIAGRAM**

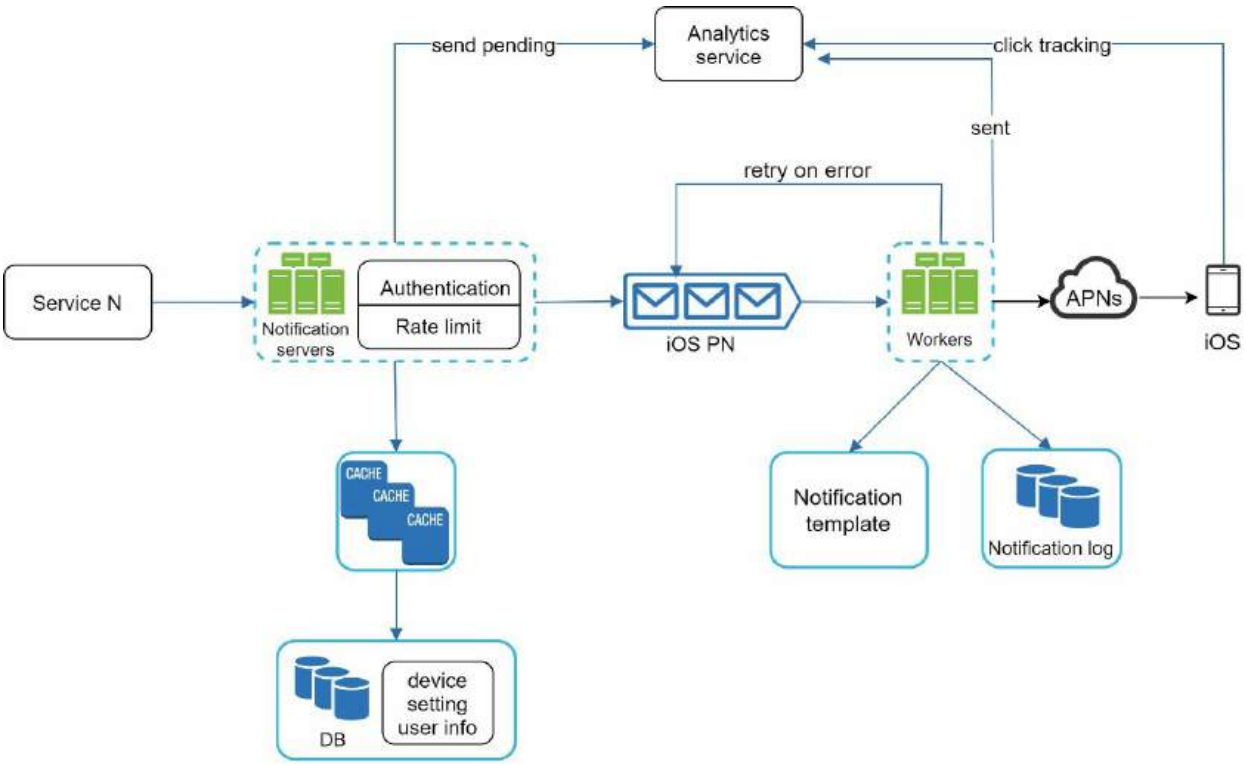
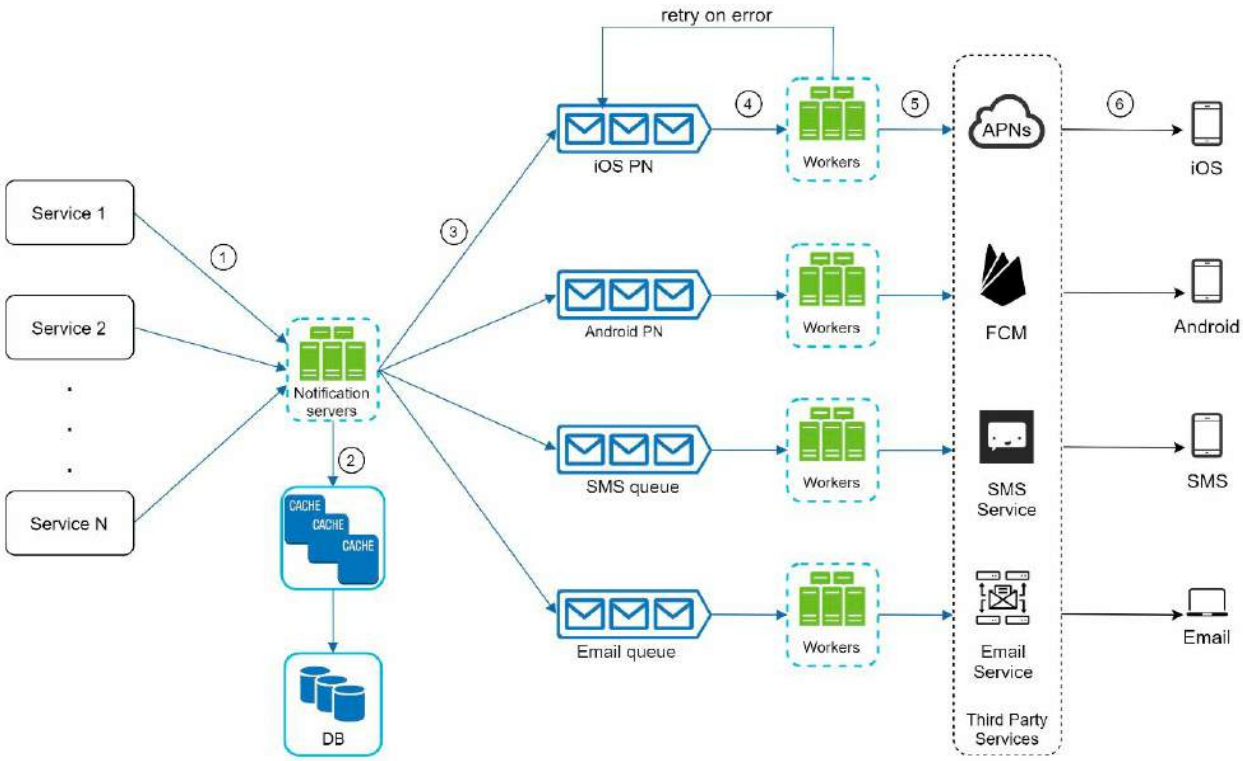



Figure 10-14

-




## STEP 1: UNDERSTANDING & QUESTIONS

- What types of notifications are we supporting?
- Is it real time?
- Supported devices?
- What triggers the applications and can the users opt out ??
- How many notifications per day?
  - 10M mobile push notifications
  - 1M SMS texts
  - 5M emails
-

## STEP 2: USER DATA MODEL

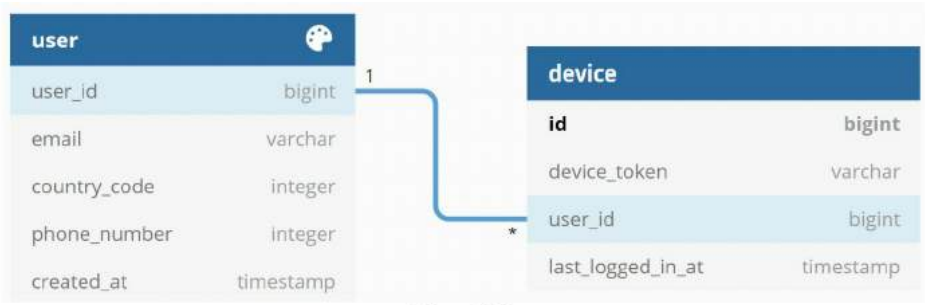
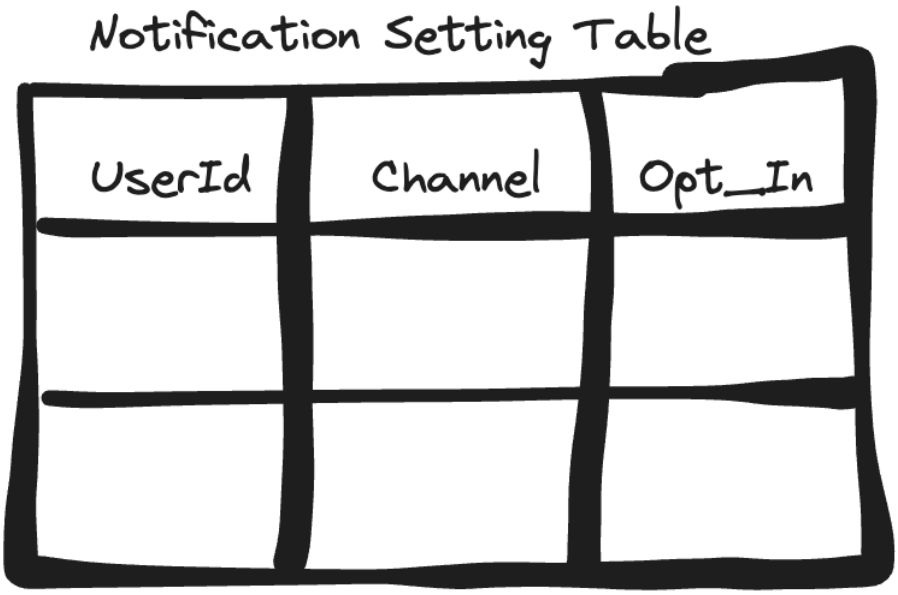
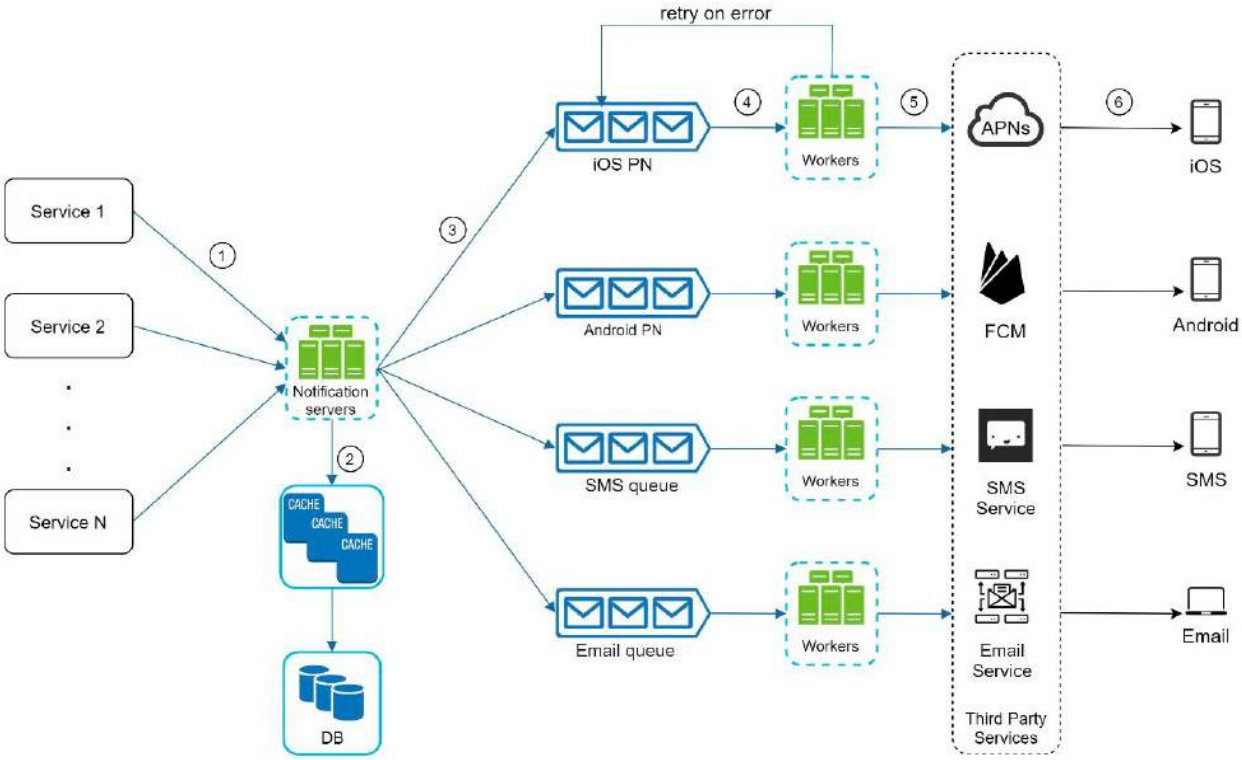



Figure 10-8

- Notification Setting table to store if the user opted for this type of notification




## STEP 3: HIGH-LEVEL DESIGN

- 



- 
  - ***Notification servers***
    - API endpoints to send notifications. Only accessible by clients
    - Carry out basic validations
    - Query the DB or cache for user info etc
  - **API Endpoint**
    - POST https://api.example.com/v/sms/send
- 

```
{
  "to": [
    {
      "user_id": 123456
    }
  ],
  "from": {
    "email": "from_address@example.com"
  },
  "subject": "Hello, World!",
  "content": [
    {
      "type": "text/plain",
      "value": "Hello, World!"
    }
  ]
}
```

- Cache the **User info, device info and notification templates** etc
- DB has info about user, notification settings etc
- Queues for each type of notification
- Workers pick events from these queues and send them to corresponding 3rd party services

## STEP 4: DESIGN DEEP DIVE

- Data cannot be lost. Hence, **retry mechanism is introduced by saving the events in notification log DB**
- **Notification Template**

A large notification system sends out millions of notifications per day, and many of these notifications follow a similar format. Notification templates are introduced to avoid building every notification from scratch. A notification template is a preformatted notification to create your unique notification by customizing parameters, styling, tracking links, etc. Here is

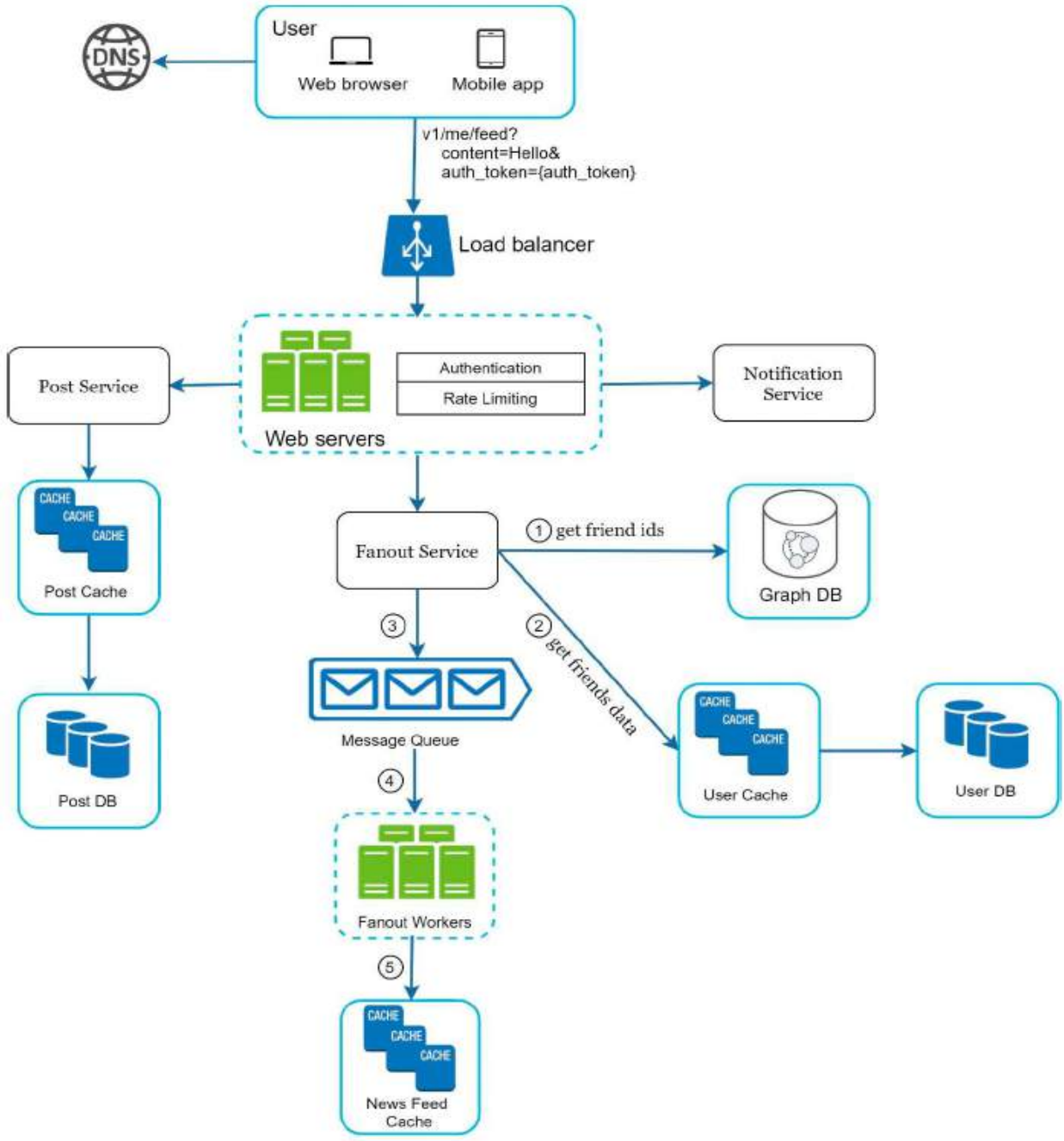
**Events tracking**

Notification metrics, such as open rate, click rate, and engagement are important in understanding customer behaviors. Analytics service implements events tracking. Integration between the notification system and the analytics service is usually required. Figure 10-13 shows an example of events that might be tracked for analytics purposes.

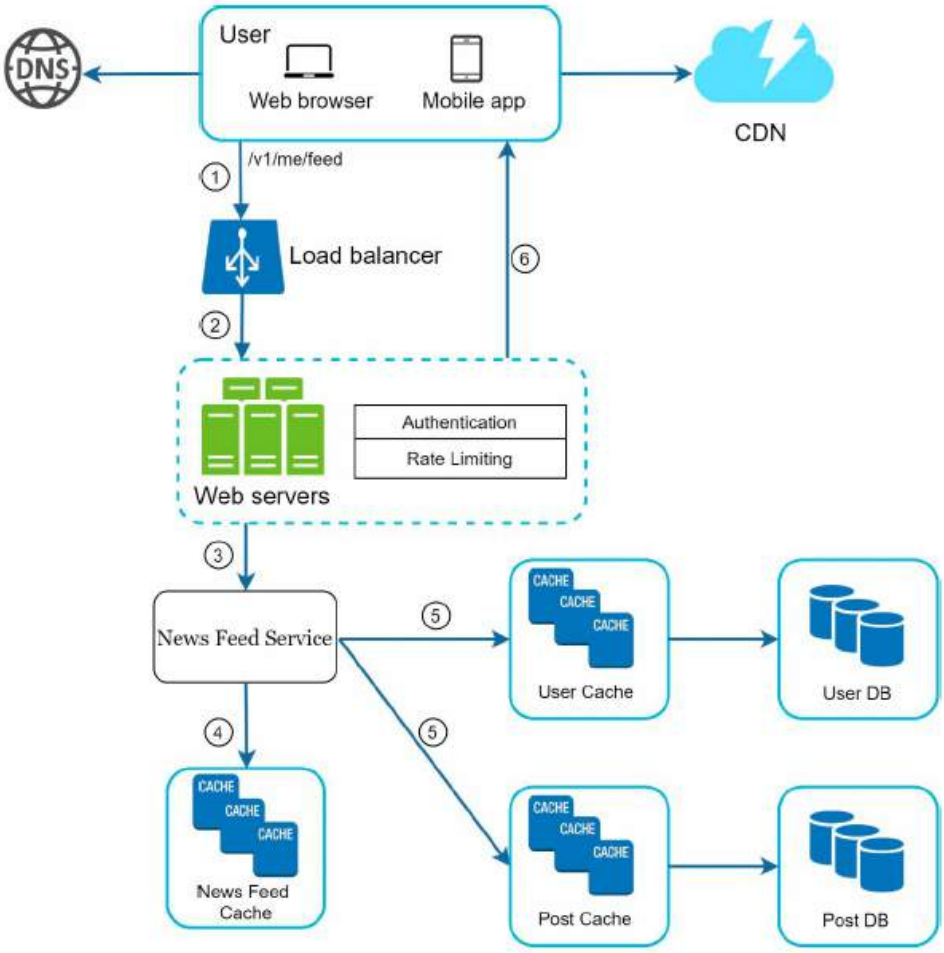
# 8 - NEWS FEED SYSTEM

## FINAL DESIGN

- **NEWS FEED PUBLISHING**




- **NEWS FEED RETRIEVAL**




## STEP 1: UNDERSTANDING & QUESTIONS

- FR
    - Tweet
    - Follow
    - Search
- Non-FR
    - Read heavy
    - Fast rendering
    - Very low latency
    - Always available
    - Lag is okay. In terms that consistency can be achieved eventually

## STEP 2: BACK OF THE ENVELOPE ESTIMATION

- 150 million daily users
- 350 million monthly active users
- Around 1.5 billion accounts
- Roughly 500 million tweets per day
    - Around 5000 tweets per sec
    - Peak load could be double of that

## APIs

**Feed publishing API**

To publish a post, a HTTP POST request will be sent to the serve

*POST /v1/me/feed*

Params:

- content: content is the text of the post.
- auth_token: it is used to authenticate API requests.

**Newsfeed retrieval API**

The API to retrieve news feed is shown below:

*GET /v1/me/feed*

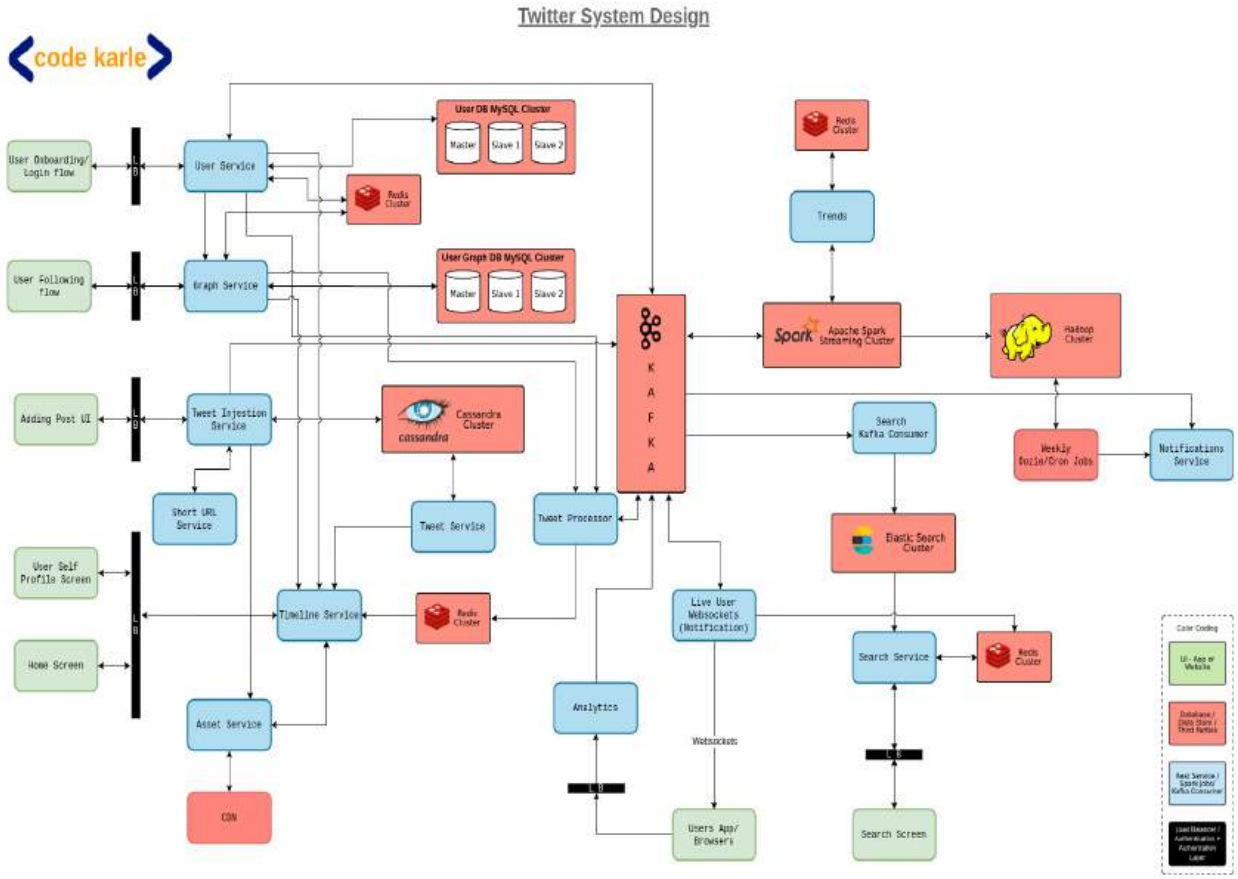
Params:

- auth_token: it is used to authenticate API requests.

## STEP 3: HIGH LEVEL DESIGN

- **MySQL for users and user relations since they are structured**
- **Cassandra for the posts**
- **S3 for any images/videos**
- Different types of users such as
    - Live
    - Active
    - Passive
    - Famous users

## STEP 4: DESIGN DEEP DIVE




***ONBOARDING FLOW***

- User Service takes care of the user related info and stores it in the MySQL Database
- User details are cached in Redis so that they can be easily sent back whenever user details are requested

***USER FOLLOWS FLOW***

- Graph service takes care of the user follow related queries
- It can cache the information as the follow and unfollow don't change for every 2 seconds or so
- It can cache who are the followers of a particular user, and who is the user following.

For Live Users, Live User websockets can be used which keep open session with the users and maintain the info. Now based on the user's interaction with this service we can also track for

how long the users are online, and when the interaction stops we can conclude that the user is not live anymore. When the user goes offline, through the websocket service an event will be fired to Kafka which will further interact with user service and save the last active time of the user in Redis

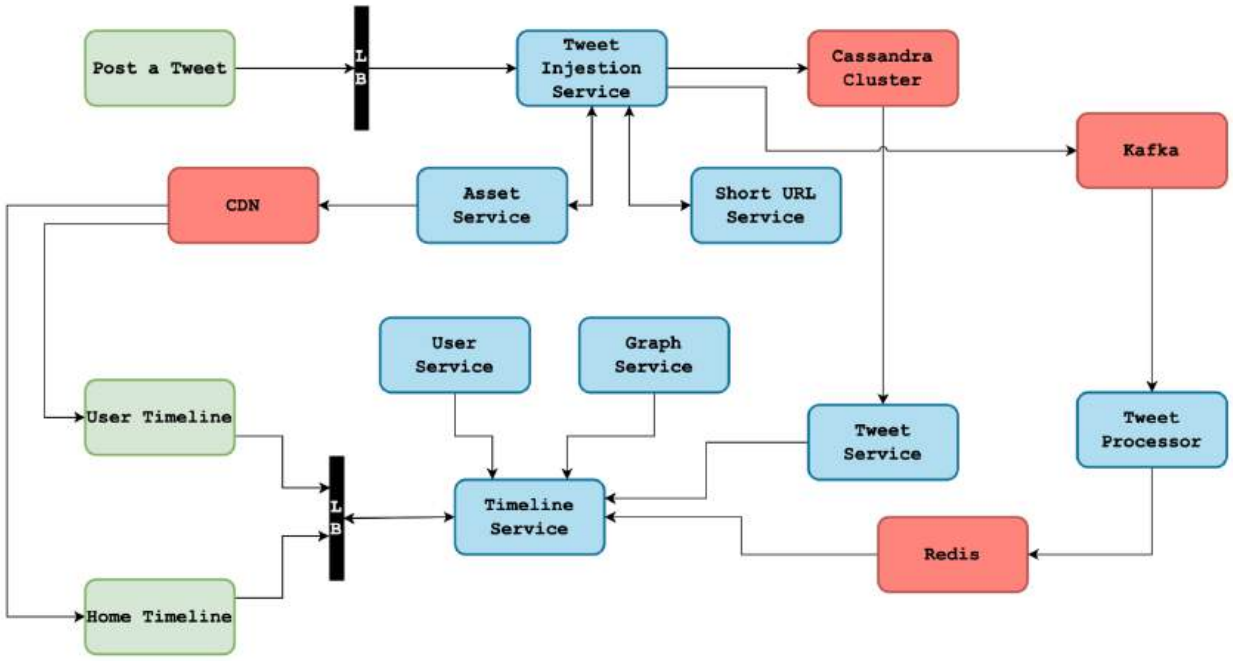
***TWEET FLOW***

- When a user posts, it first goes to the tweet ingestion service and then next steps happen
- Tweet could also contain images or URLs in it. **They are handled by Asset Service and Short URL service** respectively which are invoked from the Tweet Ingestion Service
- Now, coming to the actual steps, whenever a tweet is posted, the tweet ingestion service stores them in Cassandra. Cassandra because of its efficiency with large number of writes
- As soon as a tweet is posted, the tweet ingestion service **will fire an event to Kafka saying a tweet id was posted** by so and so user id. Now on top of our Cassandra, **sits a Tweet service that will expose APIs to get tweets by tweet id or user id.**
- Now, coming to the reading from user, his/her home timeline could be loaded in 2 ways
    - **Fanout on Write**
        - With this, the feed is precomputed during the tweet write up itself and stored
        - In this way, feed is generated in real time and can be retrieved easily during read as it is precomputed
    - **Fanout on Read**
        - With this, feed is computed whenever a read request is received.
        - Less usage of computing resources but fetching the news feed is going to be very slow
    - If we make all the queries at runtime before displaying the timeline it will slow down the rendering. So we cache the user's timeline instead. **We will pre-calculate the timeline of active users and cache it in a Redis**, so an active user can instantaneously see their timeline.
    - When a tweet is posted, an event is fired to Kafka. **Kafka communicates the same to the tweet processor** and creates the timeline for all the users that need to be notified of this recent tweet and cache it. To find out the followers that need to be notified of this change **tweet service interacts with the graph service**

- Now, we have only cached the timelines for active users. What happens when a passive user, say P1, logs in to the system? This is where the **Timeline Service** comes in. The request will reach the timeline service, timeline service will interact with the user service to identify if P1 is an active user or a passive user. Now since P1 is a passive user, its timeline is not cached in Redis. Now the timeline service will talk to the graph service to find a list of users that P1 follows, then queries the tweet service to fetch tweets of all those users, caches them in the Redis, and responds back to the client.

- For the famous users, this approach of caching feed at write is not efficient as if a celebrity with 100 Million followers tweets something, precomputing the feed for all the followers is very compute intensive

- Redis cache will only cache the tweets from non-famous users in the precalculated timelines. Timeline service knows that Redis only stores tweets from normal users. It interacts with graph service to get a list of famous users followed by our current user, say U1, and it fetches their tweets from the tweet service. It will then update these tweets in Redis and add a timestamp indicating when the timeline was last updated. When the next request comes from U1, it checks if the timestamp in Redis against U1 is from a few minutes back. If so, it will query the tweet service again. But if the timestamp is fairly recent, Redis will directly respond back to the app.



Now, this looks like a fairly efficient system, but there are some bottlenecks. Like Cassandra - which will be under a huge load, Redis - which needs to scale efficiently as it is stored completely in RAM, and Kafka - which again will receive crazy amounts of events. So we need to make sure that these components are horizontally scalable and in case of Redis, don't store old data that just unnecessarily uses up memory.

- 

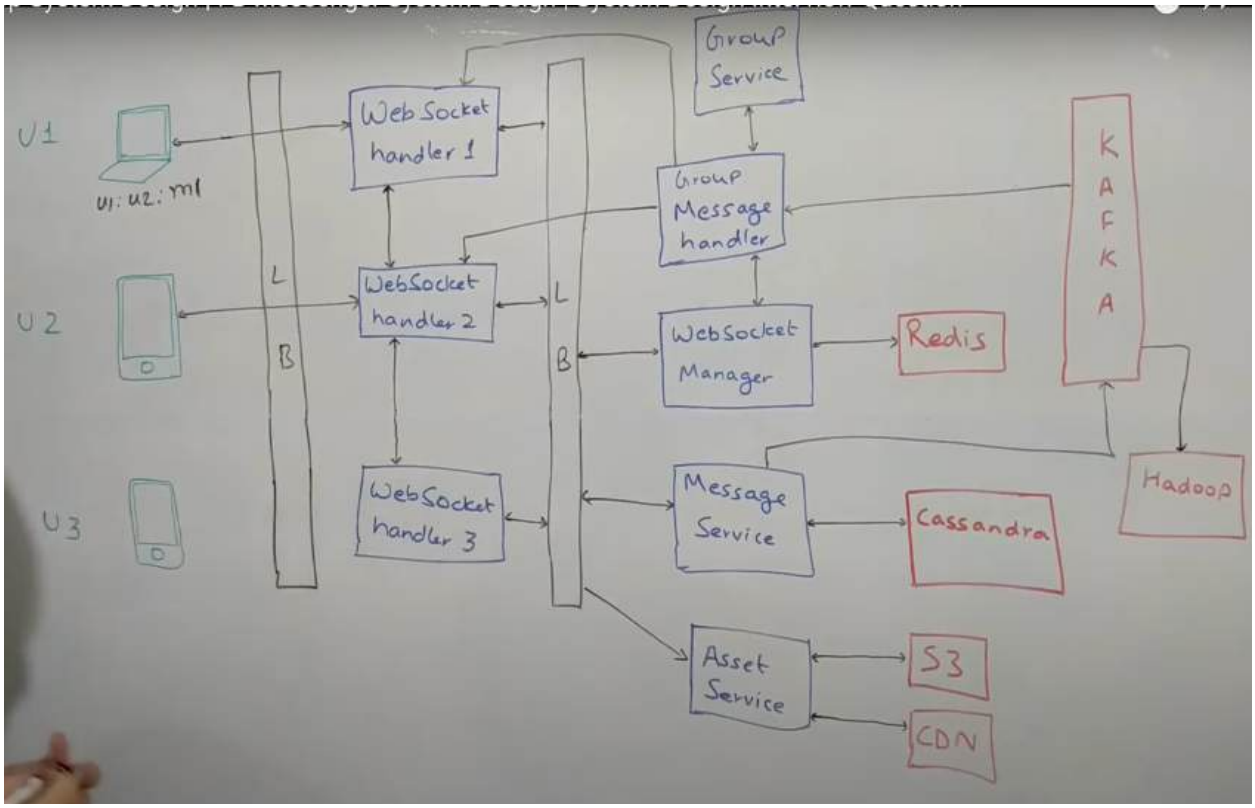
***SEARCH AND ANALYTICS***

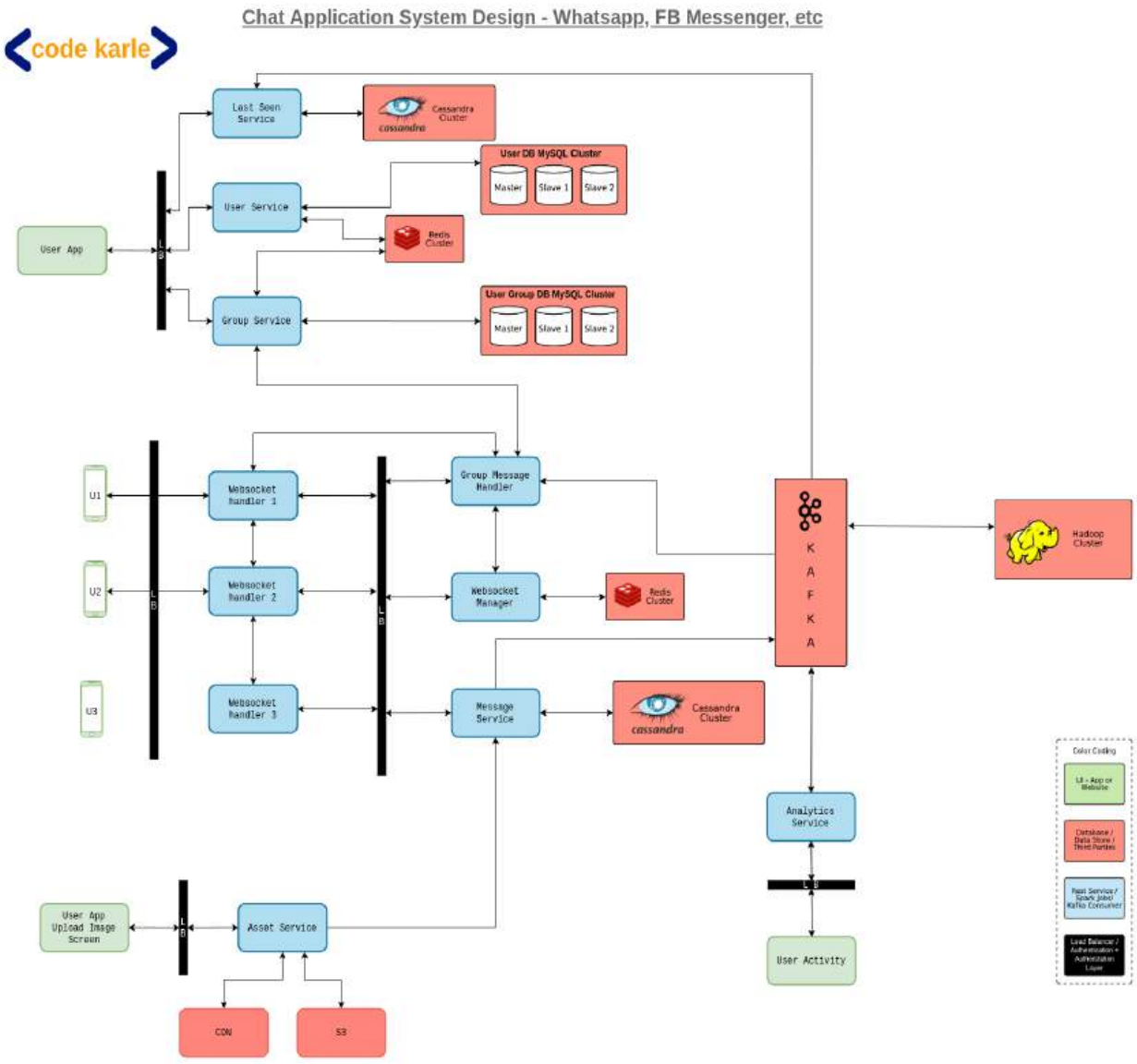
Remember the tweet **ingestion service** we discussed in the previous section? When a tweet is added to the system, it fires an event to Kafka. A search consumer listening to Kafka stores all these incoming tweets into an **Elasticsearch** database. Now when the user searches a string in **Search UI**, which talks to a **Search Service**. Search service will talk to elastic search, fetch the results, and respond back to the user.

- 

# 9 - CHAT APPLICATION

## FINAL DESIGN






## STEP 1: UNDERSTANDING & QUESTIONS

- FR
    - Support one-on-one chats
    - Support group chats
    - Not just text but also files
    - Indicate read receipts
    - Last seen

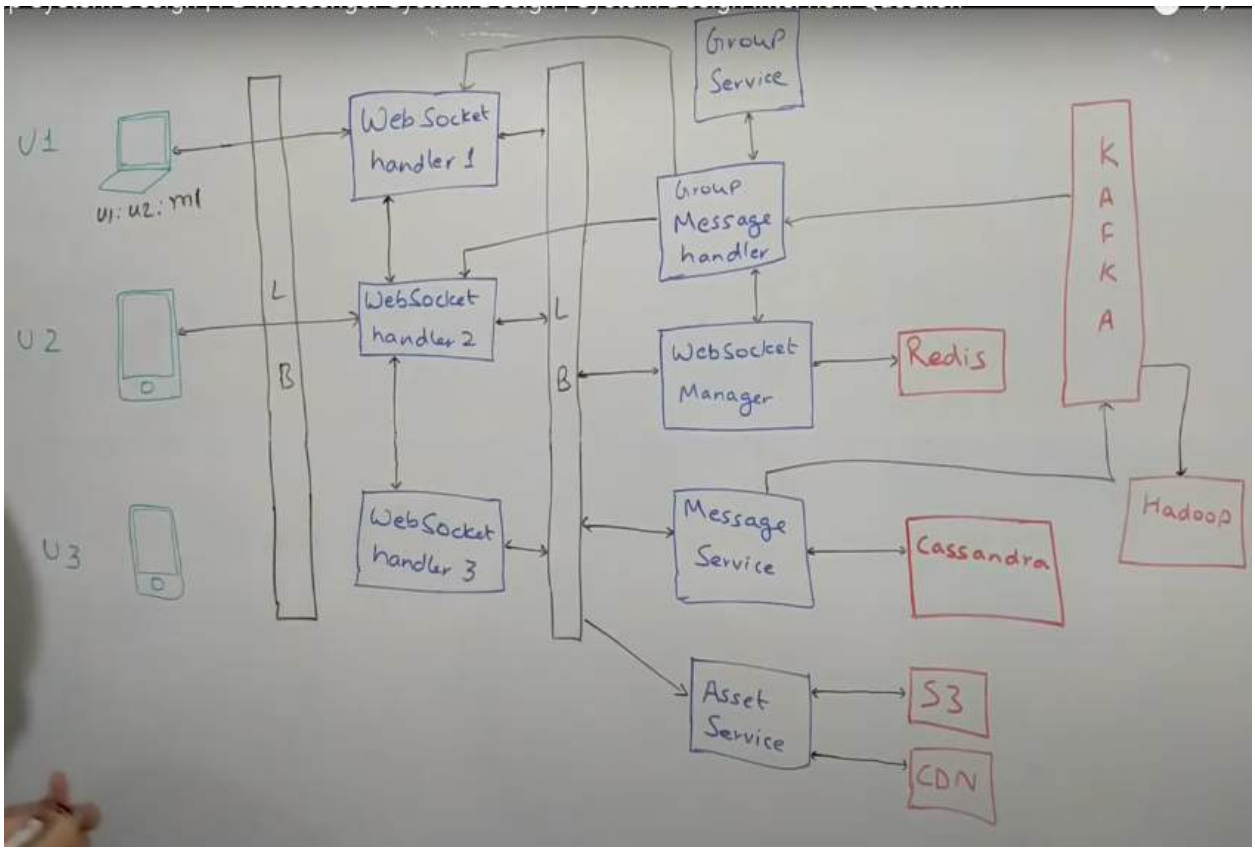
- Non-FR
    - Always available
    - Highly scalable
    - Low latency

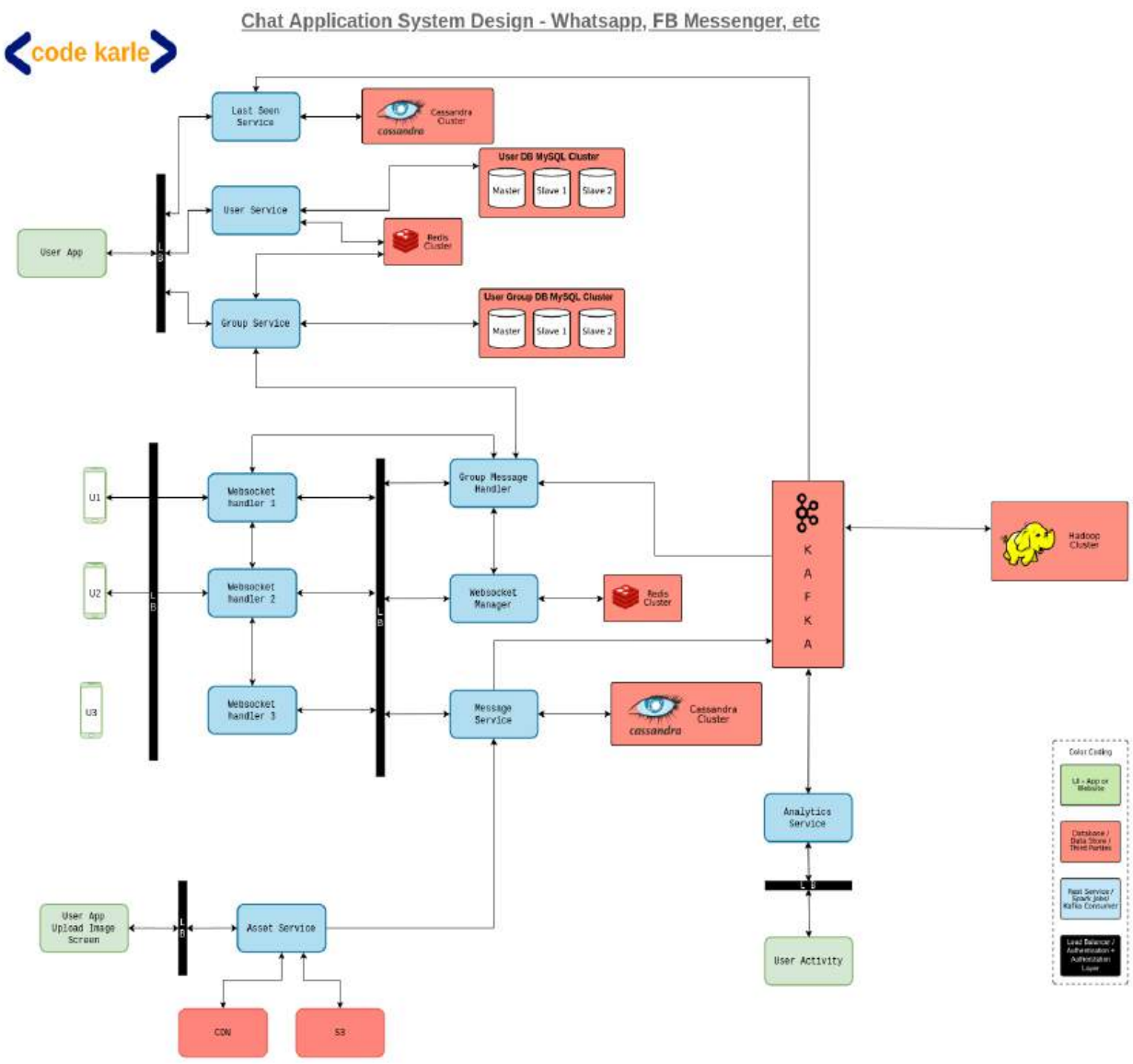
## STEP 2: BACK OF THE ENVELOPE ESTIMATION

- Approx 2 billion users
- Nearly 1.5 billion daily active users
- Around 100 billion msgs per day are exchanged
- 100 billion * 100 bytes per msg = 10 TB per day
- Besides chat messages, we also have media files, which take more than 100 Bytes per message.
- Moreover, we also have to store users' information and messages' metadata—for example, time stamp, ID, and so on. Along the way, we also need encryption and decryption for secure communication. Therefore, we would also need to store encryption keys and relevant metadata. So, to be precise, **we need more than 300 TB per month**
- 10 TB/day means 900 MB/Sec is the bandwidth estimation

## STEP 3: HIGH-LEVEL DESIGN

- First step is to understand how the communication takes place. Three ways you can think of are polling, long polling and web sockets
- Web sockets are more suitable because it's 2-way communication. Polling and long polling are not efficient enough
- Each user is connected to a Web Socket Handler (WSH). Each WSH can accommodate a lot of users. It might have 65k ports and nearly 60k ports can be used for users
- Each WSH is connected to a Web Socker Manager (WSM)
- WSM is a repository of information about which web socket handlers are connected to which users. **It sits on top of Redis**
    - Which user is connected to which web socket handler
    - What all users are connected to a web socket handler
- All these WSH's also contact with MessageService whenever a msg is received. It is more like a repository of all messages in the system. **MessageService sits on top of Cassandra**
- Message service needs to build on top of a data store that can handle ever-increasing data


U1
U2
U3
u1: u2: m1
L
B
WebSocket handler 1
WebSocket handler 2
WebSocket handler 3
Group Service
Group Message handler
WebSocket Manager
Message Service
Asset Service
Redis
KAFKA
Hadoop
Cassandra
S3
CDN




- When a User U1 wants to send msg to User U2, it sends that to WSH1. WSH1 talks to WSM and gets info about which server (WSH2 in this case) user U2 is connected to and then sends the msg to WSH2.
- At the same time, MessageService is also called to store this message in database.
- WSH2 also communicates back whether the msg has been delivered or seen to U2. In case, U1 went offline before receiving this update, then WSH2 informs the same to MessageService and the respective status is stored against this message ID in Cassandra
- WSH2 also communicates through WSM only, as it needs information about the opposite server.
    - Potential Question

        - Making so many calls to WSM every time could add a lot of latency. How to avoid that?
        - Ans: Caching the info about which users are connected to which servers can solve this but we should be careful about the cache time. Since the servers spread across globe, there could be connection issues. We don't want to send messages to wrong servers based on outdated information

## Local Caching

Let's say when U1 sent the message to U3 it was offline i.e. not connected to any web socket handler. In this case, the message will be stored in a local database on the device. As soon as the device gets connected to the network and a web socket handler, it will pull all the messages from the local database and send them.

- 

- ***GROUP CHATS***
    - Web socket handlers won't keep track of groups, it just tracks active users.
    - So, when U1 wants to send a message to G1, WSH1 gets in touch with Message service, that U1 wants to send a message to G1, and it gets stored in Cassandra as M3.
    - **Message service will now communicate with Kafka. M3 gets saved in Kafka with an instruction that it has to be sent to G1**.
    - Now Kafka will interact with something known as **Group Message Handler**. The group message handler listens to Kafka and sends out all group messages.
    - There is something called **Group Service** which keeps track of all the members in all groups. Group message handler talks to Group service to find out all the users in G1 and follows the same process as a web socket handler and delivers the message to all users individually.
    - Details could be fetched from the MySQL db and ofcourse this info could be cached using redis

- ***ASSET DELIVERY***
    - Suppose U3 is sending an image to U2. This will happen in two steps.
        - U3 will upload the image to a server and get the image id
        - Then it will send the image ID to U2 and U2 can search and download the image

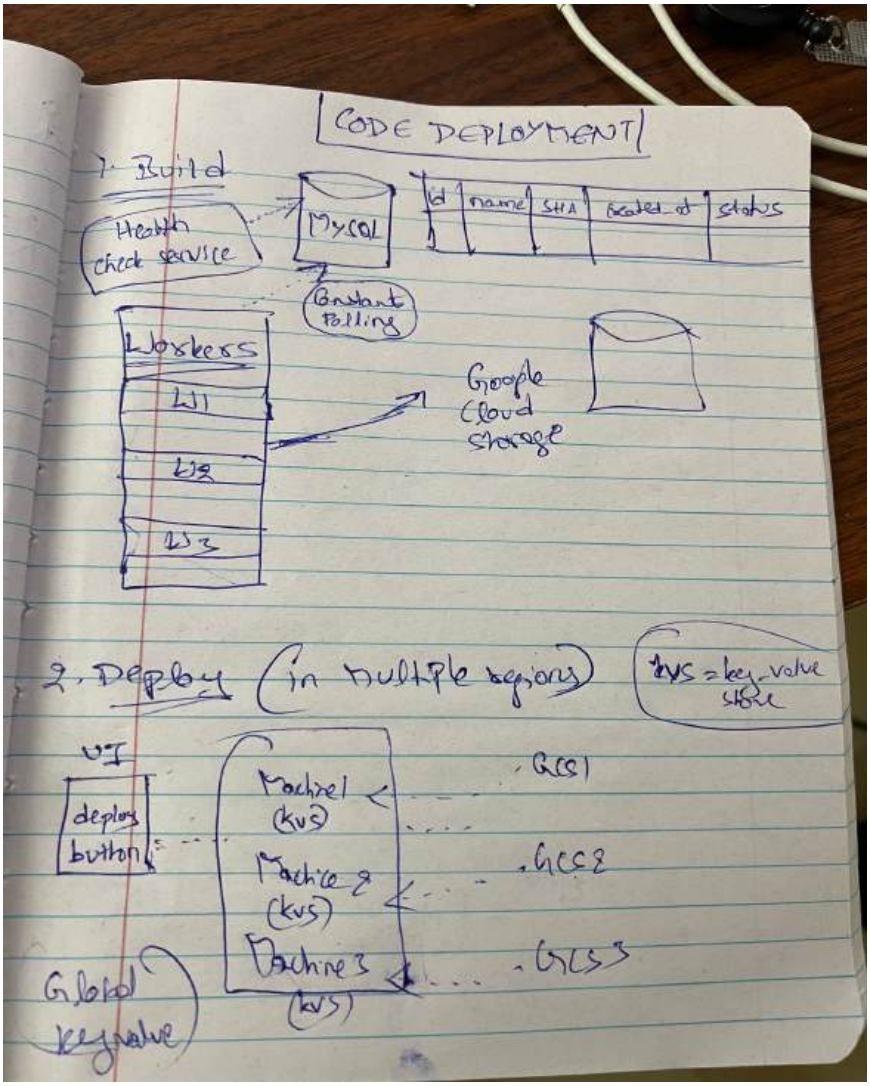
      from the server.
- The image will be compressed on the device side and sent to an Asset Service through the Load Balancer and asset service will store the content on S3.

- In case of popular images which can be forwarded by millions of users, before uploading that image to asset service, U3 will send a hash of the image and if the hash is already there in the asset service, the content will not be uploaded again, but the id against the same image/content will be sent to U2

- **User Service** stores user-related information like name, id, profile picture, preferences, etc, and usually, this is stored in MySQL database. This information is also cached in Redis.
- ***Last Seen***
    - User events - things like when the user opens or closes or does something else in the app. These events will be used to track the last seen time.

# 10 - CODE DEPLOYMENT SYSTEM

## FINAL DESIGN




## STEP 1: UNDERSTANDING & QUESTIONS

- Involves building a code and deploying it
- What kind of system are we handling? System that build binaries and deploys them globally?
- Are the machines located all over the world or are they located only in some parts of the world?
    - 5-10 regions where 100K machines are present
    - Internal system
- How should the availability be for this system?
    - Should be available. 2 to 3 nines of availability
- Entire deployment should take place in 30 mins and the **largest file could be 10 GB**

- We want to persist all the previous build and deployments

## STEP 2: RESOURCE ESTIMATION

- There would be approx **5000** deployments everyday
- We can assume that each job is going to take **15 minutes to create a build, it means each worker can create 4 builds an hour and roughly 100 builds**
- Therefore, **5000/100 = 50. We'd need 50 workers per day** on average

## STEP 3: DESIGN DEEP DIVE

### ***PART 1 : BUILD***

- These builds can be performed in FIFO basis. That is we need the **code changes that are earliest to be built and deployed first**
- Workers can pick these jobs one after the other
- We can maintain a **MySQL database** which has a table with these jobs. From this table, the worker nodes can pick the jobs and operate them.
- Once the workers pick a job and create a binary, **these binaries can be stored in BLOB storage** such as **Google Cloud Storage**
- Columns in the table can look roughly like below

| ID | NAME | SHA | CREATED_AT | STATUS |
|---|---|---|---|---|
| | | | | |

- ID is nothing but the job id
- NAME could be the name of the binary we are creating
- SHA is the commit SHA we are using for this build
- CREATED_AT is the timestamp at which this job is created
- STATUS is the status of the job
    - Could be a Enum of ENQUEUED, RUNNING, FAIL, SUCCESS
- Since this is SQL DB which has ACID properties, workers can concurrently work on the jobs where each of them pick the earliest job
- Below are the rough queries we could use
    - SELECT * from JOBS where status = "ENQUEUED" ORDER by CREATED_AT ASC LIMIT 1;
    - UPDATE JOBS SET STATUS = "RUNNING" WHERE ID = "OUR_ID"

- One enhancement we could do here is **index the CREATED_AT & STATUS columns as we are going to use them a lot**
- Each worker consistently performs polling on the queue to check if any job is in "ENQUEUED" state so that it can work on that job.
- **How do you handle when a worker crashes midway the job processing? Will that job not be in RUNNING state forever?**
    - We can have a Health Check Service which can check the last heart beat time to a job in running state. If the last heart beat column is same for very long, then we can assume that the job is abruptly left. Add extra column in table

| ID | NAME | SHA | CREATED_AT | STATUS | LAST_HEARTBEAT |
|---|---|---|---|---|---|
| | | | | | |

- Status in the tables is updated to SUCCESS only after the binary is successfully stored in the GCS
- **We have 5-10 regions. Having only one GCS is a good solution? What do you propose?**
    - Have the regional buckets or storages where the binary can be replicated aysnchronously from the main GCS
- We can have a service check that the build is deployed in all regions and only then the deployment should be performed

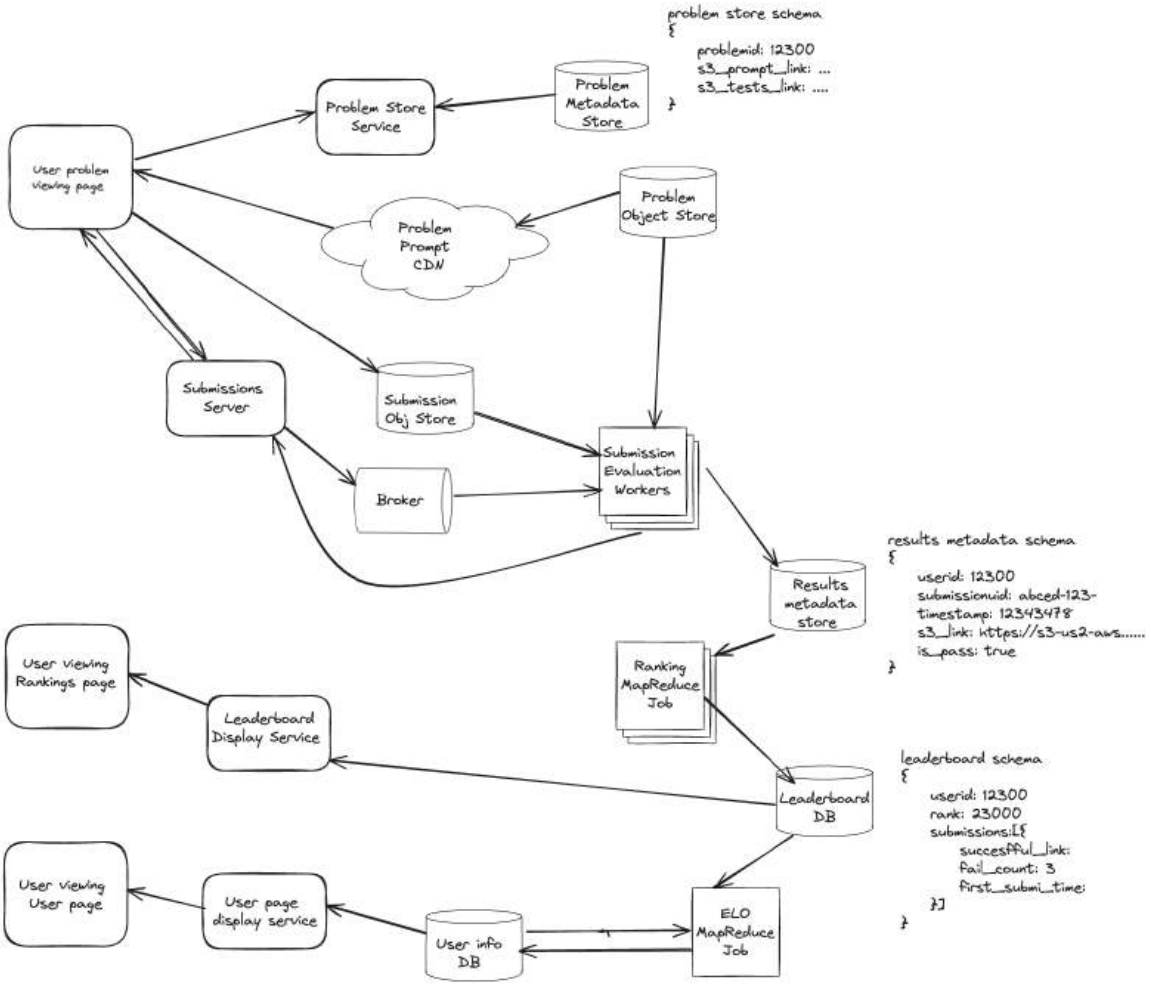
***PART 2: DEPLOYMENT***

- Downloading a binary of 10 GB and then deploying it to hundreds of thousands machines one at a time in a single region is not practical. **This is where Peer-to-Peer Network can be really helpful**. Through this Peer-to-Peer n/w all the machines in a single region can download this file to all the machines
- Now, for the Deployment assuming that a button click in a UI triggers the deployment process following steps are involved in it
    - Click on Deploy button
    - There's a global key value which contains the current build version that needs to be downloaded. Example : {"current build" : "B1"}
    - Now, in every region there is a local Key-Value Store (KVS) which polls this global Key Value store constantly and checks if there is any change in the key, current build.

- Depending upon the value in local KVS, the machines can download the binaries from GCS provided the already downloaded binary is different to whatever is mentioned in the local KVS

# 11 - ONLINE JUDGE

## FINAL DESIGN




## STEP 1: UNDERSTANDING & QUESTIONS

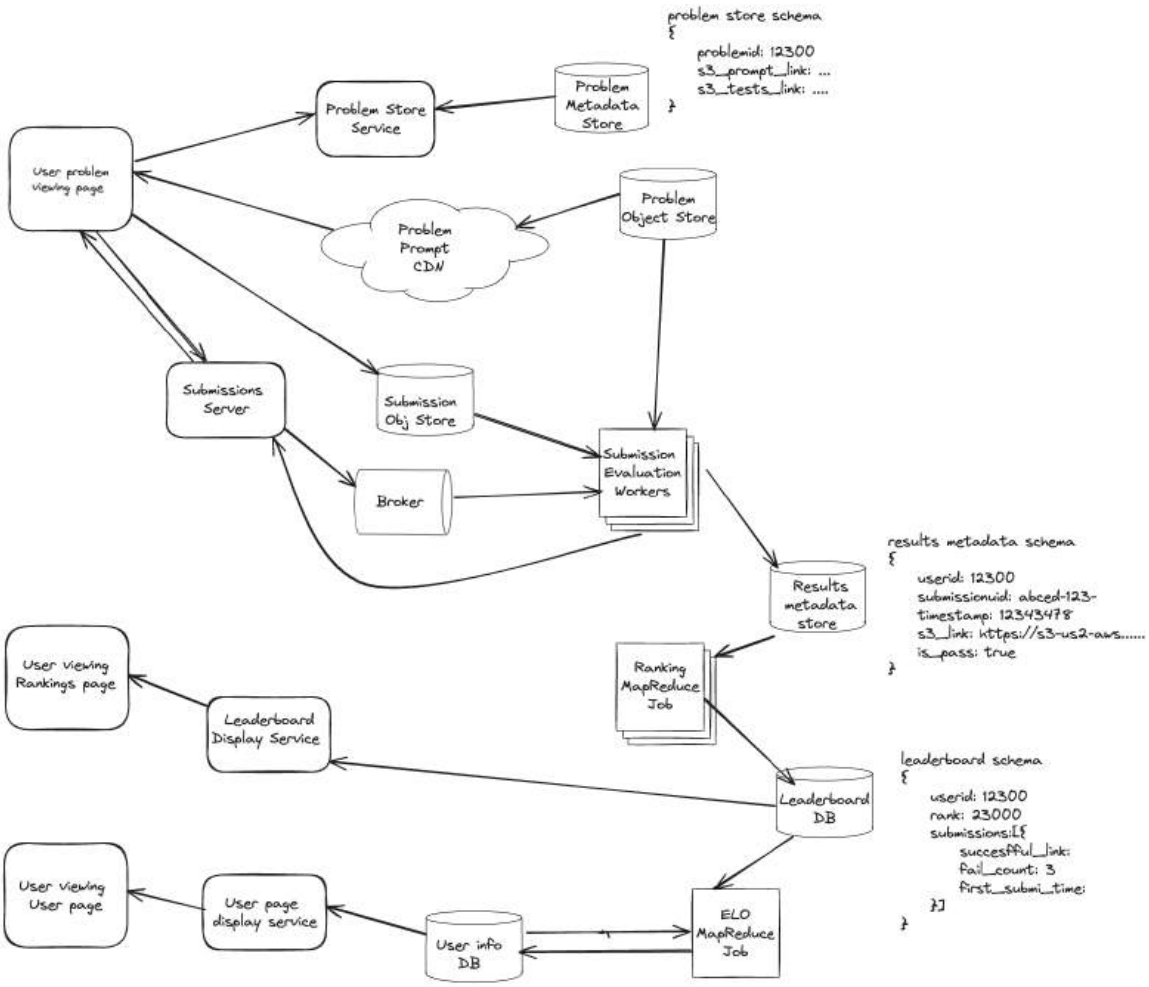
- FR

    - User gets the problem and then submits the solution
    - User would be doing it in web browser
    - Code will be executed on the server side
- NON-FR
    - Scalability
    - Availability
    - Security is important to avoid malicious code

## STEP 2: CAPACITY ESTIMATION

- Contest is 90 minutes long
- No.of contestants?
    - 20,000
- 6 submissions each on avg
- Each submission 2KB size
- Each submission takes 5 seconds of processing
- Total submissions
    - 20K * 6 =  120,000
    - 90 minutes = 5400 seconds
    - Therefore, appox 22 submissions per second
- Bandwidth
    - 22 TPS * 2 KB = 44 KB per second
- 22 submissions * 5 sec = 110 submission running concurrently
- Could be handled by 10-11 servers

## STEP 3: HIGH-LEVEL DESIGN




## STEP 4: DESIGN DEEP DIVE

- ***Problem Viewing Page***
    - User requests for problem. Problem Store service gets the metadata of the problem and eventually gets the problem images if any from the S3 link
    - Since, users are distributed and problem statement is static, it could be stored in **CDN**
    - Once the submit is clicked, it goes to submissions server. **Submission server sends a presigned url to user and then the submission can be stored in object store**. Kind of S3. Submissions server also generates an event for this and places it in Queue. We have submissions valuation workers constantly polling this queue who do the processing. They get the solution from the submissions store and then the test cases from the problem object store

- For the result, **Websocket to send server sent event back to client**
- If the submission is successful, then it places it in results store and also send the information to submissions server who will communicate this to the user.
- We can have a Map Reduce run on this Results Store DB and decide the rankings for the user and place it in the Leaderboard DB

- ***User Contest Rankings Page***
  - When user visits this page, It calls the LeaderBoard service which fetches the results from the LeaderBoard DB
  - These could probably be cached as well
  - On top of LeaderBoard DB, we have another Map Reduce which works for the global ranking of the User
  - These results are again placed in User info DB

- ***User Page***
  - This calls the User page display service which fetches the info from the User info DBCassandra could be used for the Results DB and the Leaderboard DB as it has more reads

# 12 - NEARBY FRIENDS

## FINAL DESIGN

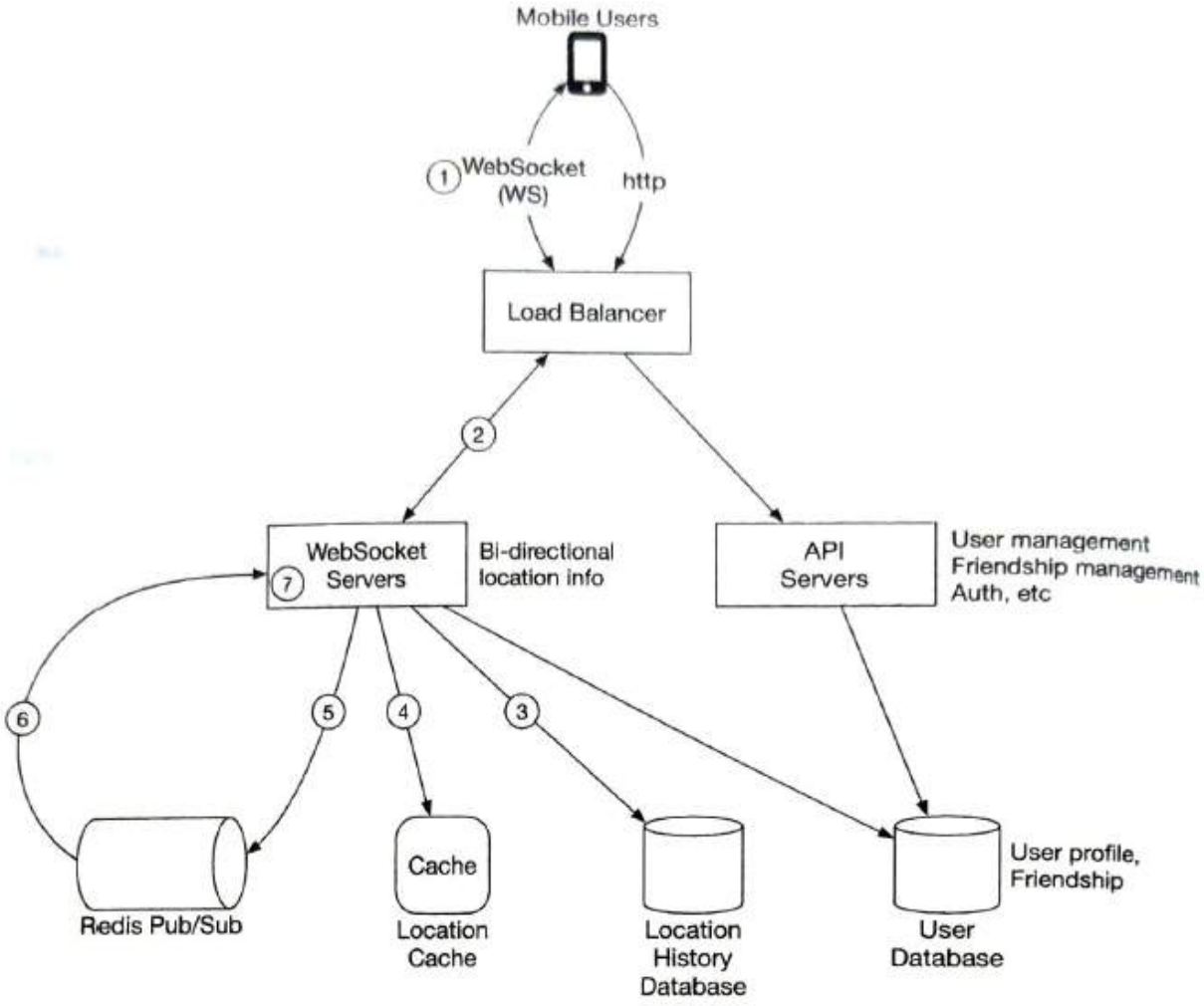



Figure 2.7: Periodic location update

1. The mobile client sends a location update to the load balancer.
2. The load balancer forwards the location update to the persistent connection on the WebSocket server for that client.
3. The WebSocket server saves the location data to the location history database.
4. The WebSocket server updates the new location in the location cache. The update refreshes the TTL. The WebSocket server also saves the new location in a variable in the user's WebSocket connection handler for subsequent distance calculations.
5. The WebSocket server publishes the new location to the user's channel in the Redis Pub/Sub server. Steps 3 to 5 can be executed in parallel.
6. When Redis Pub/Sub receives a location update on a channel, it broadcasts the update to all the subscribers (WebSocket connection handlers). In this case, the subscribers are all the online friends of the user sending the update. For each subscriber (i.e., for each of the user's friends), its WebSocket connection handler would receive the user location update.
7. On receiving the message, the WebSocket server, on which the connection handler lives, computes the distance between the user sending the new location (the location



data is in the message) and the subscriber (the location data is stored in a variable with the WebSocket connection handler for the subscriber).

8. This step is not drawn on the diagram. If the distance does not exceed the search radius, the new location and the last updated timestamp are sent to the subscriber's client. Otherwise, the update is dropped.

## STEP 1: UNDERSTANDING & QUESTIONS

- FR
  - How close is considered to be near by?
    - 5 Miles
  - Can the distance be assumed to be straight line distance?
    - Yes

    - How many users? How many actual users of near by feature?
        - 100 million and around 10% of them
    - Do we need to store location history?
        - Yes
    - If a friend is offline for more than 10 minutes can he/she be ignored?
        - Yes
- Non-FR
    - Low Latency
    - Reliability
    - Eventual Consistency

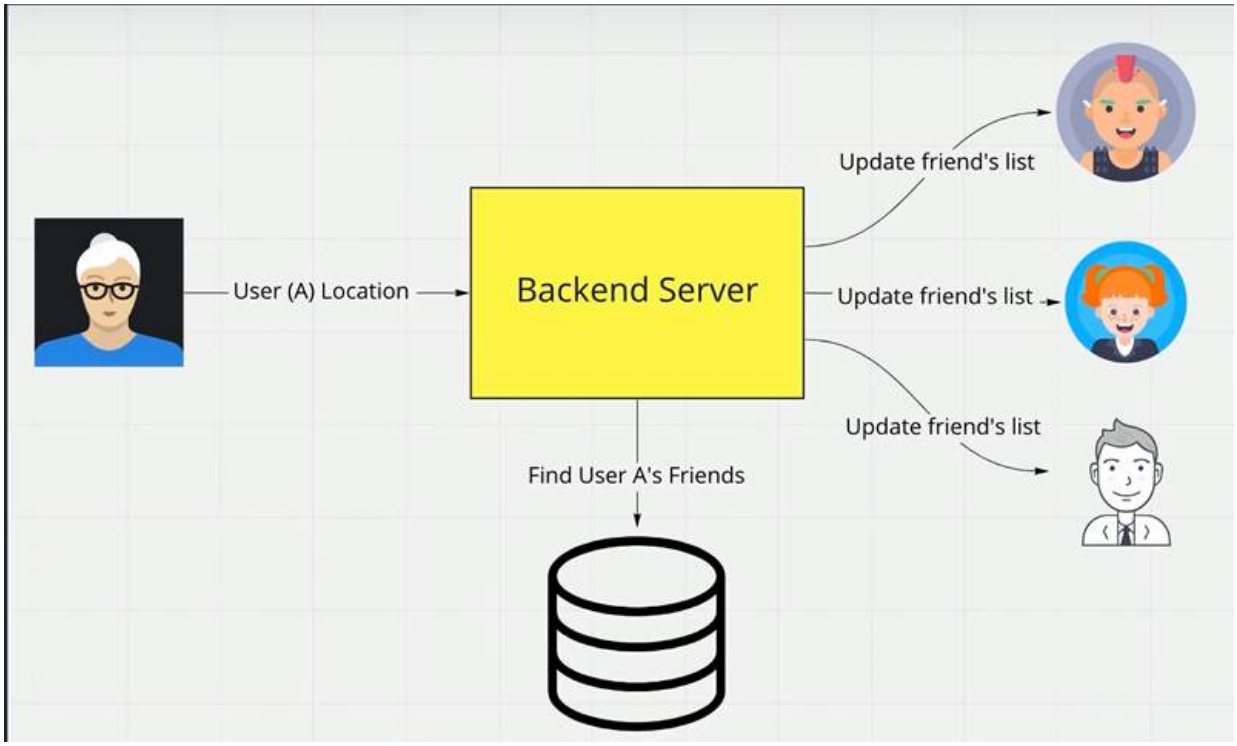
## STEP 2: CAPACITY ESTIMATION

- Each User sends the update every 30 seconds and we have around 10% of 100 million using this feature
- 10 Million / 30 seconds = **334000 updates per second**
- If we assume that each user has around 400 friends. Even if we only have 10 percent of the friends online
    - 334000 * 400 * 10% ~= 14 Million location updates per second

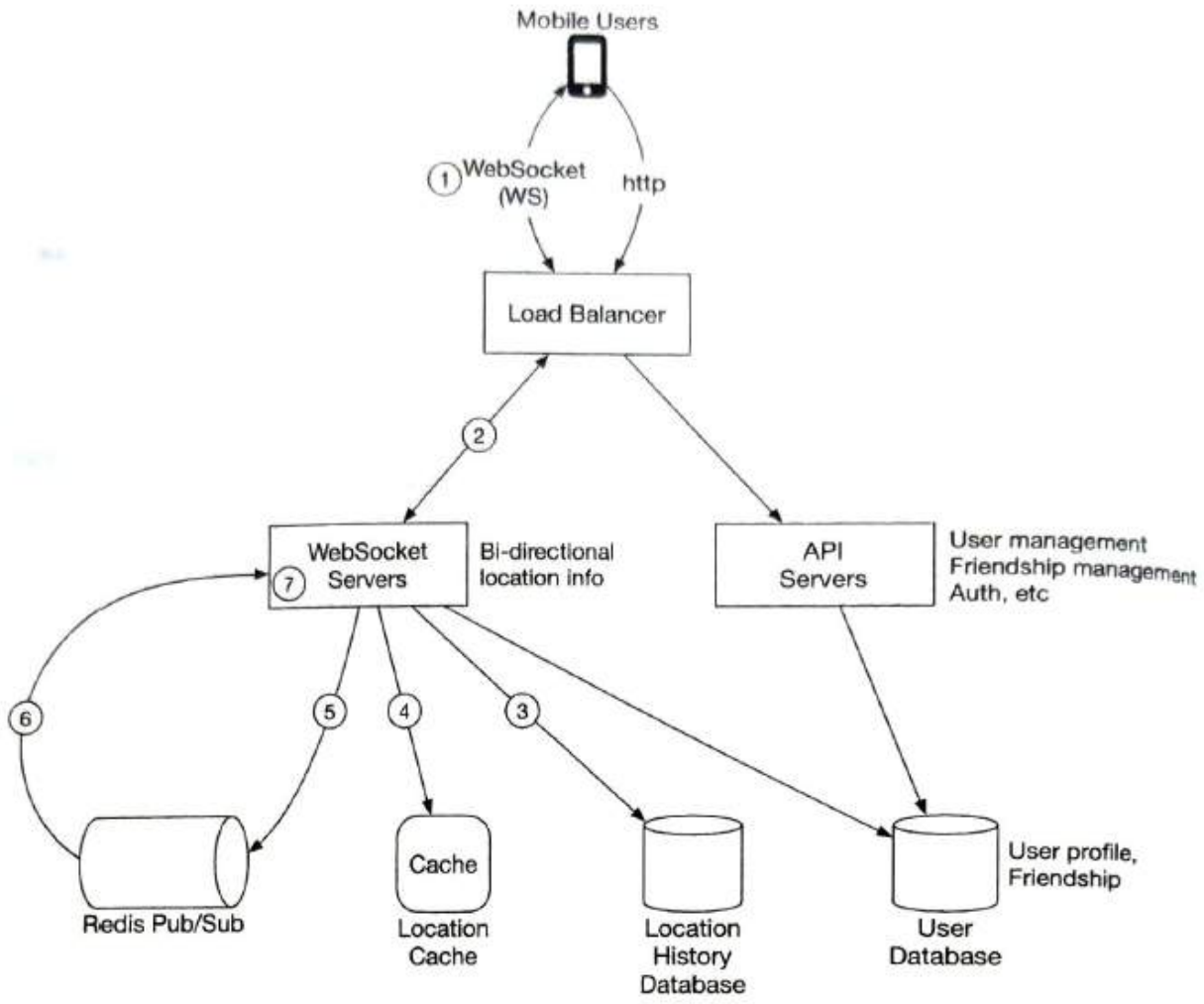
## STEP 3: HIGH-LEVEL DESIGN

- A very high-level and inefficient approach could be that whenever there is an update from the user, it can be received by the backend which can store it and then send the same update to all the friends of that user. **This is extremely inefficient and infeasible as we saw above that there nearly 14 Million updates every second**

**INEFFICIENT**

- 


- Approach proposed is as the following.




- Figure 2.7: Periodic location update

- Load Balancers balance the load evenly and the API servers are primarily responsible for adding/removing friends and updating user profiles
- WebSocket Servers are stateful servers to handle near real-time update of friends' locations. When there is an update from a friend, it is updated to the client through this connection. It is also responsible for the client Initialization

### Redis location cache

Redis is used to store the most recent location data for each active user. There is a Time to Live (TTL) set on each entry in the cache. When the TTL expires, the user is no longer active and the location data is expunged from the cache. Every update refreshes the TTL. Other KV stores that support TTL could also be used.

- 

### User database

The user database stores user data and user friendship data. Either a relational database or a NoSQL database can be used for this.

### Location history database

This database stores users' historical location data. It is not directly related to the "nearby friends" feature.

### Redis Pub/Sub server

Redis Pub/Sub [2] is a very lightweight message bus. Channels in Redis Pub/Sub are very cheap to create. A modern Redis server with GBs of memory could hold millions of channels (also called topics). Figure 2.6 shows how Redis Pub/Sub works.

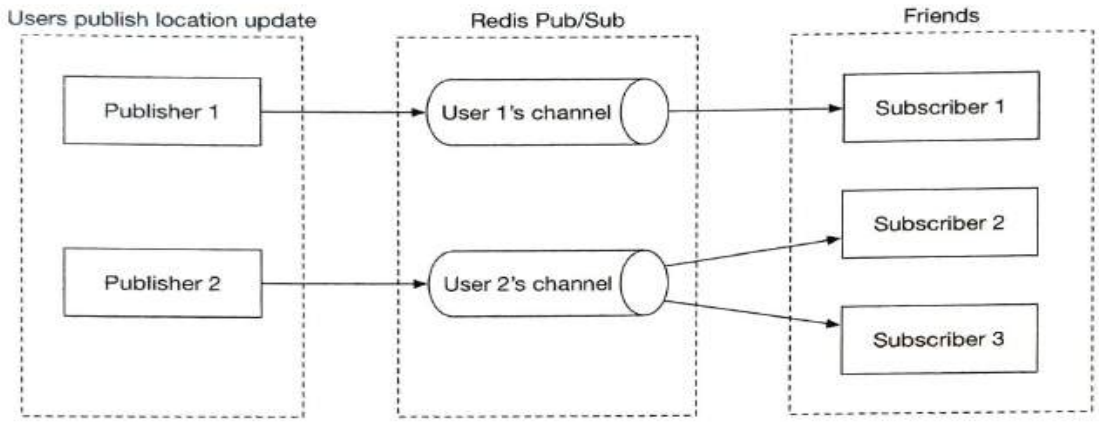



Figure 2.6: Redis Pub/Sub

In this design, location updates received via the WebSocket server are published to the user's own channel in the Redis Pub/Sub server. A dedicated WebSocket connection handler for each active friend subscribes to the channel. When there is a location update, the WebSocket handler function gets invoked, and for each active friend, the function recomputes the distance. If the new distance is within the search radius, the new location and timestamp are sent via the WebSocket connection to the friend's client. Other

- 

1. The mobile client sends a location update to the load balancer.
2. The load balancer forwards the location update to the persistent connection on the WebSocket server for that client.
3. The WebSocket server saves the location data to the location history database.
4. The WebSocket server updates the new location in the location cache. The update refreshes the TTL. The WebSocket server also saves the new location in a variable in the user's WebSocket connection handler for subsequent distance calculations.
5. The WebSocket server publishes the new location to the user's channel in the Redis Pub/Sub server. Steps 3 to 5 can be executed in parallel.
6. When Redis Pub/Sub receives a location update on a channel, it broadcasts the update to all the subscribers (WebSocket connection handlers). In this case, the subscribers are all the online friends of the user sending the update. For each subscriber (i.e., for each of the user's friends), its WebSocket connection handler would receive the user location update.
7. On receiving the message, the WebSocket server, on which the connection handler lives, computes the distance between the user sending the new location (the location



data is in the message) and the subscriber (the location data is stored in a variable with the WebSocket connection handler for the subscriber).

8. This step is not drawn on the diagram. If the distance does not exceed the search radius, the new location and the last updated timestamp are sent to the subscriber's client. Otherwise, the update is dropped.

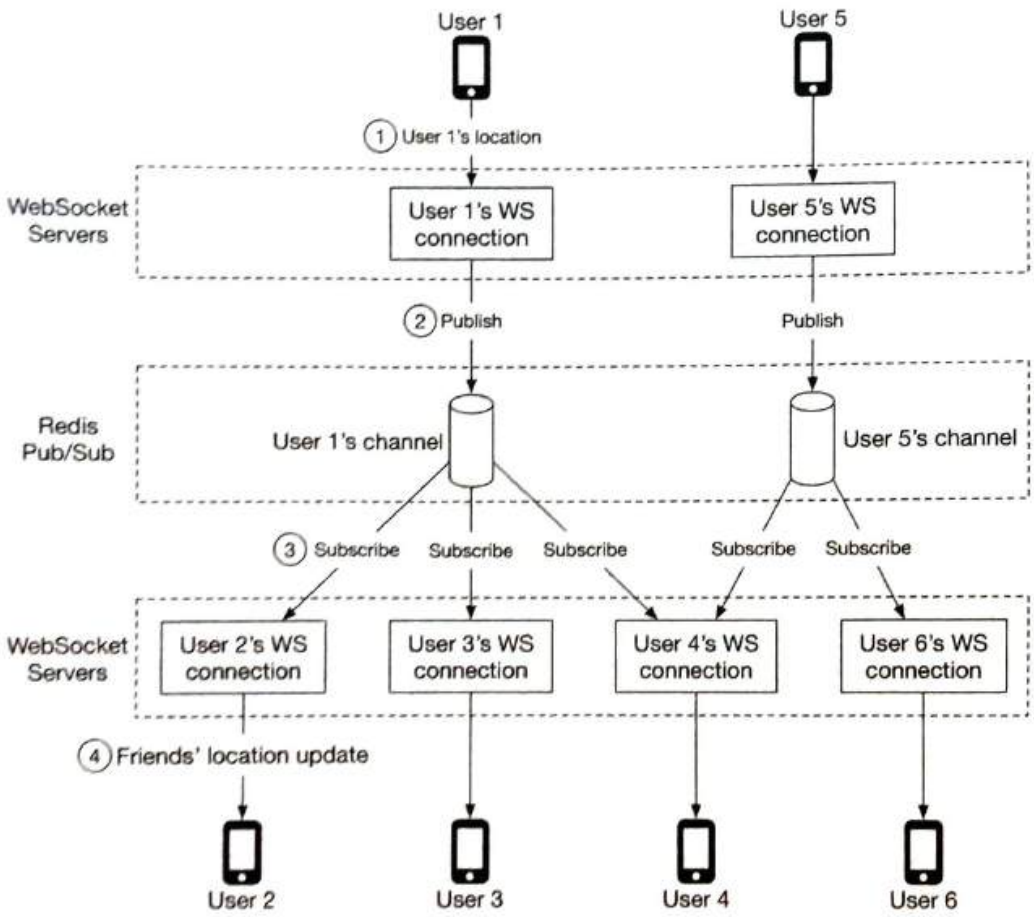



Figure 2.8: Send location update to friends

1. When User 1's location changes, their location update is sent to the WebSocket server which holds User 1's connection.
2. The location is published to User 1's channel in Redis Pub/Sub server.
3. Redis Pub/Sub server broadcasts the location update to all subscribers. In this case,

## STEP 4: API DESIGN & DATA MODEL

## API design

Now that we have created a high-level design, let's list APIs needed.

**WebSocket**: Users send and receive location updates through the WebSocket protocol. At the minimum, we need the following APIs.

**1. Periodic location update**
Request: Client sends latitude, longitude, and timestamp.
Response: Nothing.

**2. Client receives location updates**
Data sent: Friend location data and timestamp.

**3. WebSocket initialization**
Request: Client sends latitude, longitude, and timestamp.
Response: Client receives friends' location data.

**4. Subscribe to a new friend**
Request: WebSocket server sends friend ID.
Response: Friend's latest latitude, longitude, and timestamp.

**5. Unsubscribe a friend**
Request: WebSocket server sends friend ID.
Response. Nothing.

**HTTP requests**: the API servers handle tasks like adding/removing friends, updating user profiles, etc. These are very common and we will not go into detail here.

- ***Location Cache***

- For Current location, cache is enough since that's all we need. **Redis is an excellent choice for this**

| key | value |
|---|---|
| user_id | {latitude, longitude, timestamp} |

Table 2.1: Location cache

- ***Location History DB***

- **Cassandra**

The location history database stores users' historical location data and thc schema looks like this:

| user_id | latitude | longitude | timestamp |
|---|---|---|---|

We need a database that handles the heavy-write workload well and can be horizontally scaled. Cassandra is a good candidate. We could also use a relational database. However, with a relational database, the historical data would not fit in a single instance so we need to shard that data. The most basic approach is to shard by user ID. This sharding scheme ensures that load is evenly distributed among all the shards, and operationally, it is easy to maintain.

- 

## STEP 5: DESIGN DEEP DIVE

- ***Client Initialization***

### Client initialization

The mobile client on startup establishes a persistent WebSocket connection with one o the WebSocket server instances. Each connection is long-running. Most modern lan guages are capable of maintaining many long-running connections with a reasonably small memory footprint.

When a WebSocket connection is initialized, the client sends the initial location of th user, and the server performs the following tasks in the WebSocket connection han dler.

1. It updates the user's location in the location cache.
2. It saves the location in a variable of the connection handler for subsequent calcula tions.
3. It loads all the user's friends from the user database.
4. It makes a batched request to the location cache to fetch the locations for all th friends. Note that because we set a TTL on each entry in the location cache to matc our inactivity timeout period, if a friend is inactive then their location will not be i the location cache.
5. For each location returned by the cache, the server computes the distance betwee the user and the friend at that location. If the distance is within the search radiu the friend's profile, location, and last updated timestamp are returned over the Web Socket connection to the client.
6. For each friend, the server subscribes to the friend's channel in the Redis Pub/Su server. We will explain our use of Redis Pub/Sub shortly. Since creating a new chan nel is cheap, the user subscribes to all active and inactive friends. The inactive friend will take up a small amount of memory on the Redis Pub/Sub server, but they wi not consume any CPU or I/O (since they do not publish updates) until they com online.
7. It sends the user's current location to the user's channel in the Redis Pub/Sub serve

- ***Location Cache & PUB/SUB***

### Location cache

We choose Redis to cache the most recent locations of all the active users. As mentioned earlier, we also set a TTL on each key. The TTL is renewed upon every location update. This puts a cap on the maximum amount of memory used. With 10 million active users at peak, and with each location taking no more than 100 bytes, a single modern Redis server with many GBs of memory should be able to easily hold the location information for all users.

However, with 10 million active users roughly updating every 30 seconds, the Redis server will have to handle 334K updates per second. That is likely a little too high, even for a modern high-end server. Luckily, this cache data is easy to shard. The location data for each user is independent, and we can evenly spread the load among several Redis servers by sharding the location data based on user ID.

To improve availability, we could replicate the location data on each shard to a standby node. If the primary node goes down, the standby could be quickly promoted to minimize downtime.

### Redis Pub/Sub server

The Pub/Sub server is used as a routing layer to direct messages (location updates) from one user to all the online friends. As mentioned earlier, we choose Redis Pub/Sub because it is very lightweight to create new channels. A new channel is created when someone subscribes to it. If a message is published to a channel that has no subscribers, the message is dropped, placing very little load on the server. When a channel is created, Redis uses a small amount of memory to maintain a hash table and a linked list [3] to track the subscribers. If there is no update on a channel when a user is offline, no CPU cycles are used after a channel is created. We take advantage of this in our design in the following ways:

1. We assign a unique channel to every user who uses the "nearby friends" feature. A user would, upon app initialization, subscribe to each friend's channel, whether the friend is online or not. This simplifies the design since the backend does not need to handle subscribing to a friend's channel when the friend becomes active, or handling unsubscribing when the friend becomes inactive.
2. The tradeoff is that the design would use more memory. As we will see later, memory use is unlikely to be the bottleneck. Trading higher memory use for a simpler architecture is worth it in this case.

- We'd need distributed Redis servers as there are many updates for each second. Nearly 1 lakh. So, we can have a hash ring which coordinates with multiple redis servers. Using

that, we can locate where our client's channel is located and accordingly the publishing can be made

When we inevitably have to scale, be mindful of these potential issues:

- When we resize a cluster, many channels will be moved to different servers on the hash ring. When the service discovery component notifies all the WebSocket servers of the hash ring update, there will be a ton of resubscription requests.
- During these mass resubscription events, some location updates might be missed by the clients. Although occasional misses are acceptable for our design, we should minimize the occurrences.
- Because of the potential interruptions, resizing should be done when usage is at its lowest in the day.

How is resizing actually done? It is quite simple. Follow these steps:

- Determine the new ring size, and if scaling up, provision enough new servers.
- Update the keys of the hash ring with the new content.
- Monitor your dashboard. There should be some spike in CPU usage in the WebSocket cluster.

- ***Adding/Removing Friends***

### Adding/removing friends

What should the client do when the user adds or removes a friend? When a new friend is added, the client's WebSocket connection handler on the server needs to be notified, so it can subscribe to the new friend's Pub/Sub channel.

Since the "nearby friends" feature is within the ecosystem of a larger app, we can assume that the "nearby friends" feature could register a callback on the mobile client whenever a new friend is added. The callback, upon invocation, sends a message to the WebSocket server to subscribe to the new friend's Pub/Sub channel. The WebSocket server also

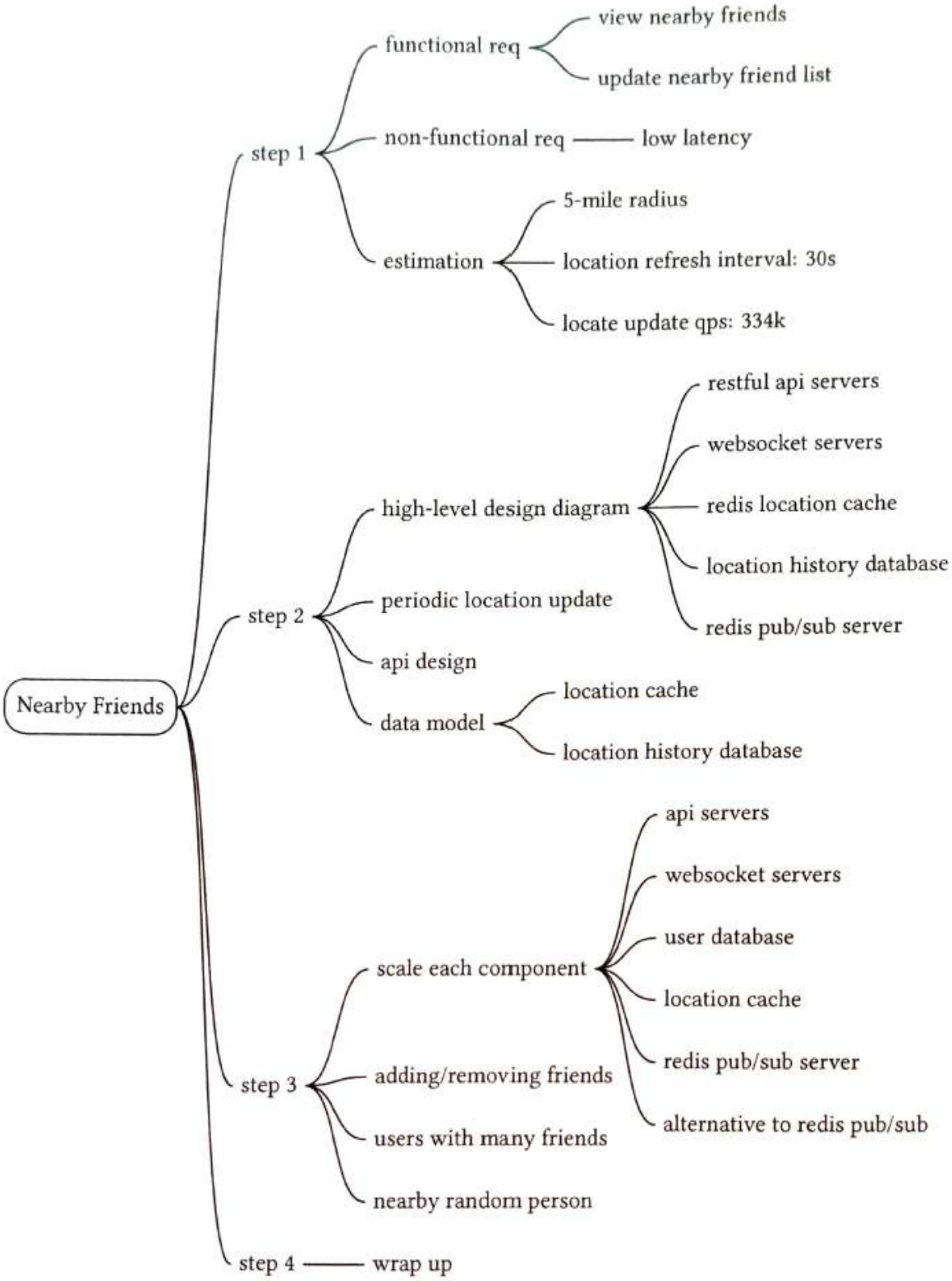


returns a message containing the new friend's latest location and timestamp, if they are active.

Likewise, the client could register a callback in the application whenever a friend is removed. The callback would send a message to the WebSocket server to unsubscribe from the friend's Pub/Sub channel.

This subscribe/unsubscribe callback could also be used whenever a friend has opted in or out of the location update.

- ***SUMMARY***


Nearby Friends
step 1
functional req
view nearby friends
update nearby friend list
non-functional req
low latency
estimation
5-mile radius
location refresh interval: 30s
locate update qps: 334k
step 2
high-level design diagram
restful api servers
websocket servers
redis location cache
location history database
redis pub/sub server
periodic location update
api design
data model
location cache
location history database
step 3
scale each component
api servers
websocket servers
user database
location cache
redis pub/sub server
alternative to redis pub/sub
adding/removing friends
users with many friends
nearby random person
step 4
wrap up

# 13 - METRICS MONITORING (GRAFANA, PROMETHEUS ETC)

## FINAL DESIGN

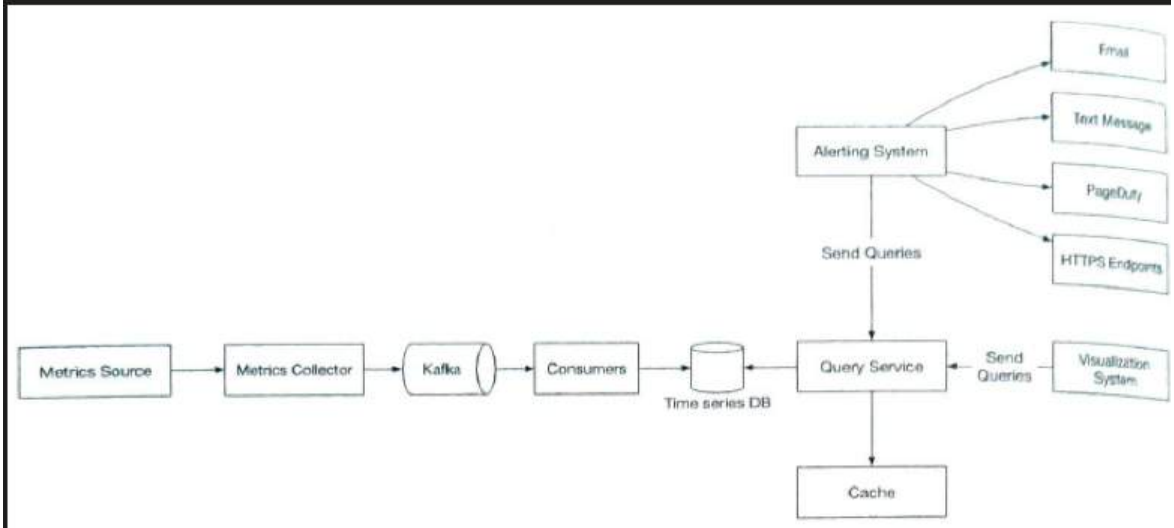



Figure 5.22: Final design

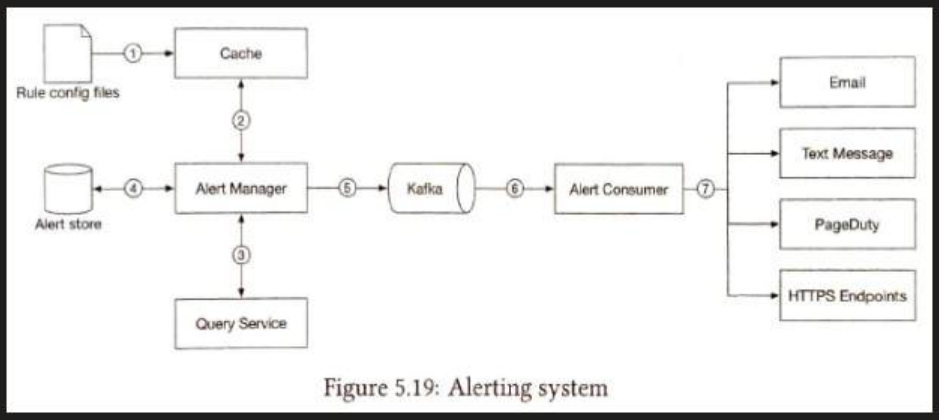



Figure 5.19: Alerting system

## STEP 1: UNDERSTANDING & QUESTIONS

- **FR**
  - Who is the system for? Internal usage?
  - What kind of metrics do we want to collect?
  - What is the scale of this infastructure?
    - 100M DAU, 1000 server pools and 100 machines per each pool
  - How long should the date be held?
    - 1 year retention
  - What are the supported alerting channels?

        - Email, phone etc
    - Metrics to be monitored
        - CPU Usage
        - Memory Usage
        - Request Count

- **NON-FR**
    - High scalability
    - Low Latency
    - Flexibility

## STEP 2: CAPACITY ESTIMATION

- 100M DAU
- 100 server pools, 100 machines per pool and 100 metrics per machine
    - Equals 10 million metrics

## STEP 3: HIGH-LEVEL DESIGN

- Data collection
- Data transmission
- Data Storage
- Analytics
- Visualization

- ***DATA MODEL***
    - Metrics data is usually recorded as time-series data.
    - Write-heavy. Millions of operational metrics are written per day
    - Read volume could be bursty in between

- ***DATA STORAGE***
    - Using a generic relational or NoSQL DB won't be enough. They won't be easy to use. Moreover, a SQL DB can't handle such heavy writes
    - Ideal ones are the **time-series database.**
    - Ex: **InfluxDB and Prometheus**
    - and Amazon offers Timestream as a time-series database [13]. According to DB-engines [14], the two most popular time-series databases are InfluxDB [15] and Prometheus, which are designed to store large volumes of time-series data and quickly perform real-time analysis on that data. Both of them primarily rely on an in-memory cache and on-disk storage. And they both handle durability and performance quite well. As shown

- ***HIGH-LEVEL DESIGN***

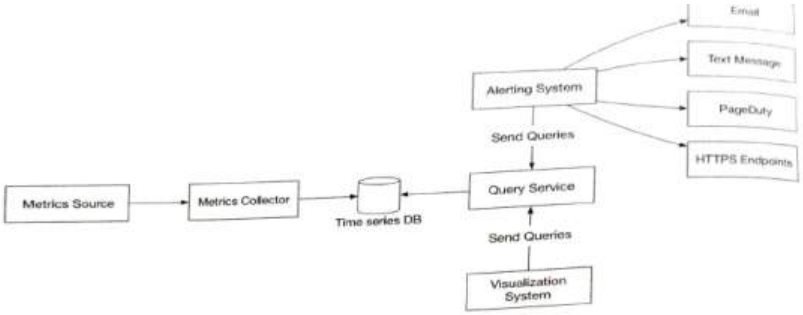



Figure 5.6: High-level design

- **Metrics source.** This can be application servers, SQL databases, message queues, etc.
- **Metrics collector.** It gathers metrics data and writes data into the time-series database.
- **Time-series database.** This stores metrics data as time series. It usually provides a custom query interface for analyzing and summarizing a large amount of time-series data. It maintains indexes on labels to facilitate the fast lookup of time-series data by labels.
- **Query service.** The query service makes it easy to query and retrieve data from the time-series database. This should be a very thin wrapper if we choose a good time-series database. It could also be entirely replaced by the time-series database's own query interface.
- **Alerting system.** This sends alert notifications to various alerting destinations.
- **Visualization system.** This shows metrics in the form of various graphs/charts.

## STEP 4: DESIGN DEEP DIVE

In this, we talk about the following

- Metrics Collection
- Scaling the metrics transmission pipeline
- Query Service
- Storage Layer
- Alerting System
- Visualization

- ***Metrics Collection***

  There are 2 ways that the metrics can be collected, push or pull methods

  - ***Pull Method***

    - We have dedicated metric collectors which pull data from the metric sources

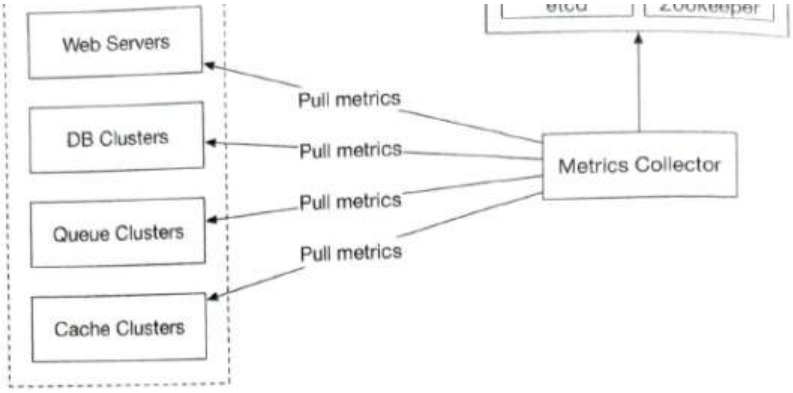



Figure 5.8: Pull model

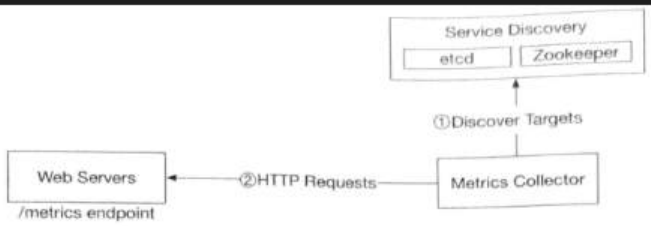



Figure 5.10: Pull model in detail

1. The metrics collector fetches configuration metadata of service endpoints from Service Discovery. Metadata include pulling interval, IP addresses, timeout and retry parameters, etc.
2. The metrics collector pulls metrics data via a pre-defined HTTP endpoint (for example, `/metrics`). To expose the endpoint, a client library usually needs to be added to the service. In Figure 5.10, the service is Web Servers.
3. Optionally, the metrics collector registers a change event notification with Service Discovery to receive an update whenever the service endpoints change. Alternatively, the metrics collector can poll for endpoint changes periodically.

At our scale, a single metrics collector will not be able to handle thousands of servers. We must use a pool of metrics collectors to handle the demand. One common problem when there are multiple collectors is that multiple instances might try to pull data from the same resource and produce duplicate data. There must exist some coordination scheme among the instances to avoid this.

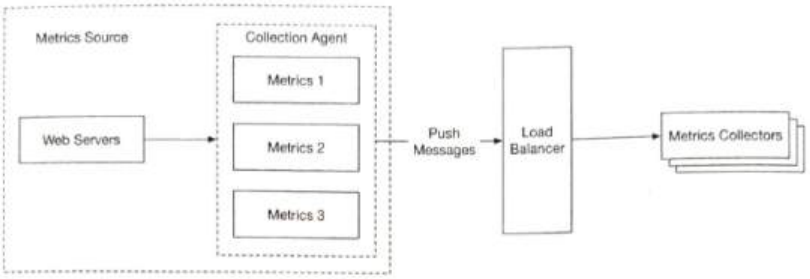
One potential approach is to designate each collector to a range in a consistent hash ring, and then map every single server being monitored by its unique name in the hash ring.

- ***Push Method***
  - In a push method, there are collections agents installed in every source and they collect the metrics from the respective sources and push that to the metric collector

Aggregation is an effective way to reduce the volume of data sent to the metrics collector. If the push traffic is high and the metrics collector rejects the push with an error, the agent

could keep a small buffer of data locally (possibly by storing them locally on disk), and resend them later. However, if the servers are in an auto-scaling group where they are rotated out frequently, then holding data locally (even temporarily) might result in data loss when the metrics collector falls behind.

To prevent the metrics collector from falling behind in a push model, the metrics collector should be in an auto-scaling cluster with a load balancer in front of it (Figure 5.13). The cluster should scale up and down based on the CPU load of the metric collector servers.




- ***Scaling the metrics transmission pipeline***

Let's zoom in on the metrics collector and time-series databases. Whether you use the push or pull model, the metrics collector is a cluster of servers, and the cluster receives enormous amounts of data. For either push or pull, the metrics collector cluster is set up for auto-scaling, to ensure that there are an adequate number of collector instances to handle the demand.

However, there is a risk of data loss if the time-series database is unavailable. To mitigate this problem, we introduce a queueing component as shown in Figure 5.15.

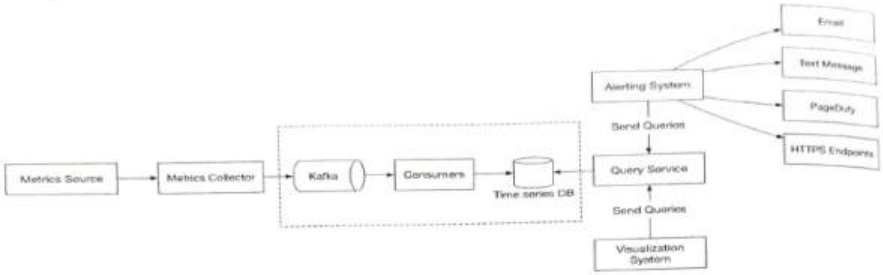



Figure 5.15: Add queues

In this design, the metrics collector sends metrics data to queuing systems like Kafka. Then consumers or streaming processing services such as Apache Storm, Flink, and Spark, process and push data to the time-series database. This approach has several advantages:

- Kafka is used as a highly reliable and scalable distributed messaging platform.
- It decouples the data collection and data processing services from each other.
- It can easily prevent data loss when the database is unavailable, by retaining the data in Kafka.

### Scale through Kafka

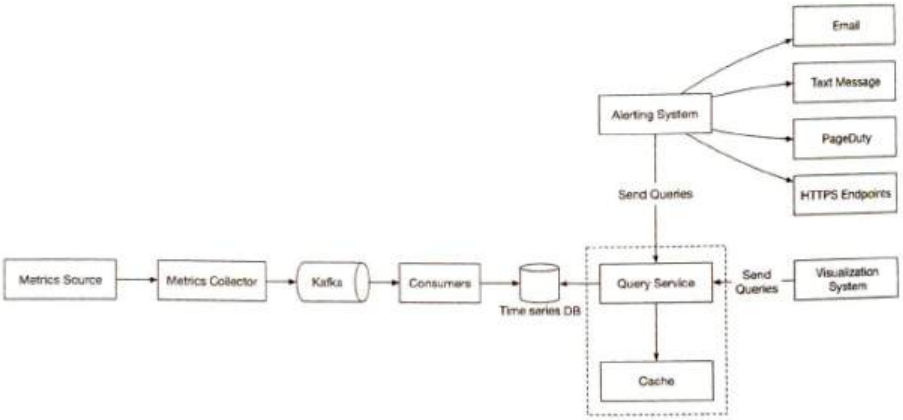
There are a couple of ways that we can leverage Kafka's built-in partition mechanism to scale our system.

- Configure the number of partitions based on throughput requirements.
- Partition metrics data by metric names, so consumers can aggregate data by metrics names.
- Further partition metrics data with tags/labels.
- Categorize and prioritize metrics so that important metrics can be processed first.

- ***Query Service***
  - Consists of cluster of servers which access time-series databases and handle requests from visualization and alerting systems
  -

### Cache layer

To reduce the load of the time-series database and make query service more performant, cache servers are added to store query results, as shown in Figure 5.17.




- ***Storage Layer***

- Data encoding and compression can be used to reduce the size of the data

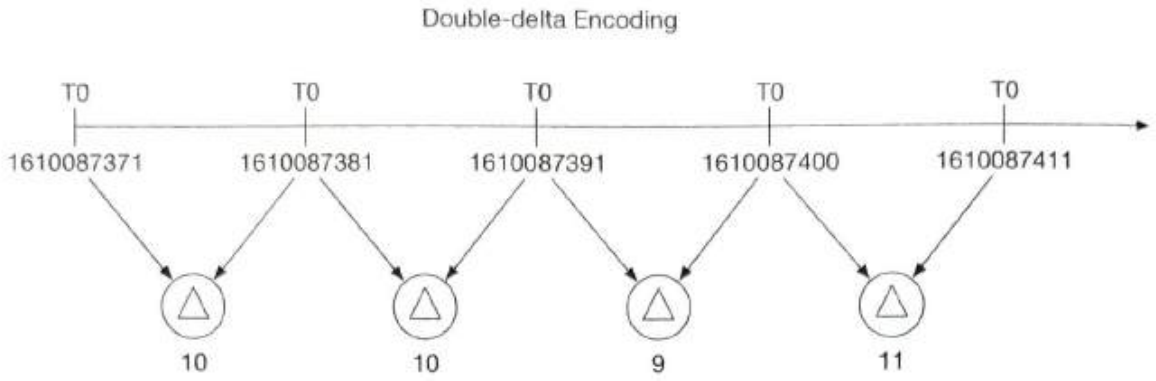



Figure 5.18: Data encoding

As you can see in the image above, 1610087371 and 1610087381 differ by only 10 seconds, which takes only 4 bits to represent, instead of the full timestamp of 32 bits. So, rather than storing absolute values, the delta of the values can be stored along with one base value like: 1610087371, 10, 10, 9, 11.

- 
- ***Alerting System***

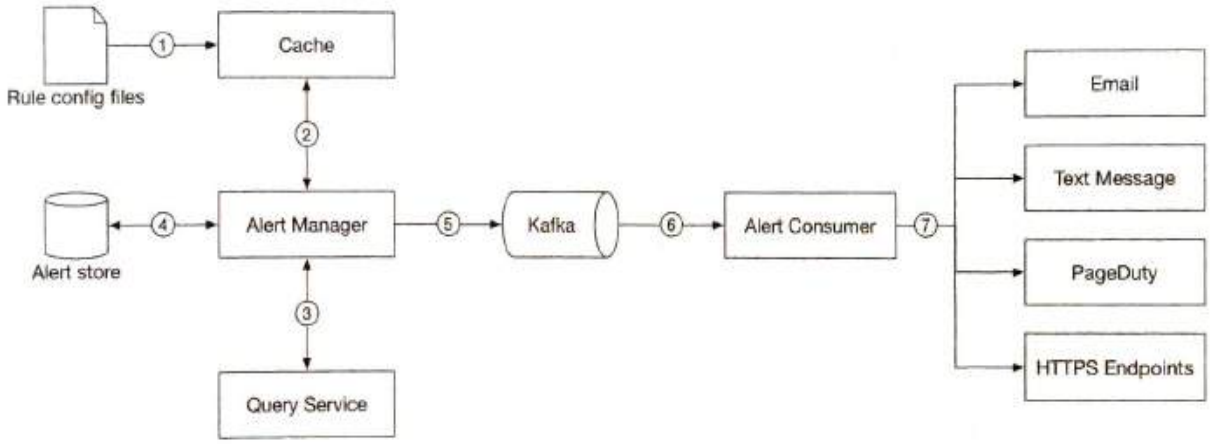



Figure 5.19: Alerting system

- 

1. Load config files to cache servers. Rules are defined as config files on the disk. YAML [29] is a commonly used format to define rules. Here is an example of alert rules:

```
- name: instance_down
rules:

# Alert for any instance that is unreachable for >5
minutes.
- alert: instance_down
    expr: up == 0
    for: 5m
    labels:
    severity: page
```

2. The alert manager fetches alert configs from the cache.

3. Based on config rules, the alert manager calls the query service at a predefined interval. If the value violates the threshold, an alert event is created. The alert manager is responsible for the following:
    - Filter, merge, and dedupe alerts. Here is an example of merging alerts that are triggered within one instance within a short amount of time (instance 1) (Figure 5.20).

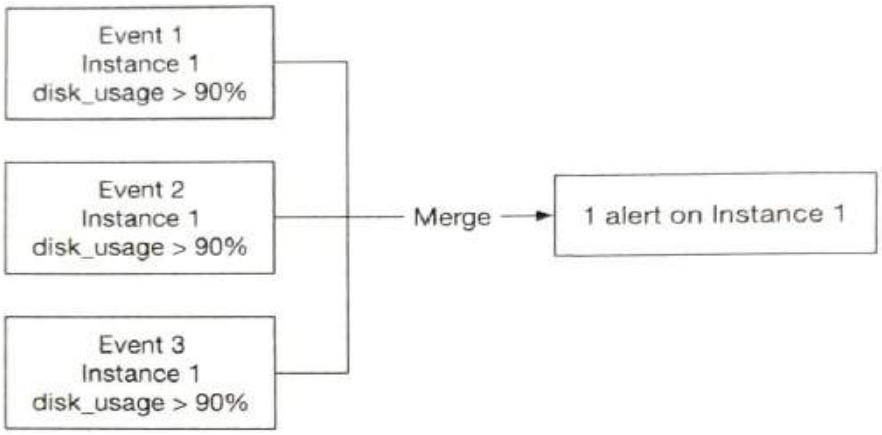



Figure 5.20: Merge alerts

    - Access control. To avoid human error and keep the system secure, it is essential to restrict access to certain alert management operations to authorized individuals only.
    - Retry. The alert manager checks alert states and ensures a notification is sent at least once.
4. The alert store is a key-value database, such as Cassandra, that keeps the state (inactive, pending, firing, resolved) of all alerts. It ensures a notification is sent at least once.
5. Eligible alerts are inserted into Kafka.
6. Alert consumers pull alert events from Kafka.
7. Alert consumers process alert events from Kafka and send notifications over to different channels such as email, text message, PagerDuty, or HTTP endpoints.

# 14 - PAYMENT SYSTEM

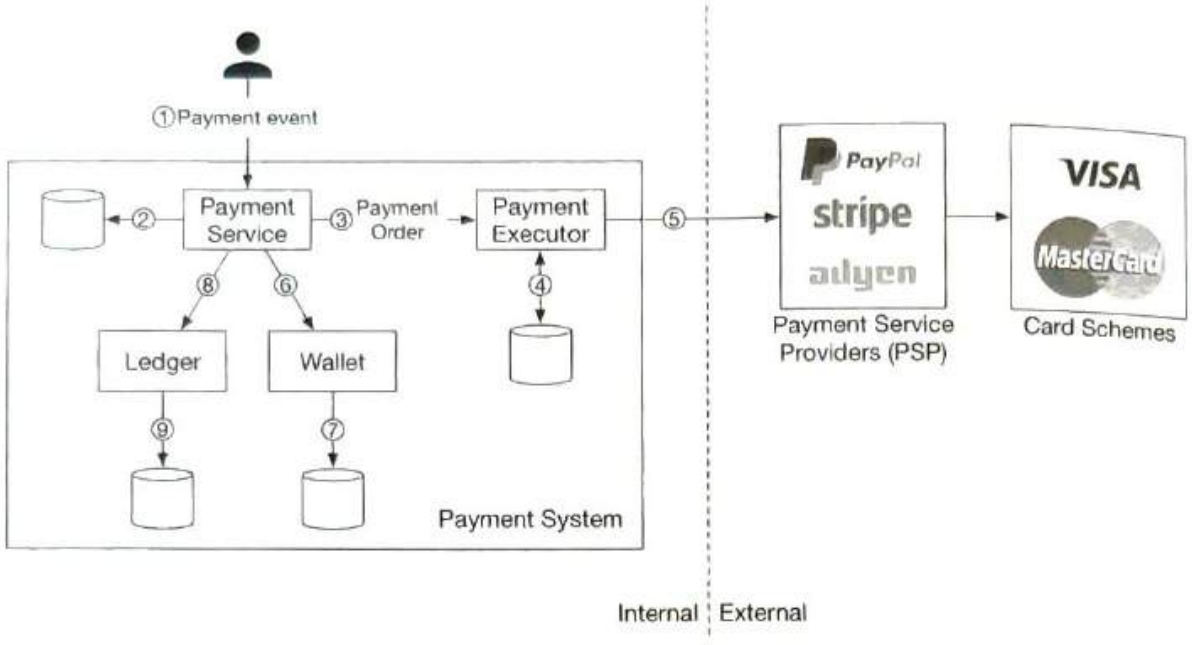
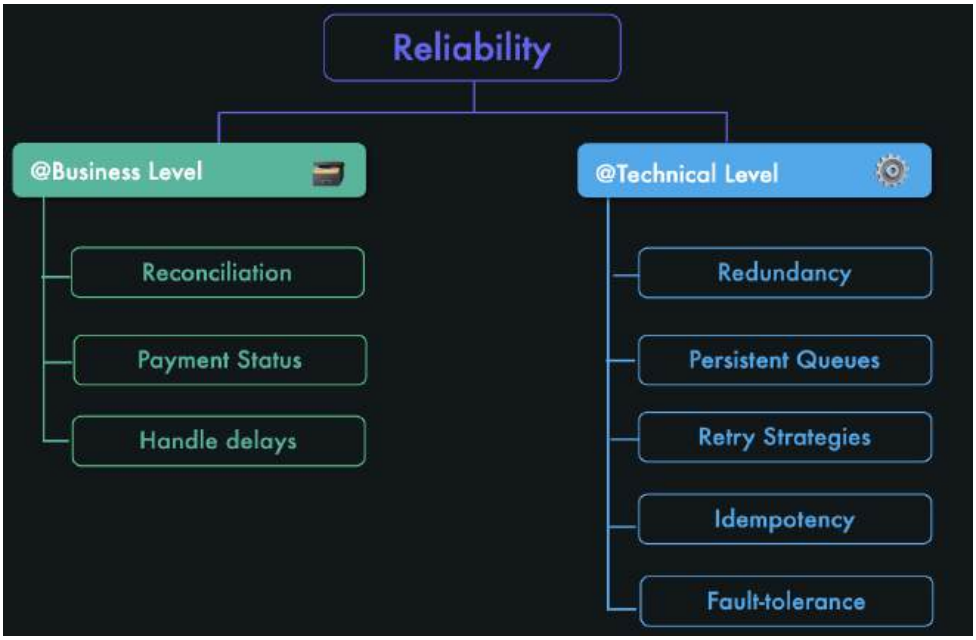



Figure 11.2: Pay-in flow

## STEP 1: UNDERSTANDING & QUESTIONS

- ***FR***
  - What kind of a payment system? Internal system?
  - What all payment options are supported?
  - Are we handling the payments ourselves? Usually Payment Service Providers these
  - Do we store the credit card information on our system?
  - Global payment system?
  - Are we handling both pay-in and pay-out flow?

- ***NON-FR***




## STEP 2: CAPACITY ESTIMATION

- 1 million transactions per day. Therefore, 10 transactions per second
- That's not very high. Hence, we can understand that the DB is not going to need to handle high throughput
- RDBMS is enough

## STEP 3: HIGH-LEVEL DESIGN

- *Pay-In Flow*

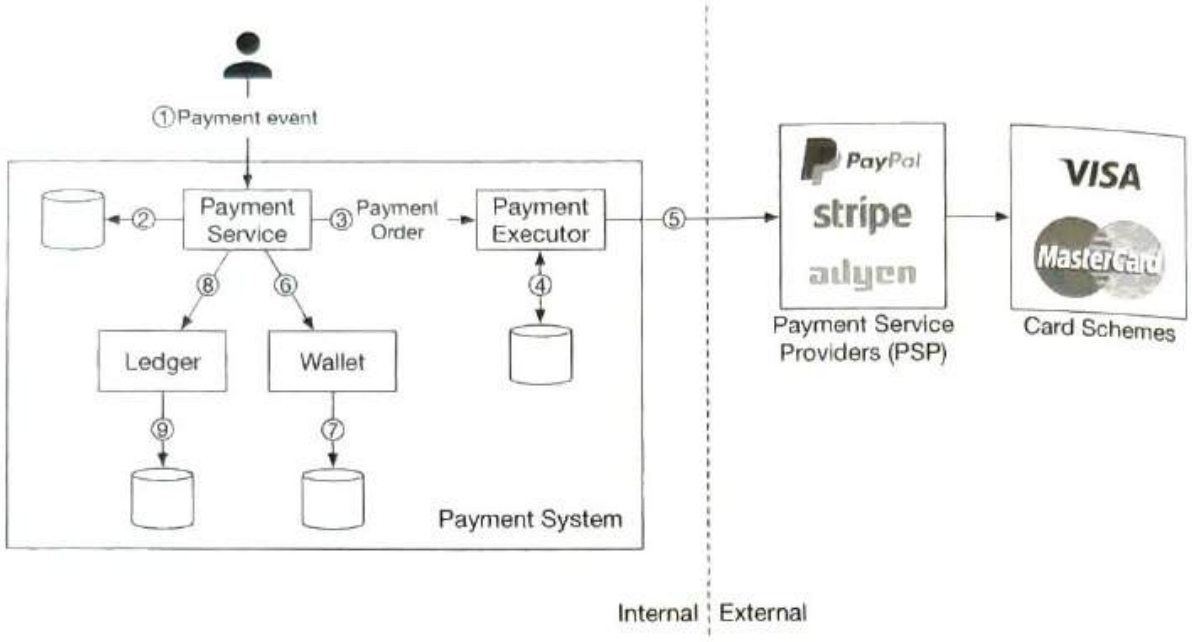



Figure 11.2: Pay-in flow

- [Payment Service]
  - [Accepts Payments]
  - [Does risk checks (using 3rd party)]
- [Payment Executor]
  - [Executes a payment via Payment Service Provider]
- [Ledger]

### Ledger

The ledger keeps a financial record of the payment transaction. For example, when a user pays the seller $1, we record it as debit $1 from the user and credit $1 to the seller. The ledger system is very important in post-payment analysis, such as calculating the total revenue of the e-commerce website or forecasting future revenue.

- [Wallet]
  - [Keeps the account balance of the merchant. Records how much a user has paid in total]

1. When a user clicks the "place order" button, a payment event is generated and sent to the payment service.
2. The payment service stores the payment event in the database.
3. Sometimes, a single payment event may contain several payment orders. For example, you may select products from multiple sellers in a single checkout process. If the e-commerce website splits the checkout into multiple payment orders, the payment service calls the payment executor for each payment order.
4. The payment executor stores the payment order in the database.
5. The payment executor calls an external PSP to process the credit card payment.
6. After the payment executor has successfully processed the payment, the payment service updates the wallet to record how much money a given seller has.
7. The wallet server stores the updated balance information in the database.
8. After the wallet service has successfully updated the seller's balance information, the payment service calls the ledger to update it.
9. The ledger service appends the new ledger information to the database.

## STEP 4: APIs & Data Model

- POST /v1/payments

**POST /v1/payments**

This endpoint executes a payment event. As mentioned above, a single payment event may contain multiple payment orders. The request parameters are listed below:

| Field | Description | Type |
|---|---|---|
| buyer_info | The information of the buyer | json |
| checkout_id | A globally unique ID for this checkout | string |
| credit_card_info | This could be encrypted credit card information or a payment token. The value is PSP-specific. | json |
| payment_orders | A list of the payment orders | list |

Table 11.1: API request parameters (execute a payment event)

| Field | Description | Type |
|---|---|---|
| seller_account | Which seller will receive the money | string |
| amount | The transaction amount for the order | string |
| currency | The currency for the order | string (ISO 4217 [4]) |
| payment_order_id | A globally unique ID for this payment | string |

Table 11.2: payment_orders

- payment_order_id is globally unique. It is used by our system and the PSP as the deduplication ID, also called as the idempotency key

- GET /v1/payments/{:id}
    - Returns the execution status of a single payment order based on the payment_order_id

- DATA MODEL
    - We need 2 tables for payment event and payment order
    - RDBMS is preferred to NoSQL DB because of its ACID properties

The payment event table contains detailed payment event information. This is what it looks like:

| Name | Type |
|---|---|
| checkout_id | string **PK** |
| buyer_info | string |
| seller_info | string |
| credit_card_info | depends on the card provider |
| is_payment_done | boolean |

Table 11.3: Payment event

The payment order table stores the execution status of each payment order. This is what it looks like:

| Name | Type |
|---|---|
| payment_order_id | String **PK** |
| buyer_account | string |
| amount | string |
| currency | string |
| checkout_id | string **FK** |
| payment_order_status | string |
| ledger_updated | boolean |
| wallet_updated | boolean |

Table 11.4: Payment order

In the payment order table (Table 11.4), payment_order_status is an enumerated type (enum) that keeps the execution status of the payment order. Execution status includes NOT_STARTED, EXECUTING, SUCCESS, FAILED. The update logic is:

1. The initial status of payment_order_status is NOT_STARTED.



2. When the payment service sends the payment order to the payment executor, the payment_order_status is EXECUTING.
3. The payment service updates the payment_order_status to SUCCESS or FAILED depending on the response of the payment executor.

Once the payment_order_status is SUCCESS, the payment service calls the wallet service to update the seller balance and update the wallet_updated field to TRUE. Here we simplify the design by assuming wallet updates always succeed.

Once it is done, the next step for the payment service is to call the ledger service to update the ledger database by updating the ledger_updated field to TRUE.

When all payment orders under the same checkout_id are processed successfully, the payment service updates the is_payment_done to TRUE in the payment event table. A scheduled job usually runs at a fixed interval to monitor the status of the in-flight payment orders. It sends an alert when a payment order does not finish within a threshold so that engineers can investigate it.

## STEP 5: DESIGN DEEP DIVE

**We focus on the following things in this section**

- **PSP Integration**
- **Reconciliation**
- **Communication among internal servers**
- **Handling failed payments**
- **Exactly-once delivery**

- ***PSP Integration***

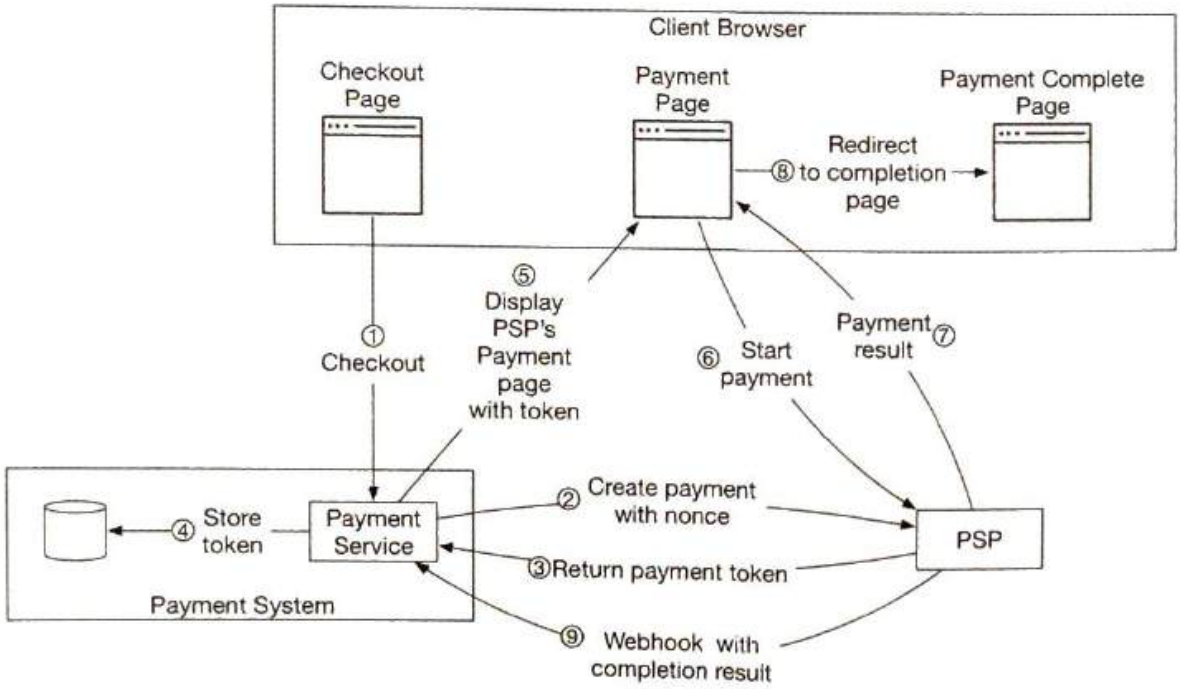



Figure 11.4: Hosted payment flow

We omitted the payment executor, ledger, and wallet in Figure 11.4 for simplicity. The payment service orchestrates the whole payment process.

1. The user clicks the "checkout" button in the client browser. The client calls the payment service with the payment order information.
2. After receiving the payment order information, the payment service sends a payment registration request to the PSP. This registration request contains payment information, such as the amount, currency, expiration date of the payment request, and the redirect URL. Because a payment order should be registered only once, there is a UUID field to ensure the exactly-once registration. This UUID is also called nonce [10]. Usually, this UUID is the ID of the payment order.
3. The PSP returns a token back to the payment service. A token is a UUID on the PSP side that uniquely identifies the payment registration. We can examine the payment registration and the payment execution status later using this token.
4. The payment service stores the token in the database before calling the PSP-hosted payment page.
5. Once the token is persisted, the client displays a PSP-hosted payment page. Mobile applications usually use the PSP's SDK integration for this functionality. Here we use Stripe's web integration as an example (Figure 11.5). Stripe provides a JavaScript library that displays the payment UI, collects sensitive payment information, and calls the PSP directly to complete the payment. Sensitive payment information is collected by Stripe. It never reaches our payment system. The hosted payment page usually needs two pieces of information:
   (a) The token we received in step 4. The PSP's javascript code uses the token to retrieve detailed information about the payment request from the PSP's backend. One important piece of information is how much money to collect.
   (b) Another important piece of information is the redirect URL. This is the web page URL that is called when the payment is complete. When the PSP's JavaScript finishes the payment, it redirects the browser to the redirect URL. Usually, the redirect URL is an e-commerce web page that shows the status of the checkout. Note that the redirect URL is different from the webhook [11] URL in step 9.

6. The user fills in the payment details on the PSP's web page, such as the credit card number, holder's name, expiration date, etc, then clicks the pay button. The PSP starts the payment processing.
7. The PSP returns the payment status.
8. The web page is now redirected to the redirect URL. The payment status that is received in step 7 is typically appended to the URL. For example, the full redirect URL could be [12]: `https://your-company.com/?tokenID=JIOUIQ123NSF&payResult=X324FSa`
9. Asynchronously, the PSP calls the payment service with the payment status via a webhook. The webhook is an URL on the payment system side that was registered with the PSP during the initial setup with the PSP. When the payment system receives payment events through the webhook, it extracts the payment status and updates the `payment_order_status` field in the Payment Order database table.

- 

- ***Reconciliation***

The answer is reconciliation. This is a practice that periodically compares the states among related services in order to verify that they are in agreement. It is usually the last line of defense in the payment system.

Every night the PSP or banks send a settlement file to their clients. The settlement file contains the balance of the bank account, together with all the transactions that took place on this bank account during the day. The reconciliation system parses the settlement file and compares the details with the ledger system. Figure 11.6 below shows where the reconciliation process fits in the system.

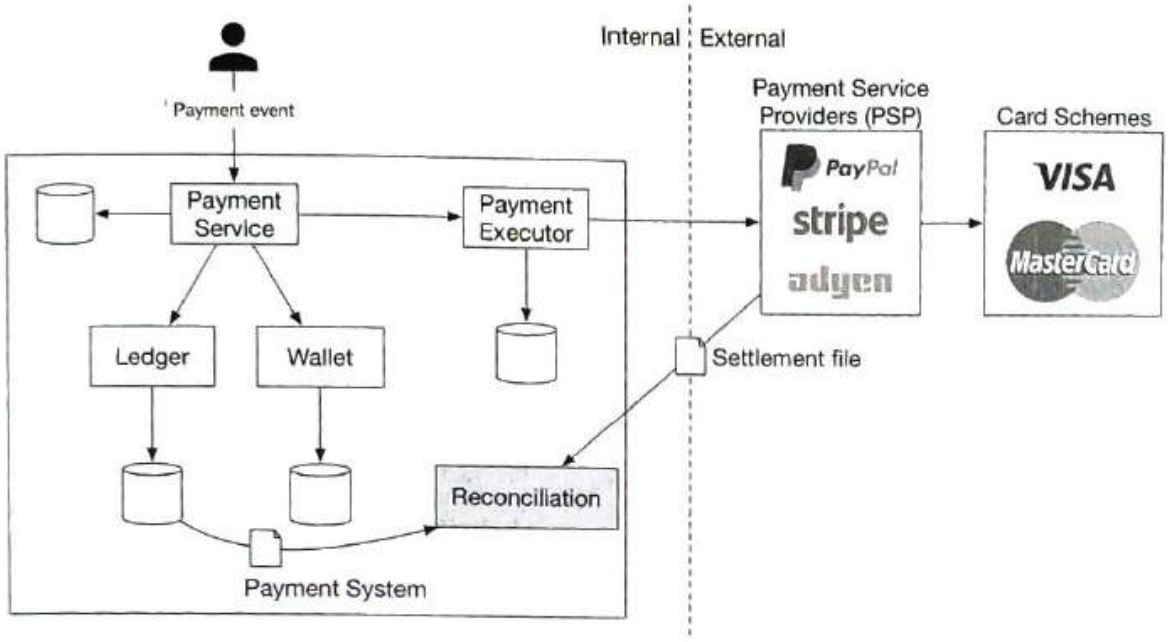



Figure 11.6: Reconciliation

- ***Communication among internal servers***

- **Asynchronous Communication**
  - **This can be preferred using queues between the services as the services can be loosely coupled then**
  - **For ledgers and wallets, we can have queues where these events can be published to which the ledger and wallet services can subscribe to. From there, data can be added to DB**

- ***Handling failed payments***

- **Whenever a failure happens, we can determine the current state of the transaction and decide whether a retry or refund is needed**

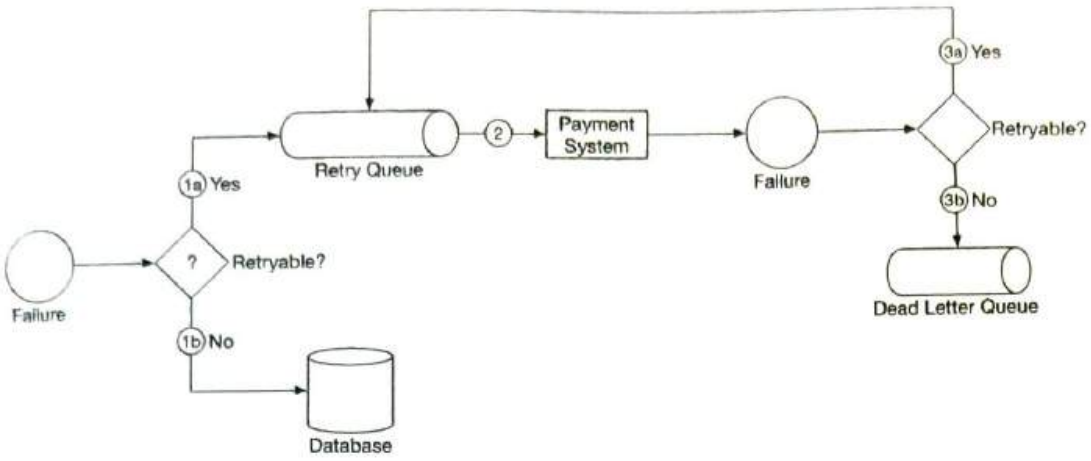



Figure 11.10: Handle failed payments

1. Check whether the failure is retryable.
   (a) Retryable failures are routed to a retry queue.
   (b) For non-retryable failures such as invalid input, errors are stored in a database.
2. The payment system consumes events from the retry queue and retries failed payment transactions.
3. If the payment transaction fails again:
   (a) If the retry count doesn't exceed the threshold, the event is routed to the retry queue.
   (b) If the retry count exceeds the threshold, the event is put in the dead letter queue. Those failed events might need to be investigated.

- ***Exactly-Once Delivery***

- **We have to make sure the payment process happens exactly once.**
- **Occasionally retry is needed as the payment could be failing due to a network error. In such cases, we can have exponential retry waiting time and ask the user to retry**

A potential problem of retrying is double payments. Let us take a look at two scenarios.

**Scenario 1**: The payment system integrates with PSP using a hosted payment page, and the client clicks the pay button twice.

**Scenario 2**: The payment is successfully processed by the PSP, but the response fails to reach our payment system due to network errors. The user clicks the "pay" button again or the client retries the payment.

- **Idempotency is the solution for this problem.**

For communication between clients (web and mobile applications) and servers, an idempotency key is usually a unique value that is generated by the client and expires after a certain period of time. A UUID is commonly used as an idempotency key and it is recommended by many tech companies such as Stripe [19] and PayPal [20]. To perform an idempotent payment request, an idempotency key is added to the HTTP header: `<idempotency-key: key_value>`.

- **In our case, it's the payment_order_id which we discussed initially.**
- **Scenario 1**

In Figure 11.12, when a user clicks "pay," an idempotency key is sent to the payment system as part of the HTTP request. In an e-commerce website, the idempotency key is usually the ID of the shopping cart right before the checkout.

For the second request, it's treated as a retry because the payment system has already seen the idempotency key. When we include a previously specified idempotency key in the request header, the payment system returns the latest status of the previous request.

To support idempotency, we can use the database's unique key constraint. For example, the primary key of the database table is served as the idempotency key. Here is how it works:

1. When the payment system receives a payment, it tries to insert a row into the database table.
2. A successful insertion means we have not seen this payment request before.
3. If the insertion fails because the same primary key already exists, it means we have seen this payment request before. The second request will not be processed.

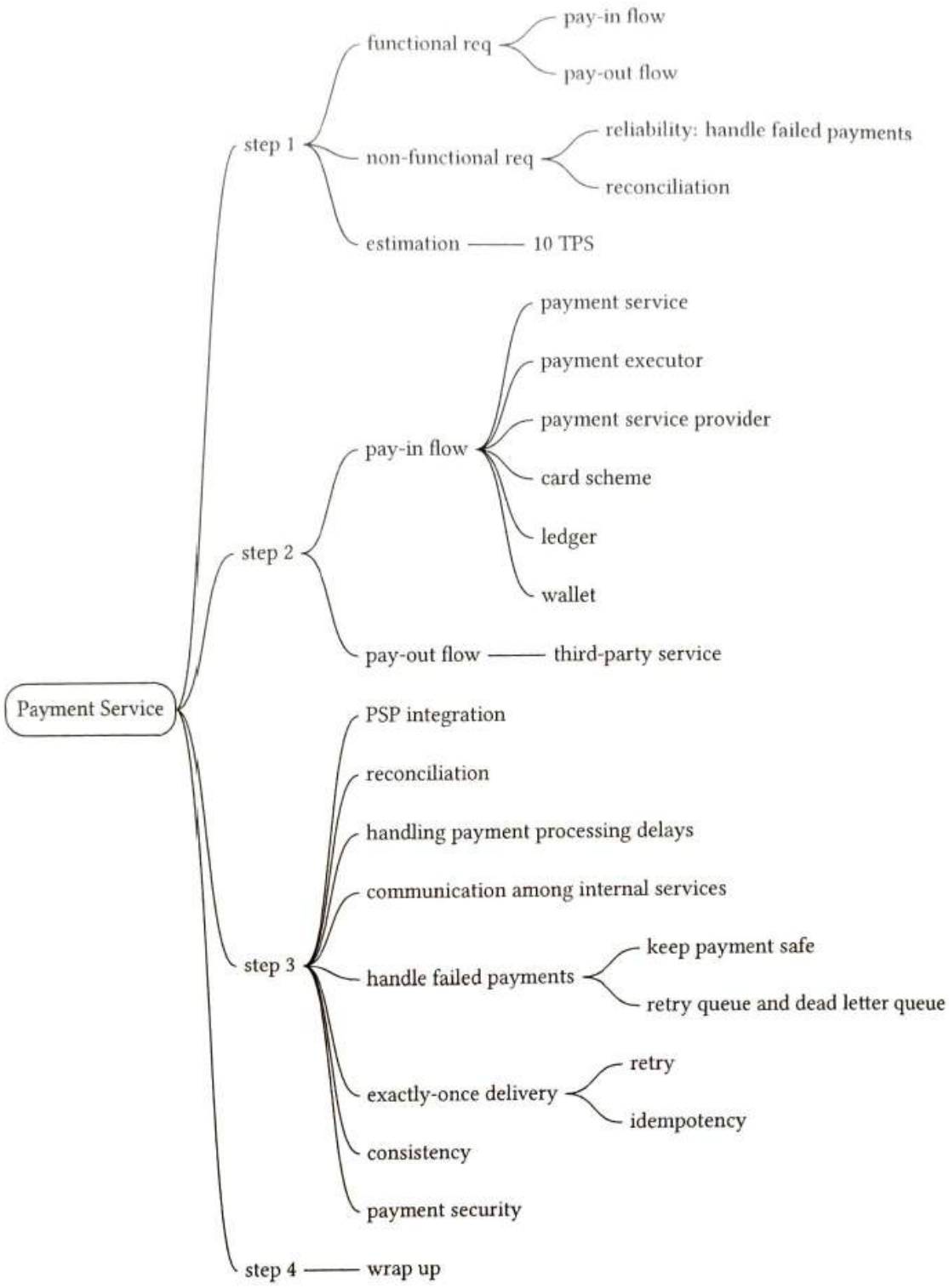
- **Scenario 2**

  As shown in Figure 11.4 (step 2 and step 3), the payment service sends the PSP a nonce and the PSP returns a corresponding token. The nonce uniquely represents the payment order, and the token uniquely maps to the nonce. Therefore, the token uniquely maps to the payment order.

  When the user clicks the "pay" button again, the payment order is the same, so the token sent to the PSP is the same. Because the token is used as the idempotency key on the PSP side, it is able to identify the double payment and return the status of the previous execution.

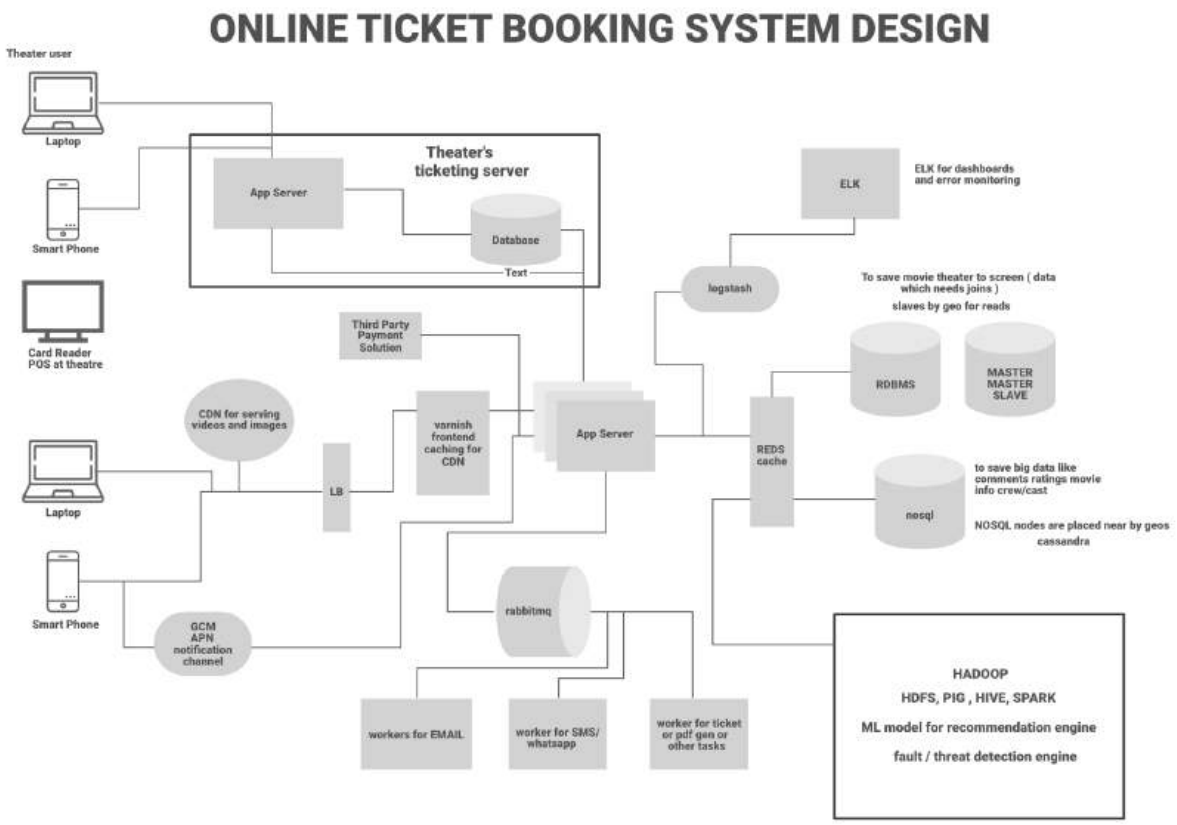
- **STEP 6: SUMMARY**

## Chapter Summary



# 15 - TICKETING SYSTEM

## FINAL DESIGN




## STEP 1: UNDERSTANDING & QUESTIONS

- FR
  - Movie booking
  - Description, Ratings, Reviews, Movie suggestions
  - Payments
  - Send the ticket by SMS/Whatsapp
  - Multiple Cities

- NON-FR

- Highly concurrent
- Moderate Latency
- Scale to different countries if needed

## STEP 2: CAPACITY ESTIMATION

- 3 billion page views per month and sells 10 million tickets a month.
- 500 cities and on average each city has 10cinemas.
- 2000 seats in each cinema and on average, there are 2 shows every day.
- Each seat booking needs 50 bytes (IDs, NumberOfSeats, ShowID, MovieID, SeatNumbers, SeatStatus, Timestamp, etc.) to store in the database.
- Information about movies and cinemas, let's assume it'll take 50 bytes.
- Total storage in a day for all shows of all cinemas of all cities: 500 cities*10 cinemas*2000 seats*2 shows*(50+50) bytes = 2GB / day.
- To store 5 years of this data, we would need around 3.6PB.

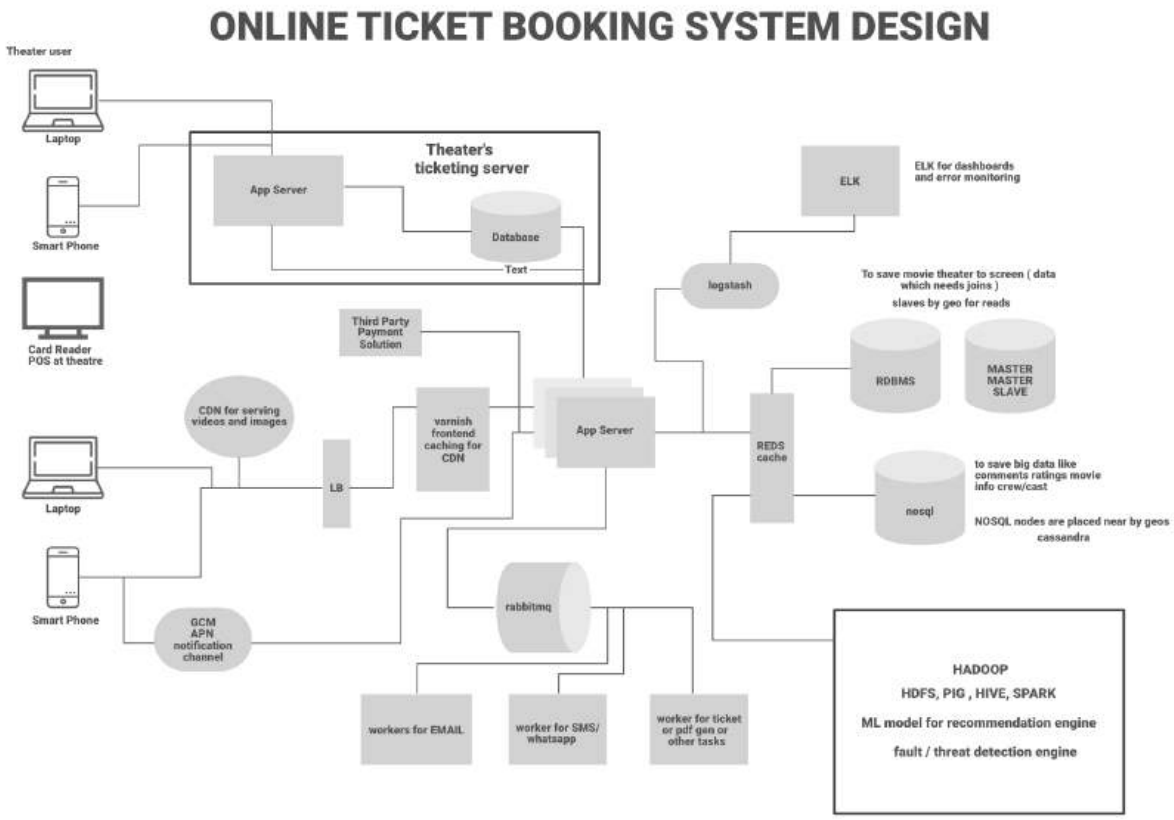
## STEP 3: APIs

- GetListOfCities()
- GetListOfEventsByCity(CityId)
- GetLocationsByCity(CityId)
- GetLocationsByEventandCity(cityid, eventid)
- GetEventsByLocationandCity(CityId, LocationId)
- GetShowTiming(eventid, locationid)
- GetAvailableSeats(eventid, locationid, showtimeid)
- VerifyUserSelectedSeatsAvailable(eventid, locationid, showtimeid, seats)
- BlockUserSelectedSeats()
- BookUserSelectedSeat()
- GetTimeoutForUserSelectedSeats()

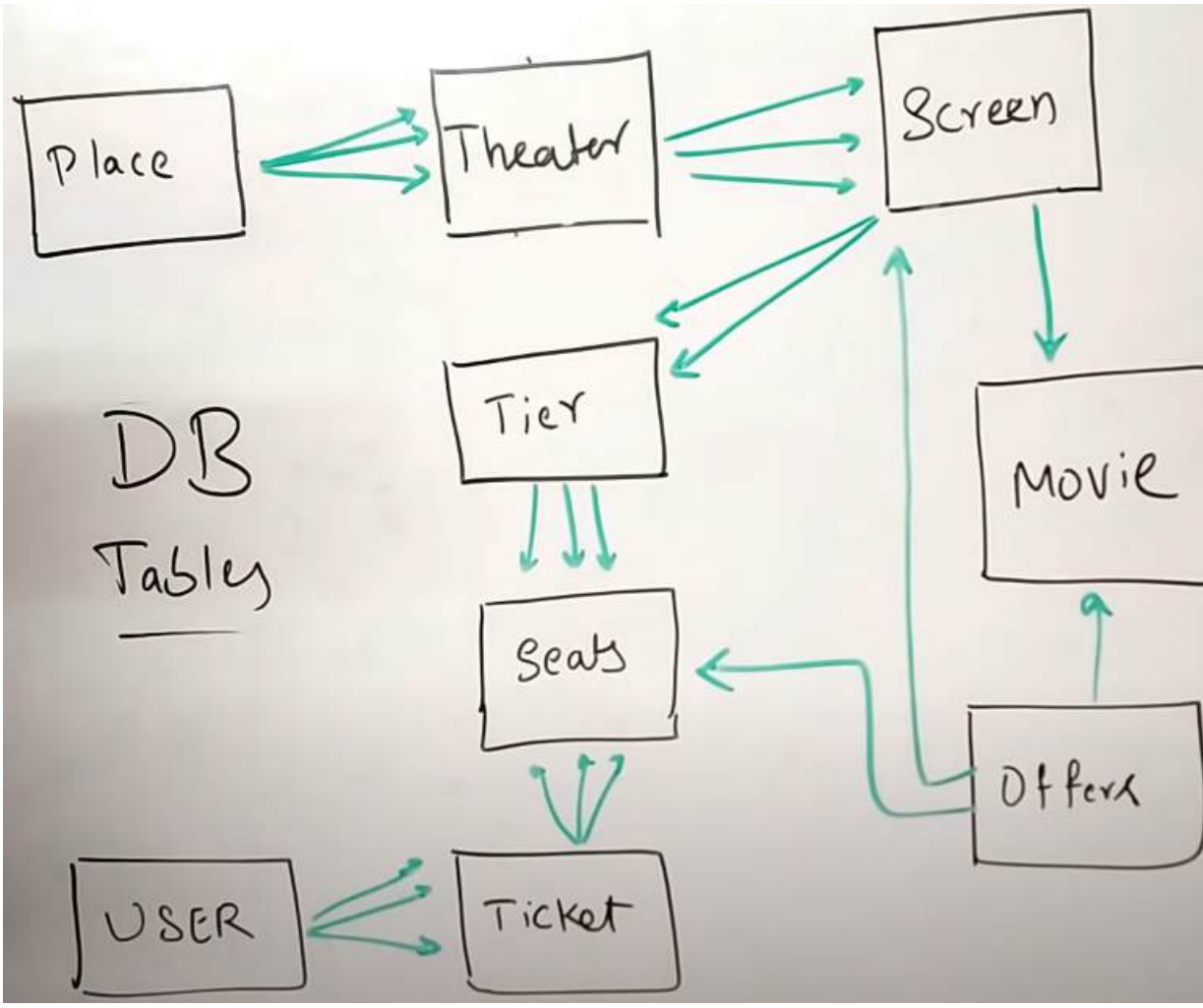
## STEP 4: DESIGN

- User looks for a city, movie, show and then books the ticket.
- CDN can be used to load the static movie information onto the user's page

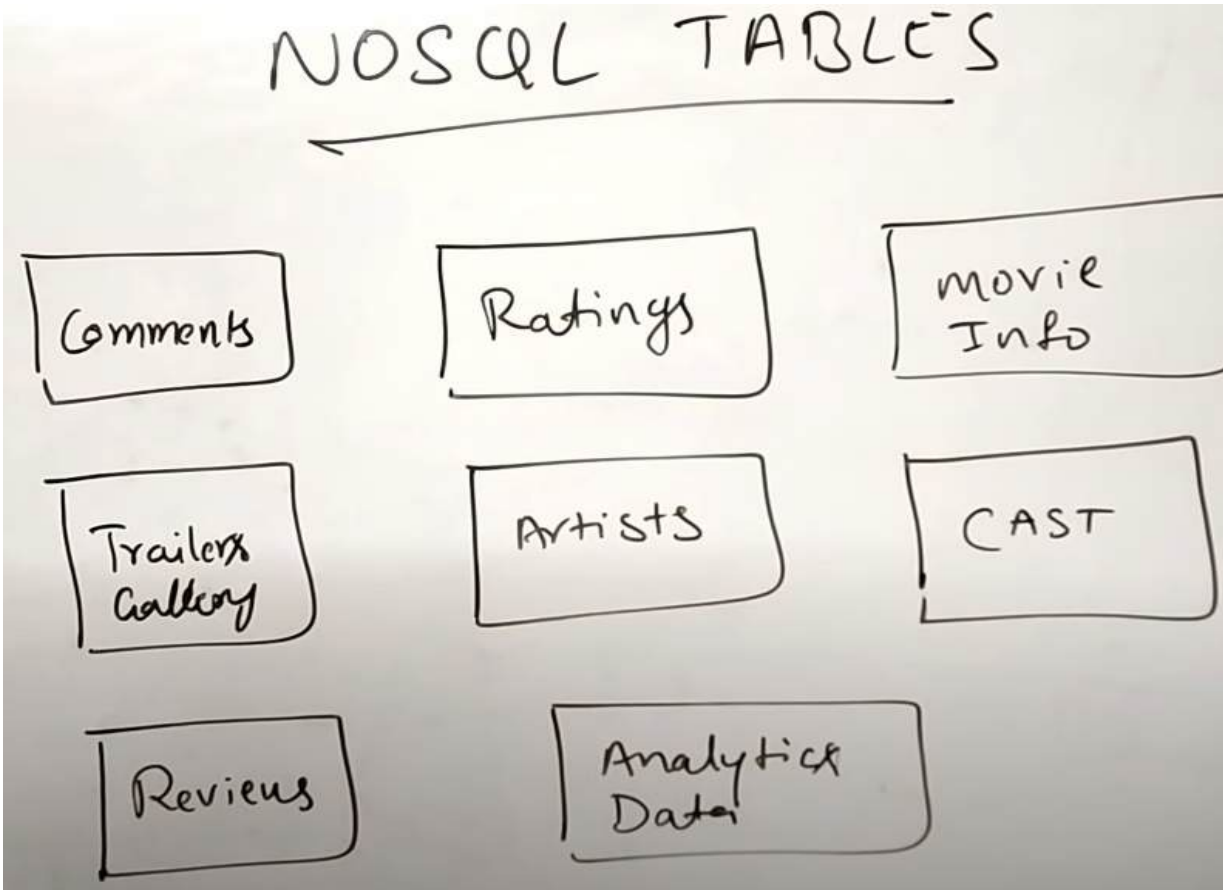
- Request goes through the load balancer and enters the app server where all the logic and processing takes place
- App servers talk to the DBs hereafter. For user and movie related information such as user login, cities, theatres, seats, movie, cast etc we can opt for RDBMS. Also, transactions can be taken care of using RDBMS. This needs to be sharded
- For movie related info, trailers, comments etc we can use NoSQL DB such as Cassandra. With this, we can have the nodes distributed as well ensuring data availability
- All these data can be cached using Redis or Memcache
- These App servers keep talking to the Theatre's ticketing server to get constant updates or to block a seat for few minutes until the customer does the payment.
- Theatre server can have an API which can do the ticket booking and give information
- Any events that deemed important can be placed in the queue such as Kafka or Rabbitmq
- Payment is taken care by the 3rd party service and once the payment is done successfully, theater generates a unique ID and sends it to the App Server. Using this unique ID, a QR code will be generated for the user's ticket which is added to the queue
- We have async workers who constantly poll this queue and look for any events. Events could be sending the movie ticket to the user via Whatsapp, Email etc
- By making these tasks asynchronously, we are reducing a lot of latency
- We are also sending all our logs and data to the Elastic Search. This can be used in order to support our search API on the page
- Similarly all the user activity and information can be dumped into Hadoop in order to build the Recommendation system depending upon the user's preferences
- At the same time, we can push the data to Spark using Kafka in between in order to analyze any trends




- ***DB TABLES***
- ***RDBMS***


Place
Theater
Screen
Tier
Movie
DB
Tables
Seats
Offers
USER
Ticket

- ***NoSQL***




***CONCURRENCY ISSUES:***

- They are:
  - same user booking the same room or seat multiple times
  - Multiple users trying to book same seat

- Few techniques to resolve this issue are
  - Pessimistic Locking
    - Places lock on a record as soon as a user starts to update it
    - Other users will have to wait until the lock is released by User 1
      - Cons
        - Deadlocks occur when multiple resources are locked
        - Not very scalable
  - Optimistic Locking

- Allows multiple users to attempt the update the same resource using version number
  1. A new column called `version` is added to the database table.
  2. Before a user modifies a database row, the application reads the version number the row.
  3. When the user updates the row, the application increases the version number by and writes it back to the database.
  4. A database validation check is put in place; the next version number should exce the current version number by 1. The transaction aborts if the validation fails ar the user tries again from step 2.

- Cons
  - Although it solves the problem in small scale, when it scales for large no.of users, it could create unpleasant user experience as multiple users try to book and in each iteration, only 1 user succeeds and the rest will have to try again
- DB Constraints

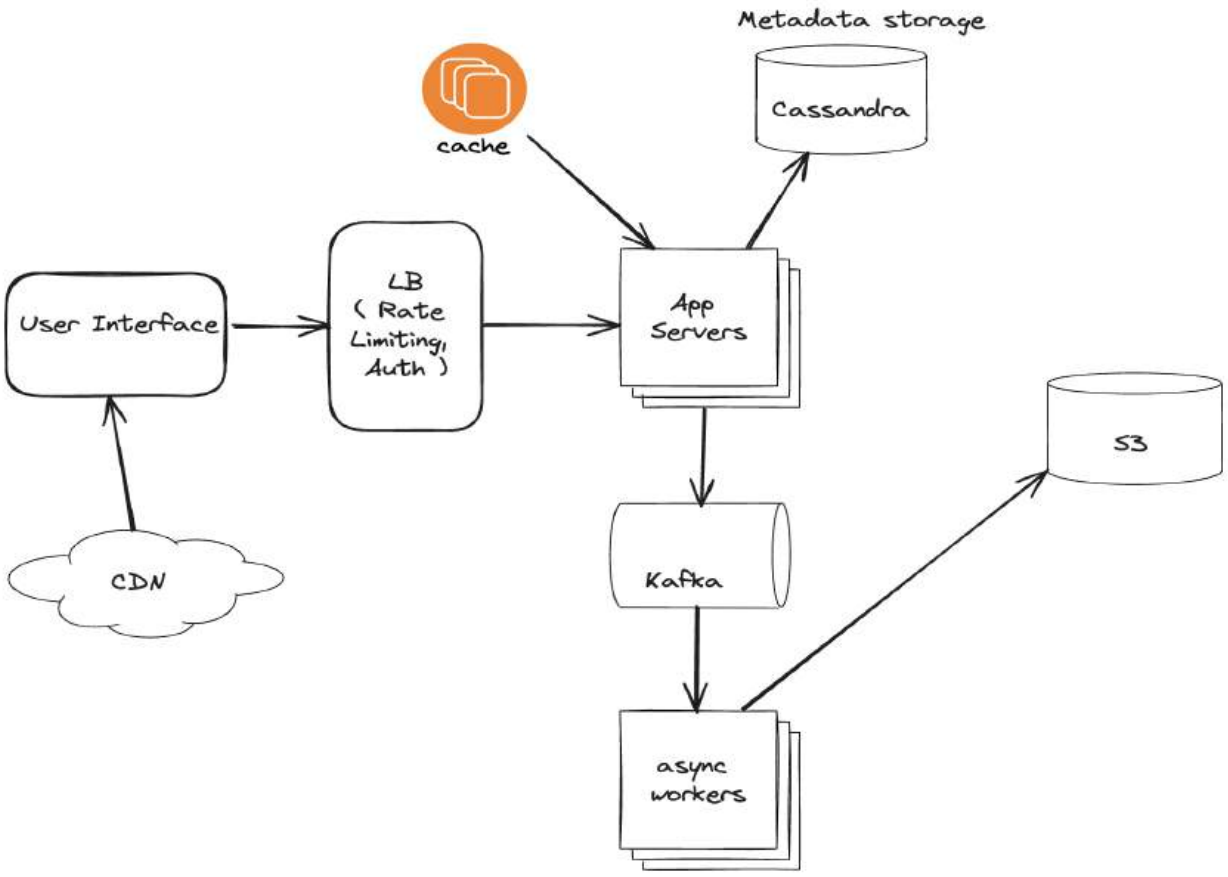
This approach is very similar to optimistic locking. Let's explore how it works. In the `room_type_inventory` table, add the following constraint:

```
CONSTRAINT `check_room_count` CHECK((`total_inventory - total_reserved
  ` >= 0))
```

Using the same example as shown in Figure 7.14, when user 2 tries to reserve a room, `total_reserved` becomes 101, which violates the `total_inventory` (100)– `total_reserved` (101) $\geq$ 0 constraint. The transaction is then rolled back.

# 16 - IMAGE UPLOAD SERVICE (WITHOUT DUPLICATES)

## FINAL DESIGN




## STEP 1: UNDERSTANDING & QUESTIONS

- FR
  - Simple portal where user uploads images
  - Million images
  - Should avoid duplication

- NON-FR
  - Scalable

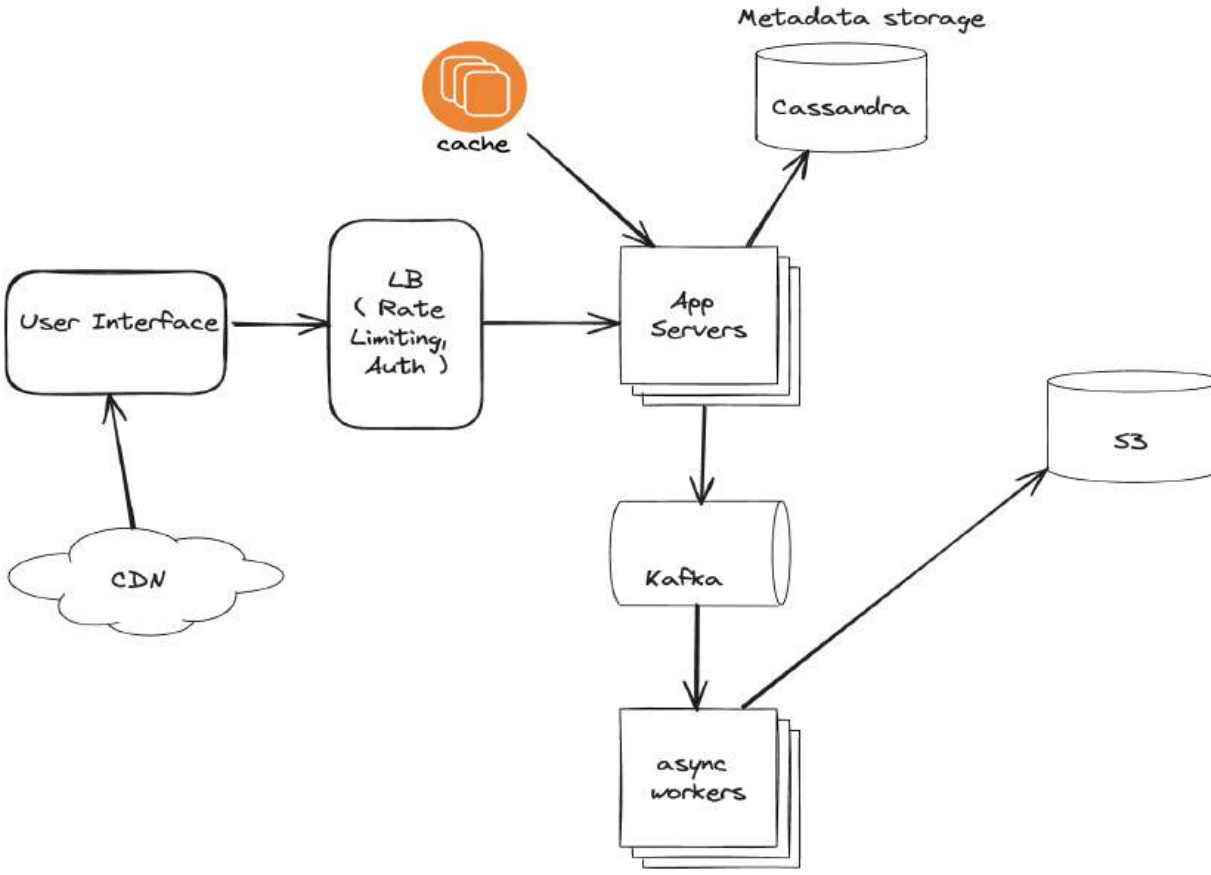
## STEP 2: RESOURCE ESTIMATION

- 10000 uploads per second
- Each image utmost 10 MB size

## STEP 3: DESIGN

- API DESIGN
  - POST
  - /api/v1/upload

- User uploads the image. Load Balancer receives the request
- It does the Rate Limiting and authentication at the LB Level itself before sending to App Servers
- Image size limit can be checked on App Server.
- Then, First step is to Hash the Image.
- There are multiple algorithms to hash the image such as pHash, aHash etc
- Then, look for this hash in the Redis cache. If not present in cache, then look for the same in Cassandra
- If not present in that DB as well, then add the Metadata such as image name, ID, hash etc to that DB (Cassandra in this case)
- Once that is done, App Server can raise an event and publish it in the queue. Image ID & Location are published in the queue
- There are async workers which constantly poll this queue and fetch the image using that location
- These are Image Processors which do the resizing, compression, sanity checks etc
- Once that is done, Image is uploaded to the S3 bucket
- We also have a CDN which can be used to serve static pages on the UI. In case of retrieving images if at all, we can also store the popular images in the CDN




# 17 - PROXIMITY SERVER

## FINAL DESIGN

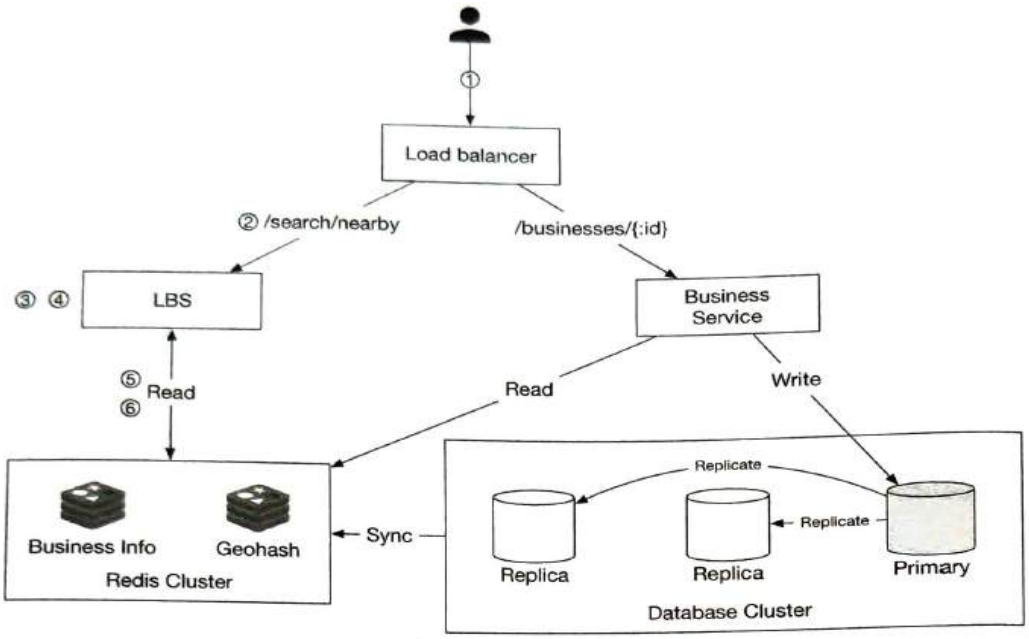



Figure 1.21: Design diagram

### Get nearby businesses

1. You try to find restaurants within 500 meters on Yelp. The client sends the user location (latitude = 37.776720, longitude = −122.416730) and radius (500m) to the load balancer.
2. The load balancer forwards the request to the LBS.
3. Based on the user location and radius info, the LBS finds the geohash length that matches the search. By checking Table 1.5, 500m map to geohash length = 6.
4. LBS calculates neighboring geohashes and adds them to the list. The result looks like this:
   `list_of_geohashes = [my_geohash, neighbor1_geohash, neighbor2_geohash, ..., neighbor8_geohash]`.
5. For each geohash in `list_of_geohashes`, LBS calls the "Geohash" Redis server to fetch corresponding business IDs. Calls to fetch business IDs for each geohash can be made in parallel to reduce latency.
6. Based on the list of business IDs returned, LBS fetches fully hydrated business information from the "Business info" Redis server, then calculates distances between a user and businesses, ranks them, and returns the result to the client.

## STEP 1: UNDERSTANDING & QUESTIONS

- Can a user specify radius?
- What's the maximum radius allowed?
- Can a user change the search radius in UI?
- What if the user keeps moving? Do we need to constantly change the results?

    - Let's assume we don't need to refresh continuously

- FR
    - Return all businesses in a radius
    - Business owners can add/delete their businesses
    - Customers can view detailed info about businesses
- NON-FR
    - Low Latency
    - High availability & Scalability

## STEP 2: RESOURCE ESTIMATION

- 100 MILLION Daily users
- 220 Million businesses
- 5 search queries per day
    - 100M * 5 / 10 power 5 seconds = 5000 QPS

## STEP 3: API & DATA MODEL

- *API DESIGN*

**GET /v1/search/nearby**

This endpoint returns businesses based on certain search criteria. In real-life applications, search results are usually paginated. Pagination [6] is not the focus of this chapter, but is worth mentioning during an interview.

Request Parameters:

| Field | Description | Type |
|---|---|---|
| latitude | Latitude of a given location | decimal |
| longitude | Longitude of a given location | decimal |
| radius | Optional. Default is 5000 meters (about 3 miles) | int |

Table 1.1: Request parameters

```
{
  "total": 10,
  "businesses":[{business object}]
}
```

### APIs for a business

The APIs related to a business object are shown in the table below.

| API | Detail |
|---|---|
| GET /v1/businesses/:id | Return detailed information about a business |
| POST /v1/businesses | Add a business |
| PUT /v1/businesses/:id | Update details of a business |
| DELETE /v1/businesses/:id | Delete a business |

Table 1.2: APIs for a business

- ***DATA MODEL***

- **READ RATIO is very high because of the following actions:**
  - **Search for nearby businesses**
  - **Viewed the detailed info of businesses**

- **WRITE VOLUME is LOW because business info is not updated so frequently**
- **For a READ-HEAVY system, MySQL can be a good fit**
- **Business table is as belows**

| business | |
|---|---|
| business_id | PK |
| address | |
| city | |
| state | |
| country | |
| latitude | |
| longtitude | |

Table 1.3: Business table

- 

- **Geo Index table is also present. For spatial operations**

Option 2: If there are multiple businesses in the same geohash, there will be multiple rows, one for each business. This means different business IDs within a geohash are stored in different rows.

| geospatial_index |
|---|
| geohash |
| business_id |

Table 1.10: `business_id` is a single ID

## STEP 4: HIGH-LEVEL DESIGN

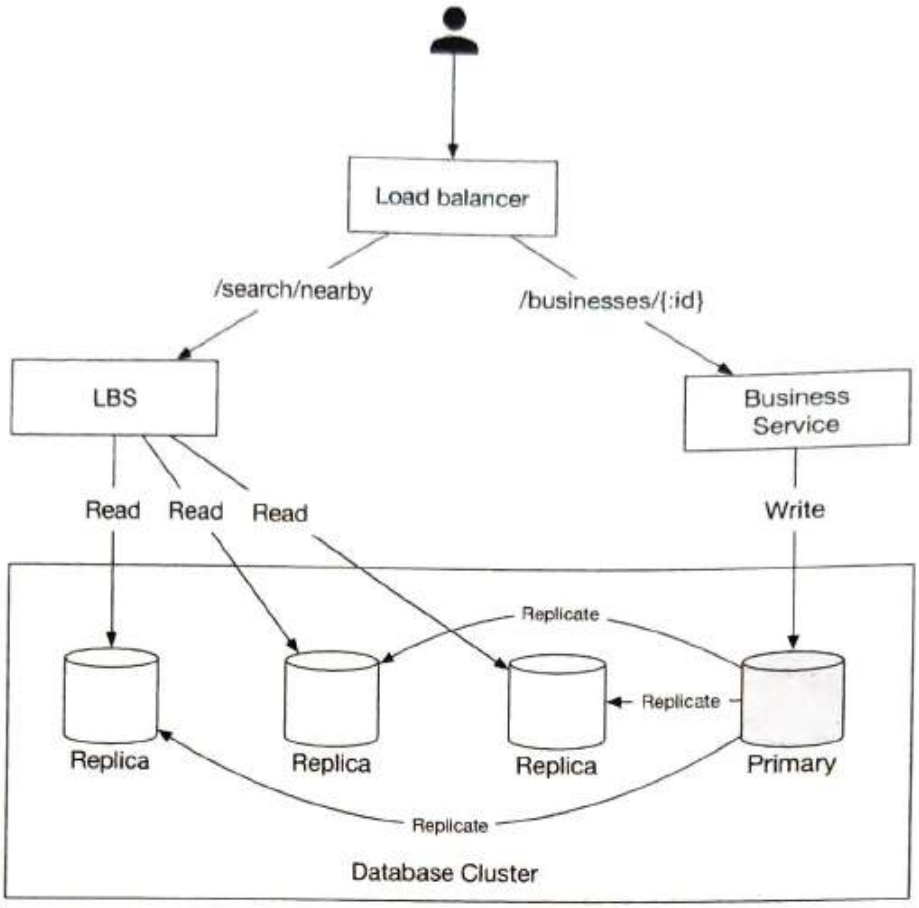



Figure 1.2: High-level design

- **Following is the Final Diagram**

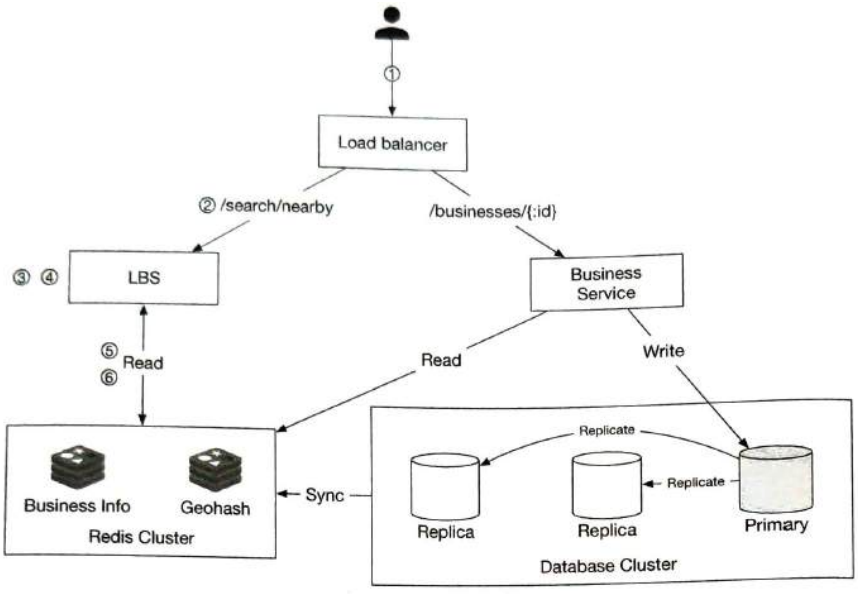



Figure 1.21: Design diagram

### Get nearby businesses

1. You try to find restaurants within 500 meters on Yelp. The client sends the user location (latitude = 37.776720, longitude = −122.416730) and radius (500m) to the load balancer.
2. The load balancer forwards the request to the LBS.
3. Based on the user location and radius info, the LBS finds the geohash length that matches the search. By checking Table 1.5, 500m map to geohash length = 6.
4. LBS calculates neighboring geohashes and adds them to the list. The result looks like this:
   `list_of_geohashes = [my_geohash, neighbor1_geohash, neighbor2_geohash, ..., neighbor8_geohash]`.
5. For each geohash in `list_of_geohashes`, LBS calls the "Geohash" Redis server to fetch corresponding business IDs. Calls to fetch business IDs for each geohash can be made in parallel to reduce latency.
6. Based on the list of business IDs returned, LBS fetches fully hydrated business information from the "Business info" Redis server, then calculates distances between a user and businesses, ranks them, and returns the result to the client.

### Location-based service (LBS)

The LBS service is the core part of the system which finds nearby businesses for a given radius and location. The LBS has the following characteristics:

- It is a read-heavy service with no write requests.
- QPS is high, especially during peak hours in dense areas.
- This service is stateless so it's easy to scale horizontally.

### Business service

Business service mainly deals with two types of requests:

- Business owners create, update, or delete businesses. Those requests are mainly write operations, and the QPS is not high.
- Customers view detailed information about a business. QPS is high during peak hours.

- **Primary DB handles all the write operations. Replicas serve the read operations**
- **Due to replication delay, there might be some discrepancy b/w data read by LBS and data written to the primary DB but it is not an issue as business info doesn't need to be updated**

- ***ALGORITHMS TO FETCH NEARBY BUSINESSES***

- **There are multiple algorithms to fetch the businesses nearby but the popular ones are Geohash, Quadtree etc**

In real life, companies might use existing geospatial databases such as Geohash in Redis [10] or Postgres with PostGIS extension [11]. You are not expected to know the internals of those geospatial databases during an interview. It's better to demonstrate your problem-solving skills and technical knowledge by explaining how the geospatial index works, rather than to simply throw out database names.

- Hash: even grid, geohash, cartesian tiers [12], etc.
- Tree: quadtree, Google S2, RTree [13], etc.

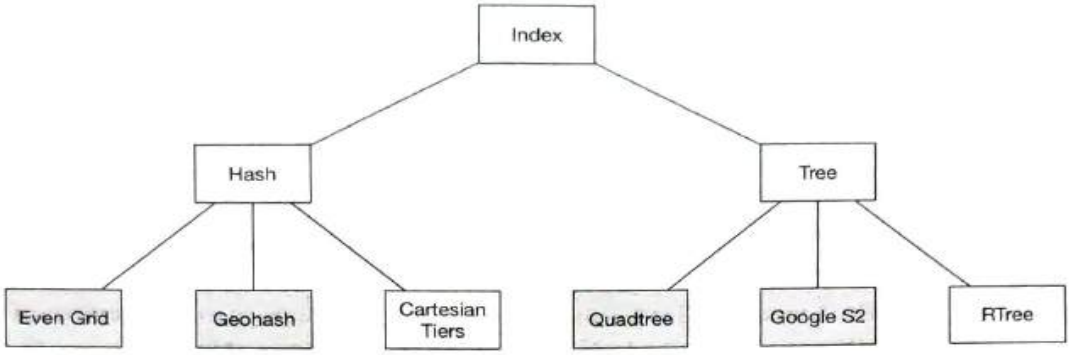



Figure 1.5: Different types of geospatial indexes

Even though the underlying implementations of those approaches are different, the high-level idea is the same, that is, **to divide the map into smaller areas and build indexes for fast search**. Among those, geohash, quadtree, and Google S2 are most widely used in real-world applications. Let's take a look at them one by one.

## Option 3: Geohash

Geohash is better than the evenly divided grid option. It works by reducing the two-dimensional longitude and latitude data into a one-dimensional string of letters and digits. Geohash algorithms work by recursively dividing the world into smaller and smaller grids with each additional bit. Let's go over how geohash works at a high level.

First, divide the planet into four quadrants along with the prime meridian and equator.

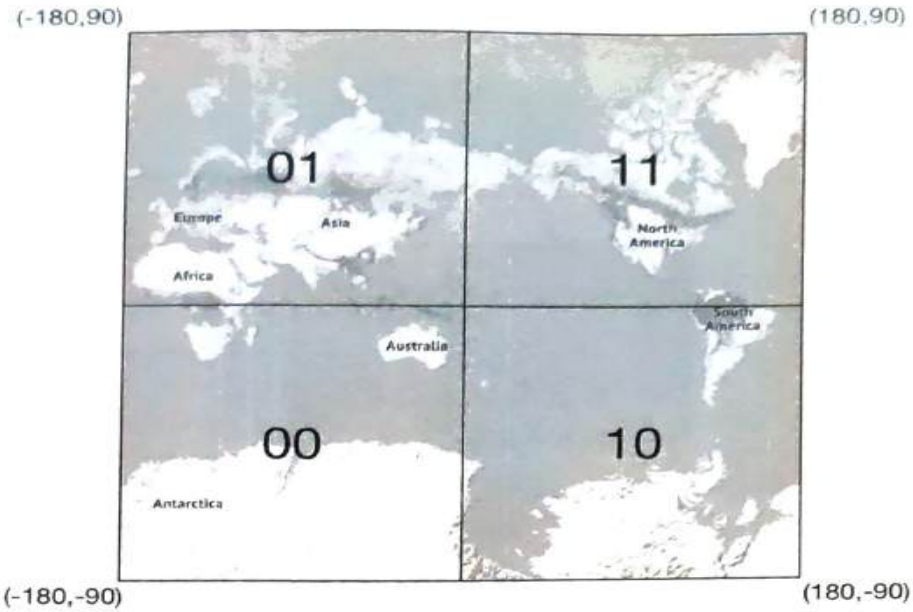



Figure 1.7: Geohash

- Latitude range $[-90, 0]$ is represented by $0$
- Latitude range $[0, 90]$ is represented by $1$
- Longitude range $[-180, 0]$ is represented by $0$
- Longitude range $[0, 180]$ is represented by $1$

Second, divide each grid into four smaller grids. Each grid can be represented by alternating between longitude bit and latitude bit.

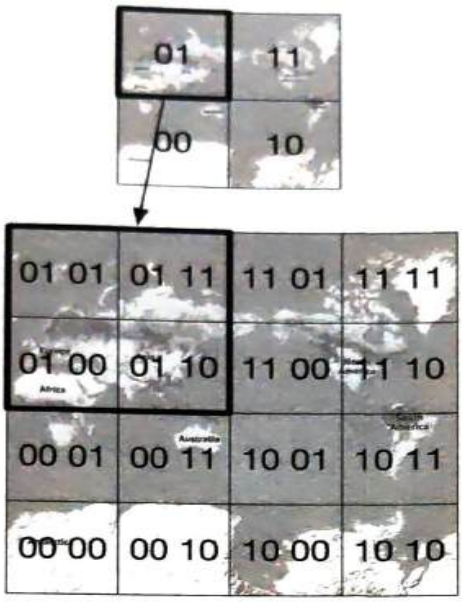



Figure 1.8: Divide grid

Repeat this subdivision until the grid size is within the precision desired. Geohash usually uses base32 representation [15]. Let's take a look at two examples.

- geohash of the Google headquarter (length = 6):
  `1001 10110 01001 10000 11011 11010` (base32 in binary) →
  **9q9hvu (base32)**
- geohash of the Facebook headquarter (length = 6):
  `1001 10110 01001 10001 10000 10111` (base32 in binary) →
  **9q9jhr (base32)**

Geohash has 12 precisions (also called levels) as shown in Table 1.4. The precision factor determines the size of the grid. We are only interested in geohashes with lengths between 4 and 6. This is because when it's longer than 6, the grid size is too small, while if it is smaller than 4, the grid size is too large (see Table 1.4).

| geohash length | Grid width × height |
|---|---|
| 1 | 5,009.4km × 4,992.6km (the size of the planet) |
| 2 | 1,252.3km × 624.1km |
| 3 | 156.5km × 156km |
| 4 | 39.1km × 19.5km |
| 5 | 4.9km × 4.9km |
| 6 | 1.2km × 609.4m |
| 7 | 152.9m × 152.4m |
| 8 | 38.2m × 19m |
| 9 | 4.8m × 4.8m |
| 10 | 1.2m × 59.5cm |
| 11 | 14.9cm × 14.9cm |
| 12 | 3.7cm × 1.9cm |

Table 1.4: Geohash length to grid size mapping (source: [16])

How do we choose the right precision? We want to find the minimal geohash length that covers the whole circle drawn by the user-defined radius. The corresponding relationship between the radius and the length of geohash is shown in the table below.

| Radius (Kilometers) | Geohash length |
|---|---|
| 0.5km (0.31 mile) | 6 |
| 1km (0.62 mile) | 5 |
| 2km (1.24 mile) | 5 |
| 5km (3.1 mile) | 4 |
| 20km (12.42 mile) | 4 |

Table 1.5: Radius to geohash mapping

This approach works great most of the time, but there are some edge cases with how the geohash boundary is handled that we should discuss with the interviewer.

### Boundary issues

Geohashing guarantees that the longer a shared prefix is between two geohashes, the closer they are. As shown in Figure 1.9, all the grids have a shared prefix: `9q8zn`.

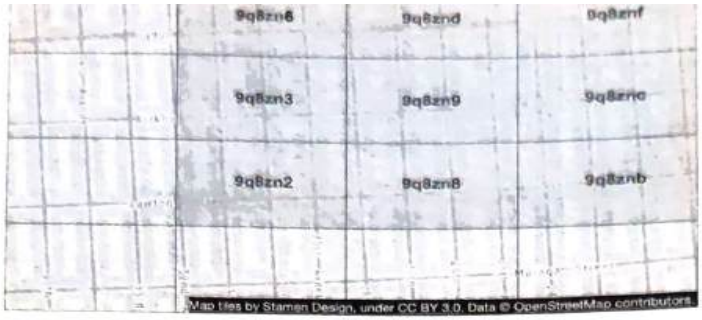



Figure 1.9: Shared prefix

### Boundary issue 1

However, the reverse is not true: two locations can be very close but have no shared prefix at all. This is because two close locations on either side of the equator or prime meridian belong to different "halves" of the world. For example, in France, La Roche-Chalais (geohash: `u000`) is just 30km from Pomerol (geohash: `ezzz`) but their geohashes have no shared prefix at all [17].

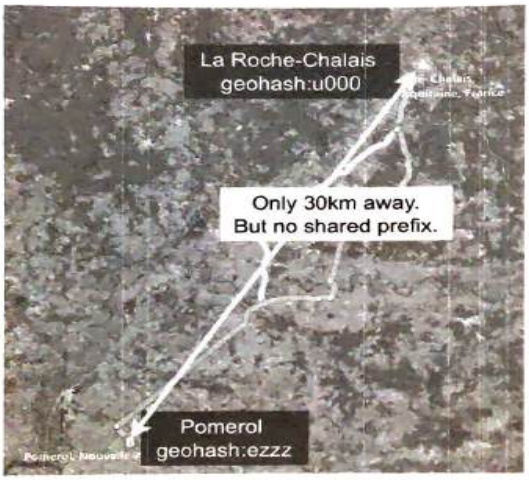


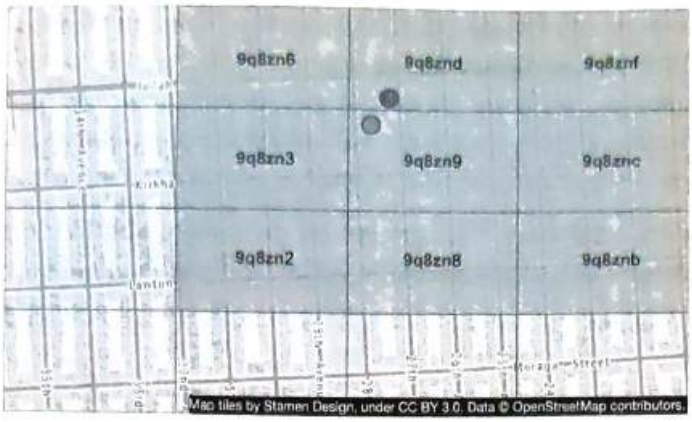

Figure 1.10: No shared prefix

Because of this boundary issue, a simple prefix SQL query below would fail to fetch all nearby businesses.

```
SELECT * FROM geohash_index WHERE geohash LIKE '9q8zn%'
```

### Boundary issue 2

Another boundary issue is that two positions can have a long shared prefix, but they belong to different geohashes as shown in Figure 1.11.




## STEP 5: DESIGN DEEP DIVE

- Database Scaling
- Caching
- Region & Availability Zones

- *Database Scaling*

- Business Table
  - It will have a lot of entries so it can be sharded using Business ID
- Geospatial Index table

  Option 2: If there are multiple businesses in the same geohash, there will be multiple rows, one for each business. This means different business IDs within a geohash are stored in different rows.

  | geospatial_index |
  | --- |
  | geohash |
  | business_id |

  Table 1.10: `business_id` is a single ID

- For the Geospatial Index table, sharding might not be the best idea because it is complex to implement. It has very small data that can be handled. Although we want

it to be distributed among multiple servers, we can simply do it by maintaining read replicas which is easier to implement

- ***Caching***

### Types of data to cache

As shown in Table 1.12, there are two types of data that can be cached to improve the overall performance of the system:

| Key | Value |
|---|---|
| geohash | List of business IDs in the grid |
| business_id | Business object |

Table 1.12: Key-value pairs in cache

### List of business IDs in a grid

Since business data is relatively stable, we precompute the list of business IDs for a given geohash and store it in a key-value store such as Redis. Let's take a look at a concrete example of getting nearby businesses with caching enabled.

1. Get the list of business IDs for a given geohash.

```
SELECT business_id FROM geohash_index WHERE geohash LIKE `{:
geohash}%`
```

2. Store the result in the Redis cache if cache misses.

```
public List<String> getNearbyBusinessIds(String geohash) {
  String cacheKey = hash(geohash);
  List<string> listOfBusinessIds = Redis.get(cacheKey);
  if (listOfBusinessIDs == null) {
    listOfBusinessIds = Run the select SQL query above;
    Cache.set(cacheKey, listOfBusinessIds, "1d");
  }
  return listOfBusinessIds;
}
```

When a new business is added, edited, or deleted, the database is updated and the cache invalidated. Since the volume of those operations is relatively small and no locking mechanism is needed for the geohash approach, update operations are easy to deal with.

According to the requirements, a user can choose the following 4 radii on the client: 500m, 1km, 2km, and 5km. Those radii are mapped to geohash lengths of 4, 5, 5, and 6, respectively. To quickly fetch nearby businesses for different radii, we cache data in Redis on all three precisions (geohash_4, geohash_5, and geohash_6).

As mentioned earlier, we have 200 million businesses and each business belongs to 1 grid in a given precision. Therefore the total memory required is:

- Storage for Redis values: 8 bytes × 200 million × 3 precisions = ~ 5GB
- Storage for Redis keys: negligible

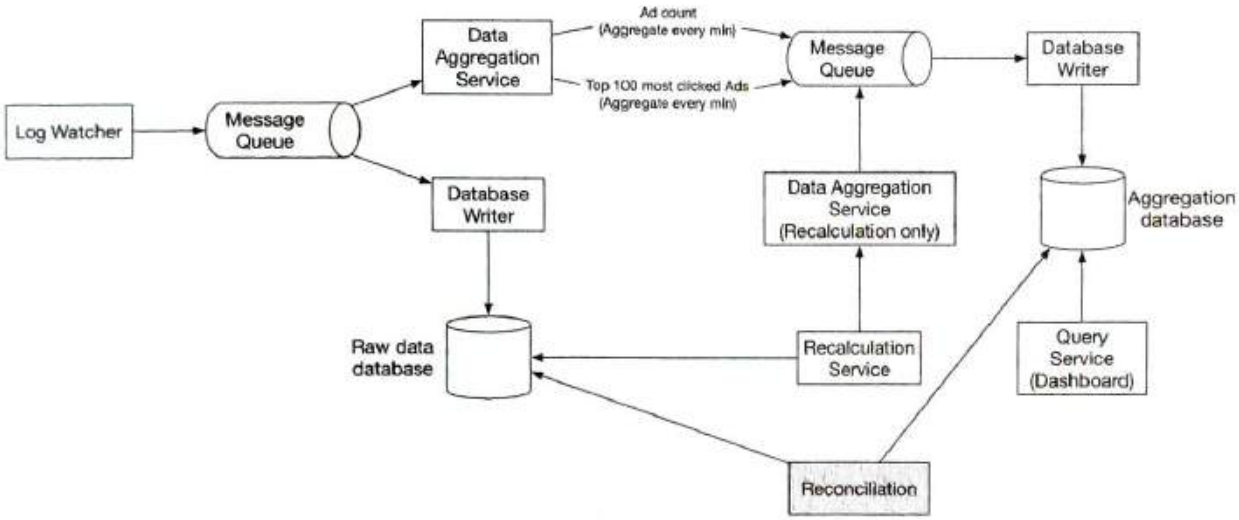


- Total memory required: ~ 5GB

We can get away with one modern Redis server from the memory usage perspective, but to ensure high availability and reduce cross continent latency, we deploy the Redis cluster across the globe. Given the estimated data size, we can have the same copy of cache data deployed globally. We call this Redis cache "Geohash" in our final architecture diagram

# 18 - AD CLICK AGGREGATOR

- ## FINAL DESIGN DIAGRAM




  - 

- 

NoSQL like Cassandra for both raw and aggregated data (because write heavy and then aggregate operations)

## STEP 1: UNDERSTANDING & QUESTIONS

  - What is the format of the input data?
    - Log file located in different servers and latest click events are appended to it
  - What is the data volume?
    - 1 billion ad clicks per day

        - 2 million ads per day
    - What are important queries to support?
    - Any edge cases that need to be considered?
    - FR
        - No. of clicks for AD_ID in the last M minutes
        - Return top 100 most clicked AD_ID in every minute
    - NON-FR
        - Handle delayed or duplicate events properly
        - Robustness
- 

## STEP 2: CAPACITY ESTIMATION

    - 1 billion ad clicks
        - 1 billion / 100000 seconds = 10,000 QPS
    - Peak QPS = 5 * 10,000 = 50,000 QPS
    - Each ad is of 0.1KB storage
        - 1 billion * 0.1 KB = 100 GB per day
        - 100 GB * 30 = 30 TB per month
- 

## STEP 3: API DESIGN & DATA MODEL

**API 1: Aggregate the number of clicks of `ad_id` in the last $M$ minutes.**

| API | Detail |
|---|---|
| `GET /v1/ads/{:ad_id}/aggregated_count` | Return aggregated event count for a given `ad_id` |

Table 6.1: API for aggregating the number of clicks

Request parameters are:



| Field | Description | Type |
|---|---|---|
| `from` | Start minute (default is now minus 1 minute) | long |
| `to` | End minute (default is now) | long |
| `filter` | An identifier for different filtering strategies. For example, `filter = 001` filters out non-US clicks | long |

Table 6.2: Request parameters for `/v1/ads/{:ad_id}/aggregated_count`

Response:

| Field | Description | Type |
|---|---|---|
| `ad_id` | The identifier of the ad | string |
| `count` | The aggregated count between the start and end minutes | long |

Table 6.3: Response for `/v1/ads/{:ad_id}/aggregated_count`

**API 2: Return top $N$ most clicked `ad_ids` in the last $M$ minutes**

| API | Detail |
|---|---|
| `GET /v1/ads/popular_ads` | Return top $N$ most clicked ads in the last $M$ minutes |

Table 6.4: API for `/v1/ads/popular_ads`

Request parameters are:

| Field | Description | Type |
|---|---|---|
| `count` | Top $N$ most clicked ads | integer |
| `window` | The aggregation window size ($M$) in minutes | integer |
| `filter` | An identifier for different filtering strategies | long |

Table 6.5: Request parameters for `/v1/ads/popular_ads`

Response:

| Field | Description | Type |
|---|---|---|
| `ad_ids` | A list of the most clicked ads | array |

Table 6.6: Response for `/v1/ads/popular_ads`

- 

  **DATA MODEL**

  - **NoSQL like Cassandra for both raw and aggregated data (because write heavy and then aggregate operations)**
- 

  - **RAW DATA**
- 

  | ad_id | click_timestamp | user_id | ip | country |
  |---|---|---|---|---|
  | ad001 | 2021-01-01 00:00:01 | user1 | 207.148.22.22 | USA |
  | ad001 | 2021-01-01 00:00:02 | user1 | 207.148.22.22 | USA |
  | ad002 | 2021-01-01 00:00:02 | user2 | 209.153.56.11 | USA |

  Table 6.7: Raw data
- 

  - **AGGREGATED DATA**
- 

| ad_id | click_minute | filter_id | count |
|---|---|---|---|
| ad001 | 202101010000 | 0012 | 2 |
| ad001 | 202101010000 | 0023 | 3 |
| ad001 | 202101010001 | 0012 | 1 |
| ad001 | 202101010001 | 0023 | 6 |

Table 6.9: Aggregated data with filters

| filter_id | region | ip | user_id |
|---|---|---|---|
| 0012 | US | 0012 | * |
| 0013 | * | 0023 | 123.1.2.3 |

Table 6.10: Filter table

- Filter is for regions.

- Should store both Raw data and Aggregated data

## STEP 4: HIGH-LEVEL DESIGN

- ***DATABASE CHOICES***
  - RAW DATA

    As shown in the back of the envelope estimation, the average write QPS is 10,000, the peak QPS can be 50,000, so the system is write-heavy. On the read side, raw da used as backup and a source for recalculation, so in theory, the read volume is low.

    Relational databases can do the job, but scaling the write can be challenging. No databases like Cassandra and InfluxDB are more suitable because they are optimize write and time-range queries.

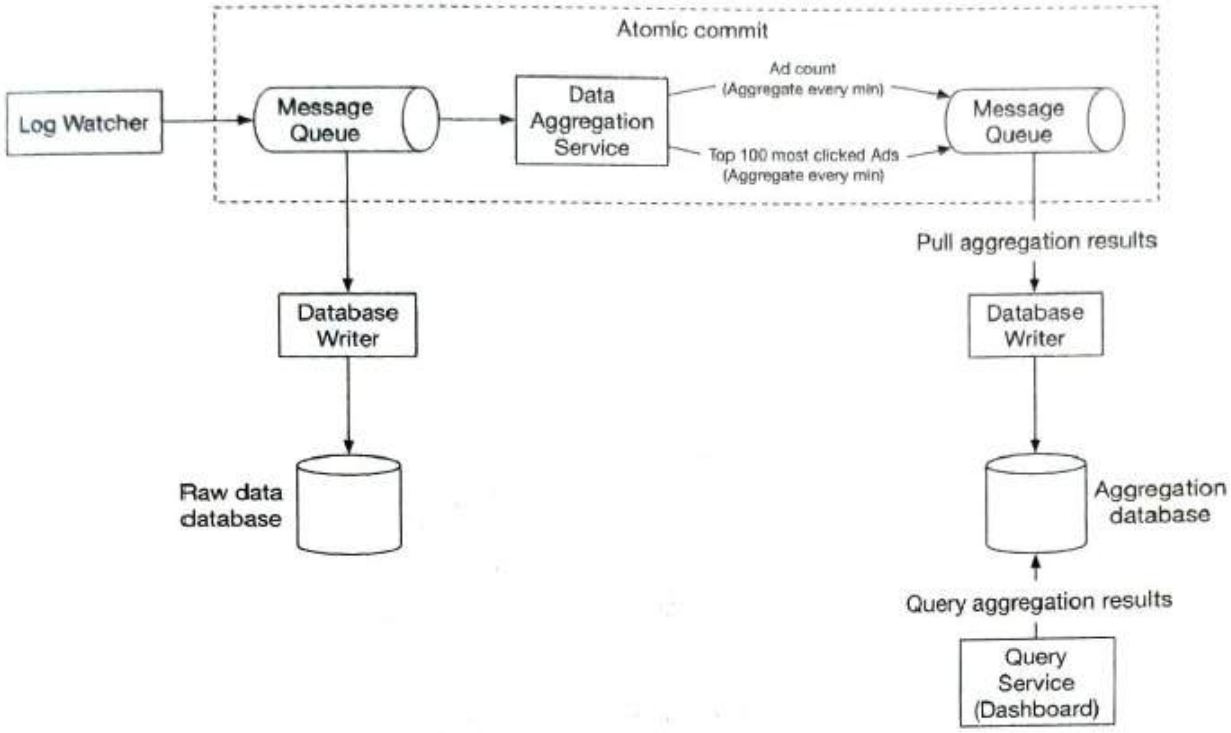
- 
  - **AGGREGATED DATA**
    - For aggregated data, it is time-series in nature and the workflow is both read and write heavy. This is because, for each ad, we need to query the database every minute to display the latest aggregation count for customers. This feature is useful for auto-refreshing the dashboard or triggering alerts in a timely manner. Since there are two million ads in total, the workflow is read-heavy. Data is aggregated and written every minute by the aggregation service, so it's write-heavy as well. We could use the same type of database to store both raw data and aggregated data.
- 
  - **DESIGN**
- 
  - 

    Figure 6.4: End-to-end exactly once
- 
  - **Queues are used to do asynchronous processing and the data in them is as follows**

- 
  - What is stored in the first message queue? It contains ad click event data as shown in Table 6.13.

| ad_id | click_timestamp | user_id | ip | country |
|---|---|---|---|---|

Table 6.13: Data in the first message queue

- 
  - 1. Ad click counts aggregated at per-minute granularity.

| ad_id | click_minute | count |
|---|---|---|

Table 6.14: Data in the second message queue

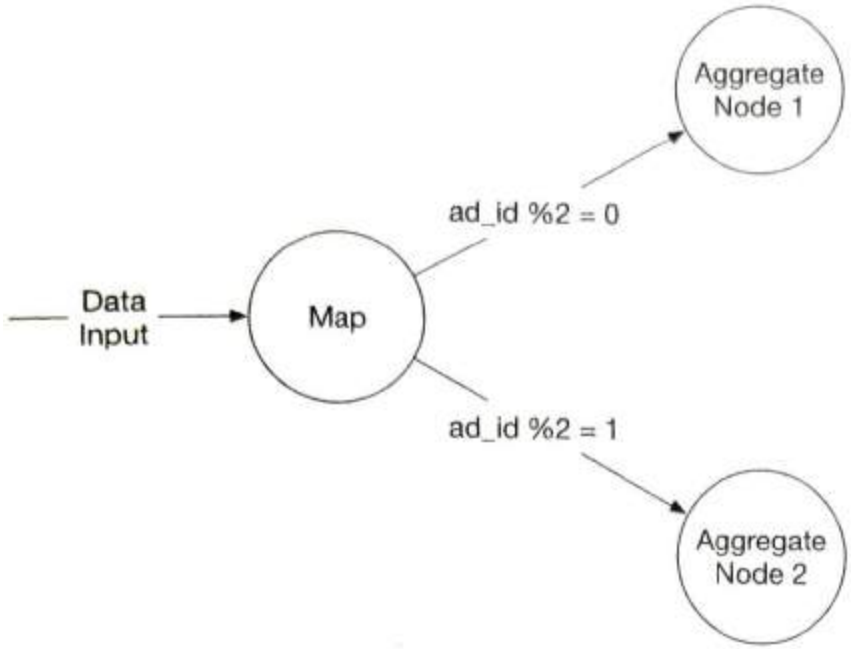


  - 2. Top $N$ most clicked ads aggregated at per-minute granularity.

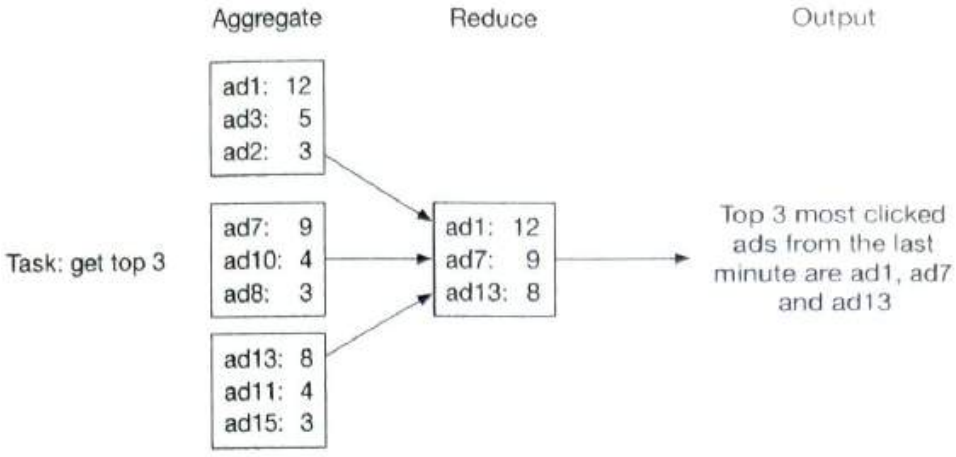
| update_time_minute | most_clicked_ads |
|---|---|

Table 6.15: Data in the second message queue

- 
  - ***AGGREGATION SERVICE***
    - MapReduce Framework is used to aggregate ad click events
    - MAP NODE
      - Map node reads data from a source and then filters and transforms the data




- AGGREGATE NODE
  - Counts the ad clicks by AD_ID in memory every time.
- REDUCE NODE
  - Reduces the aggregated results from all Aggregated nodes to final result




## STEP 5: DESIGN DEEP DIVE

- ***DATA RECALCULATION***
  - **Sometimes recalculation is needed if any errors are encountered**

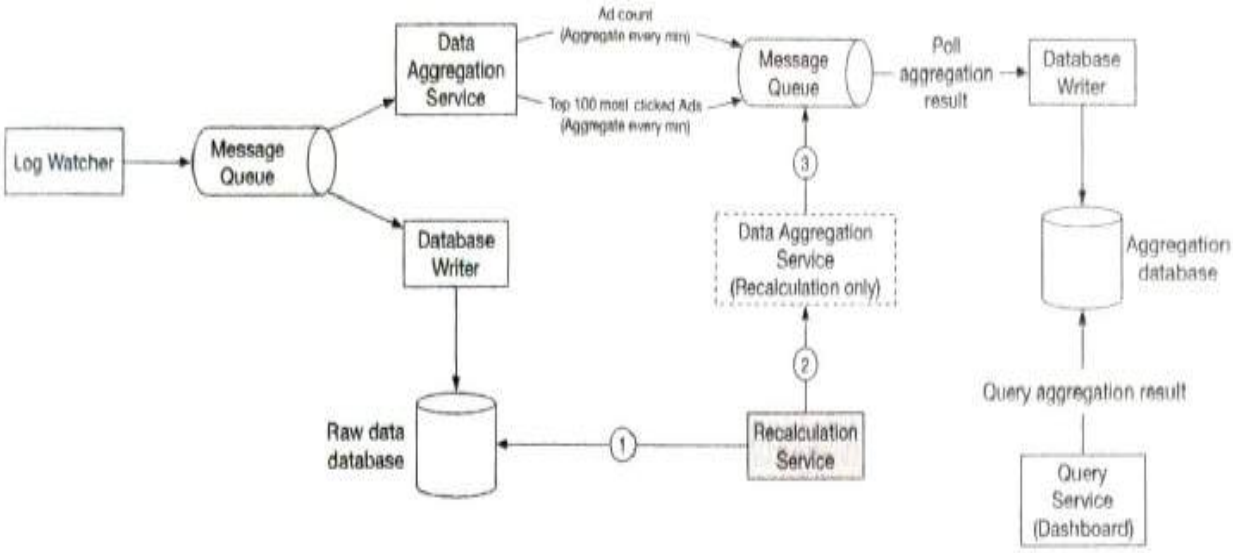



Figure 6.11: Recalculation service

1. The recalculation service retrieves data from raw data storage. This is a batched job.
2. Retrieved data is sent to a dedicated aggregation service so that the real-time processing is not impacted by historical data replay.
3. Aggregated results are sent to the second message queue, then updated in the aggregation database.

- ***SCALE THE SYSTEM***
  - **We can scale the Kafka system**

When there are hundreds of Kafka consumers in the system, consumer rebalance can be quite slow and could take a few minutes or even more. Therefore, if more consumers need to be added, try to do it during off-peak hours to minimize the impact.

### Brokers

- **Hashing key**
  Using `ad_id` as hashing key for Kafka partition to store events from the same `ad_id` in the same Kafka partition. In this case, an aggregation service can subscribe to all events of the same `ad_id` from one single partition.
- **The number of partitions**
  If the number of partitions changes, events of the same `ad_id` might be mapped to a different partition. Therefore, it's recommended to pre-allocate enough partitions in advance, to avoid dynamically increasing the number of partitions in production.
- **Topic physical sharding**
  One single topic is usually not enough. We can split the data by geography (`topic_north_america`, `topic_europe`, `topic_asia`, etc.) or by business type (`topic_web_ads`, `topic_mobile_ads`, etc).
  - Pros: Slicing data to different topics can help increase the system throughput. With fewer consumers for a single topic, the time to rebalance consumer groups is reduced.
  - Cons: It introduces extra complexity and increases maintenance costs.

- 
  - ***FAULT TOLERANCE***
    - **When aggregated node is down, we can rebuild the count by saving the system status using snapshot**

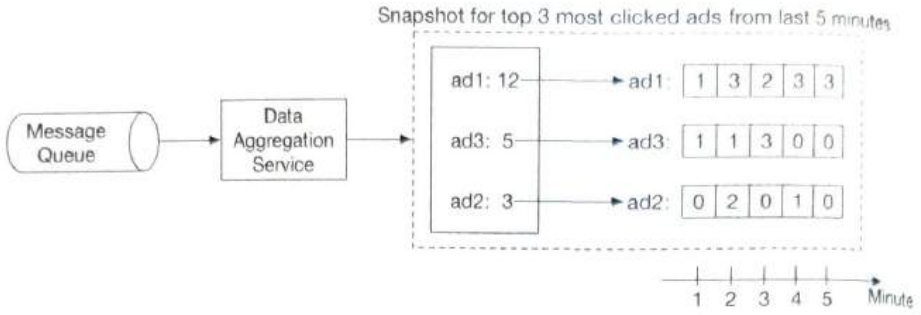



Figure 6.26: Data in a snapshot

With a snapshot, the failover process of the aggregation service is quite simple. If one aggregation service node fails, we bring up a new node and recover data from the latest snapshot (Figure 6.27). If there are new events that arrive after the last snapshot was taken, the new aggregation node will pull those data from the Kafka broker for replay.

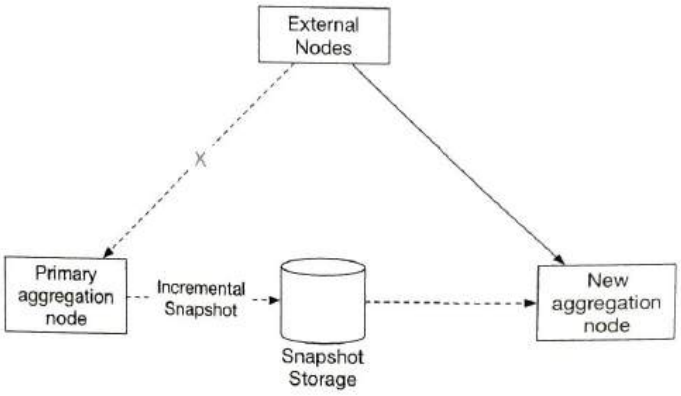



Figure 6.27: Aggregation node failover

- 
- 
  - ***RECONCILIATION***
- 

### Reconciliation

Reconciliation means comparing different sets of data in order to ensure data integrity. Unlike reconciliation in the banking industry, where you can compare your records with the bank's records, the result of ad click aggregation has no third-party result to reconcile with.

What we can do is to sort the ad click events by event time in every partition at the end of the day, by using a batch job and reconciling with the real-time aggregation result. If we have higher accuracy requirements, we can use a smaller aggregation window; for example, one hour. Please note, no matter which aggregation window is used, the result from the batch job might not match exactly with the real-time aggregation result, since some events might arrive late (see "Time" section on page 175).

- 

# 19 - GAMING LEADERBOARD

## FINAL DESIGN

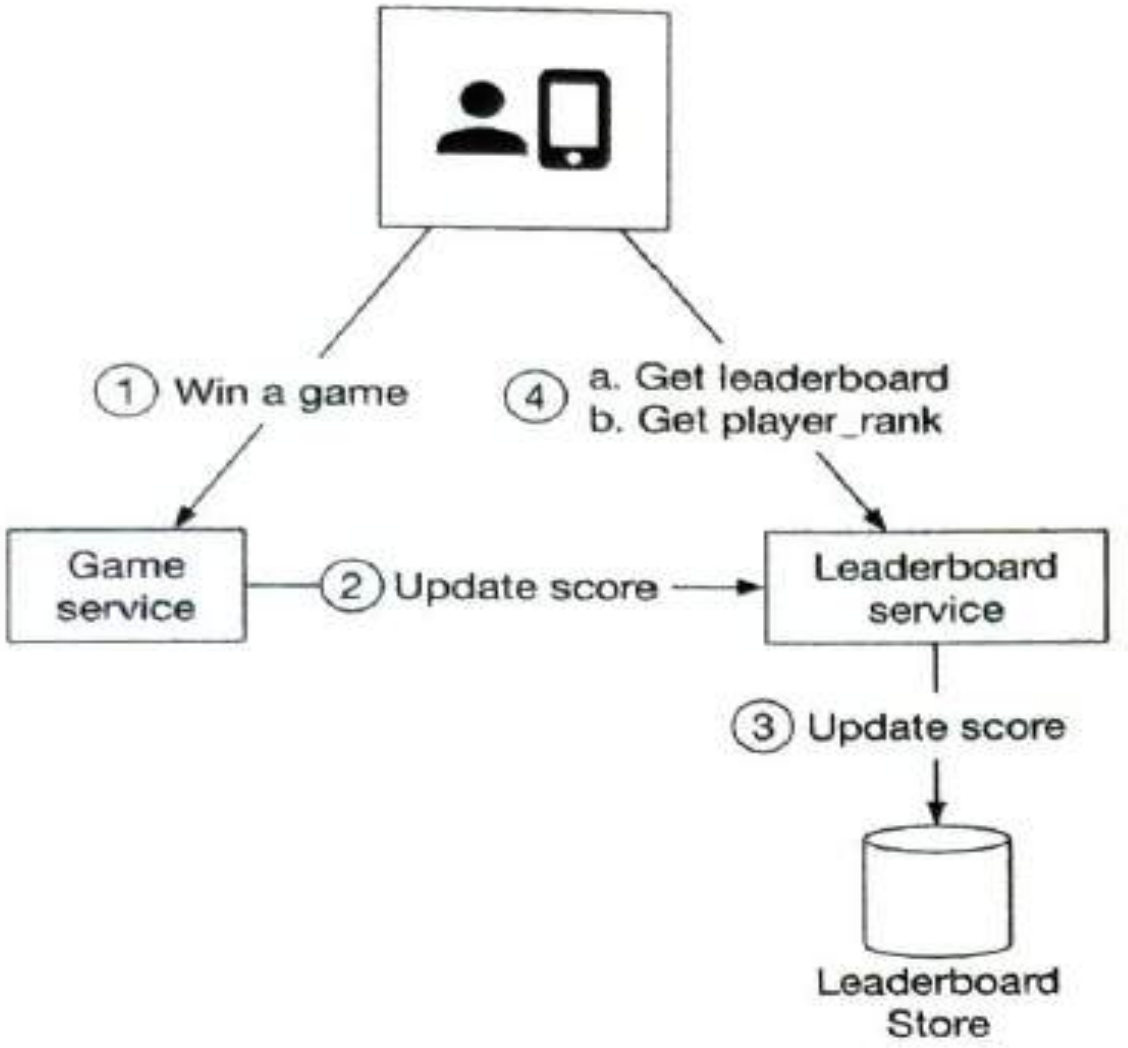



Figure 10.2: High-level design

1. When a player wins a game, the client sends a request to the game service.
2. The game service ensures the win is valid and calls the leaderboard service to update the score.
3. The leaderboard service updates the user's score in the leaderboard store.
4. A player makes a call to the leaderboard service directly to fetch leaderboard data, including:
   (a) top 10 leaderboard.
   (b) the rank of the player on the leaderboard.

## STEP 1: UNDERSTANDING & QUESTIONS

- How is the score calculated?
    - 1 point for each win
- Are all the players included in leaderboard?
    - Yes
- Tournament span?
    - Each month, a new tournament starts
- How many users in tournament?
    - 5 Million DAU & 25 Million Monthly active users
- How many matches on average?
    - Each player plays 10 matches a day
- *FR*
    - Display 10 players on board
    - Show user's specific rank
- *NON-FR*
    - Updates on board
    - Availability and Scalability

## STEP 2: CAPACITY ESTIMATION

- 5 MILLION users per day. 10 games per user
    - 5 Million * 10 / 100000 seconds = 500 QPS
    - Peak QPS = 500 * 5 = 2,500 QPS

## STEP 3: API & DATA MODEL

- POST /v1/scores (userid, points)
    - Response : 200

- GET /v1/scores
    - Fetches top 10 players

- GET /v1/scores/{user_id}
    - Gets score of that id

## STEP 4: HIGH-LEVEL DESIGN

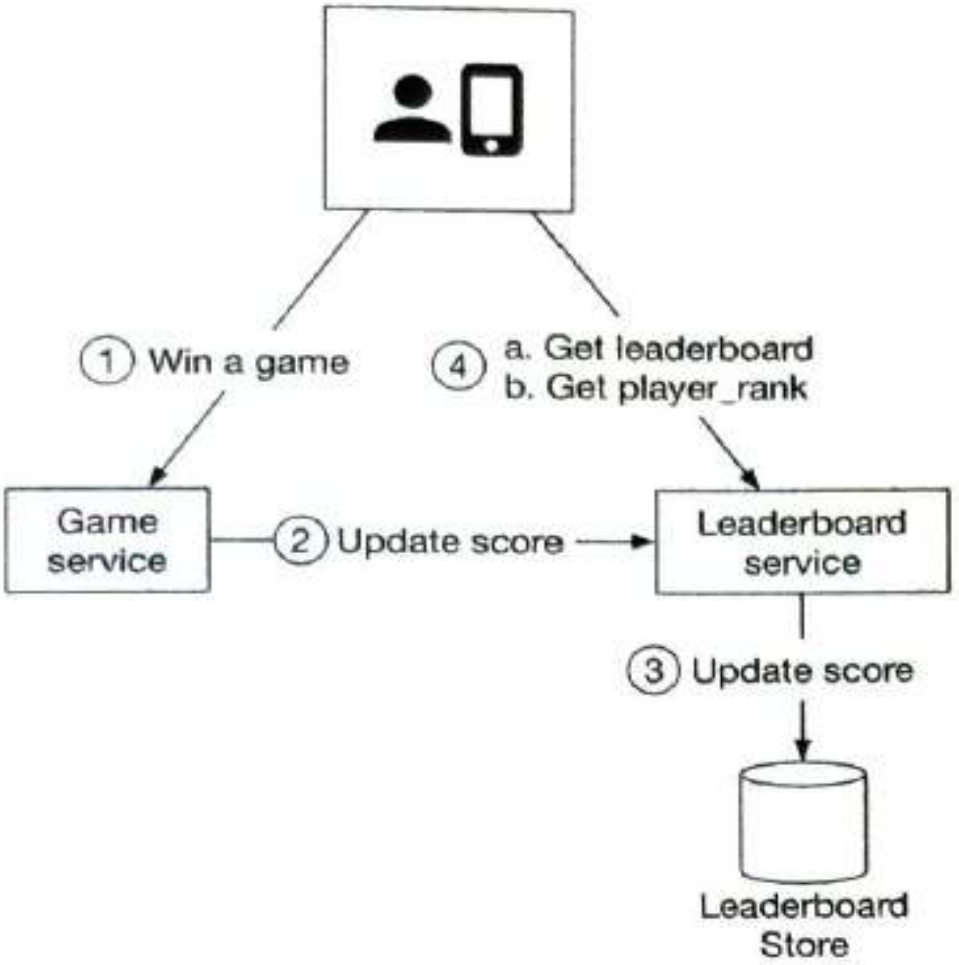



Figure 10.2: High-level design

1. When a player wins a game, the client sends a request to the game service.
2. The game service ensures the win is valid and calls the leaderboard service to update the score.
3. The leaderboard service updates the user's score in the leaderboard store.
4. A player makes a call to the leaderboard service directly to fetch leaderboard data, including:
   (a) top 10 leaderboard.
   (b) the rank of the player on the leaderboard.

## STEP 5: DEEP DIVE

- ***LEADERBOARD DATA STORE***

- ***RDBMS***

- **RDMS is not a great choice when there is a large volume of members because in order to get the rank, we have to sort all the rows and then fetch a user to know his rank or in-order to fetch top k members**

- To figure out the rank of a user, we need to sort every single player into their correct spot on the leaderboard so we can determine exactly what the correct rank is. Remember that there can be duplicate scores as well, so the rank isn't just the position of the user in the list.

  SQL databases are not performant when we have to process large amounts of continuously changing information. Attempting to do a rank operation over millions of rows is going to take 10s of seconds, which is not acceptable for the desired real-time approach. Since the data is constantly changing, it is also not feasible to consider a cache.

  A relational database is not designed to handle the high load of read queries this implementation would require. An RDS could be used successfully if done as a batch operation, but that would not align with the requirement to return a real-time position for the user

- ***REDIS SOLUTION***

- **Most suitable solution**
- **Consists of sorted sets inherently**
- **Sorted sets are implemented by hash table and skip list**

### Implementation using Redis sorted sets

Now that we know sorted sets are fast, let's take a look at the Redis operations we will use to build our leaderboard [4] [5] [6] [7]:

- `ZADD`: insert the user into the set if they don't yet exist. Otherwise, update the score for the user. It takes $O(\log(n))$ to execute.
- `ZINCRBY`: increment the score of the user by the specified increment. If the user doesn't exist in the set, then it assumes the score starts at 0. It takes $O(\log(n))$ to execute.
- `ZRANGE/ZREVRANGE`: fetch a range of users sorted by the score. We can specify the order (range vs. revrange), the number of entries, and the position to start from. This takes $O(\log(n)+m)$ to execute, where $m$ is the number of entries to fetch (which is usually small in our case), and $n$ is the number of entries in the sorted set.
- `ZRANK/ZREVRANK`: fetch the position of any user sorting in ascending/descending order in logarithmic time.

### Storage requirement

At a minimum, we need to store the user id and score. The worst-case scenario is that all 25 million monthly active users have won at least one game, and they all have entries in the leaderboard for the month. Assuming the id is a 24-character string and the score is a 16-bit integer (or 2 bytes), we need 26 bytes of storage per leaderboard entry. Given the worst-case scenario of one leaderboard entry per MAU, we would need 26 bytes $\times 25$ million $=$ 650 million bytes or $\sim$ 650MB for leaderboard storage in the Redis cache. Even if we double the memory usage to account for the overhead of the skip list and the hash for the sorted set, one modern Redis server is more than enough to hold the data.

Another related factor to consider is CPU and I/O usage. Our peak QPS from the back-of-the-envelope estimation is 2500 updates/sec. This is well within the performance envelope of a single Redis server.

One concern about the Redis cache is persistence, as a Redis node might fail. Luckily, Redis does support persistence, but restarting a large Redis instance from disk is slow. Usually, Redis is configured with a read replica, and when the main instance fails, the read replica is promoted, and a new read replica is attached.



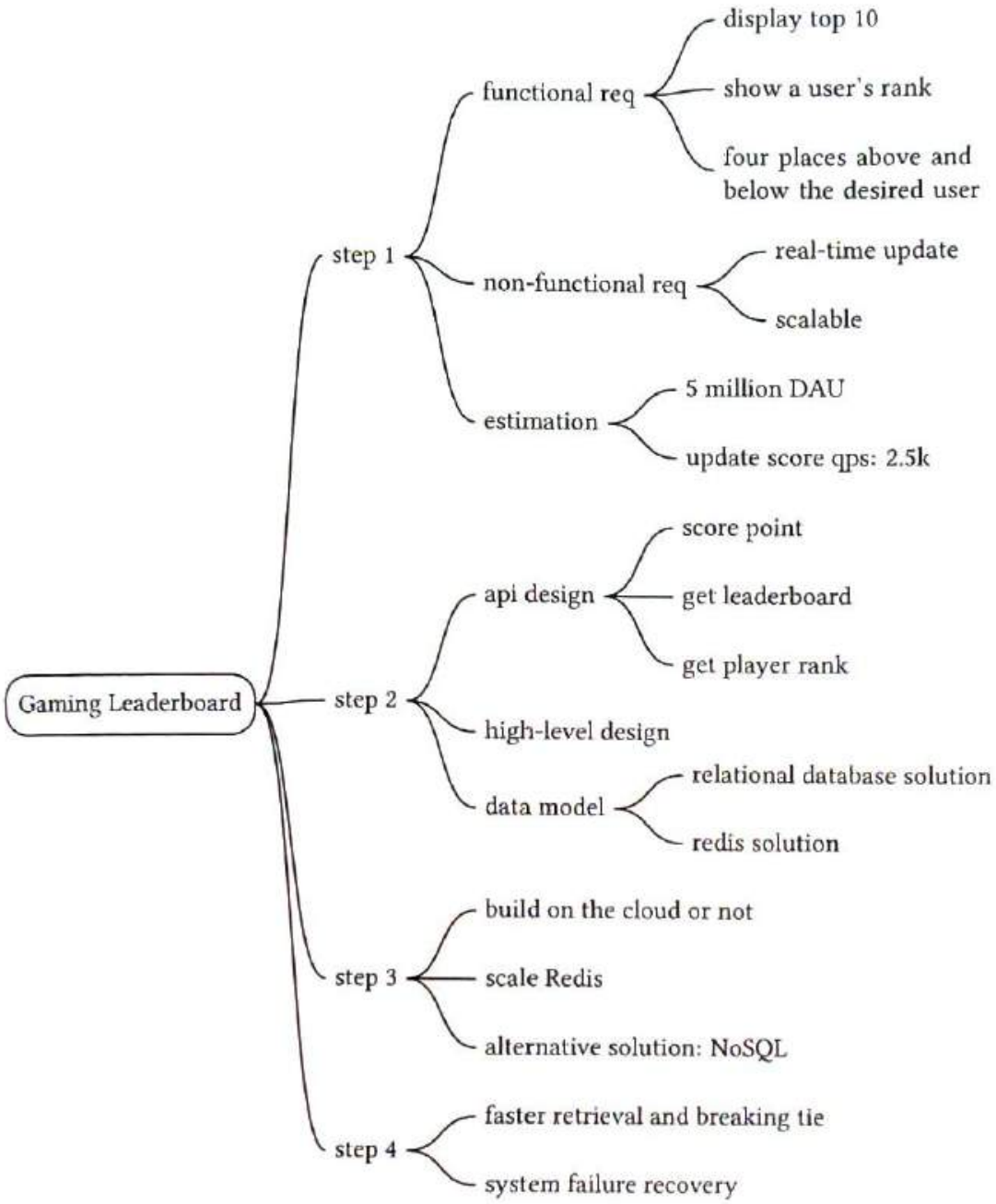
Besides, we need to have 2 supporting tables (user and point) in a relational database like MySQL. The user table would store the user ID and user's display name (in a real-world application, this would contain a lot more data). The point table would contain the user id, score, and timestamp when they won a game. This can be leveraged for other game functions such as play history, and can also be used to recreate the Redis leaderboard in the event of an infrastructure failure.

As a small performance optimization, it may make sense to create an additional cache of the user details, potentially for the top 10 players since they are retrieved most frequently. However, this doesn't amount to a large amount of data.

- 

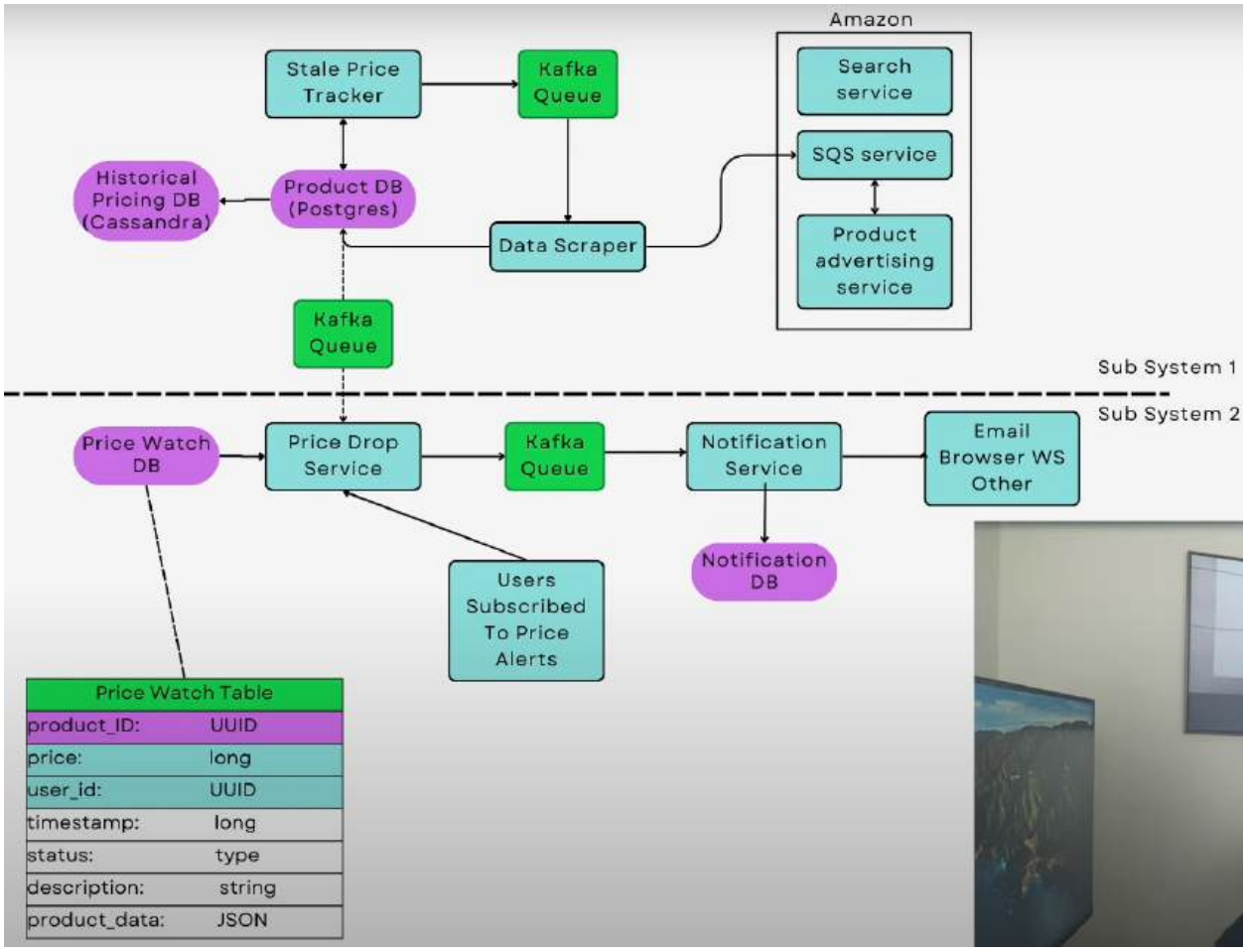
## STEP 6: SUMMARY

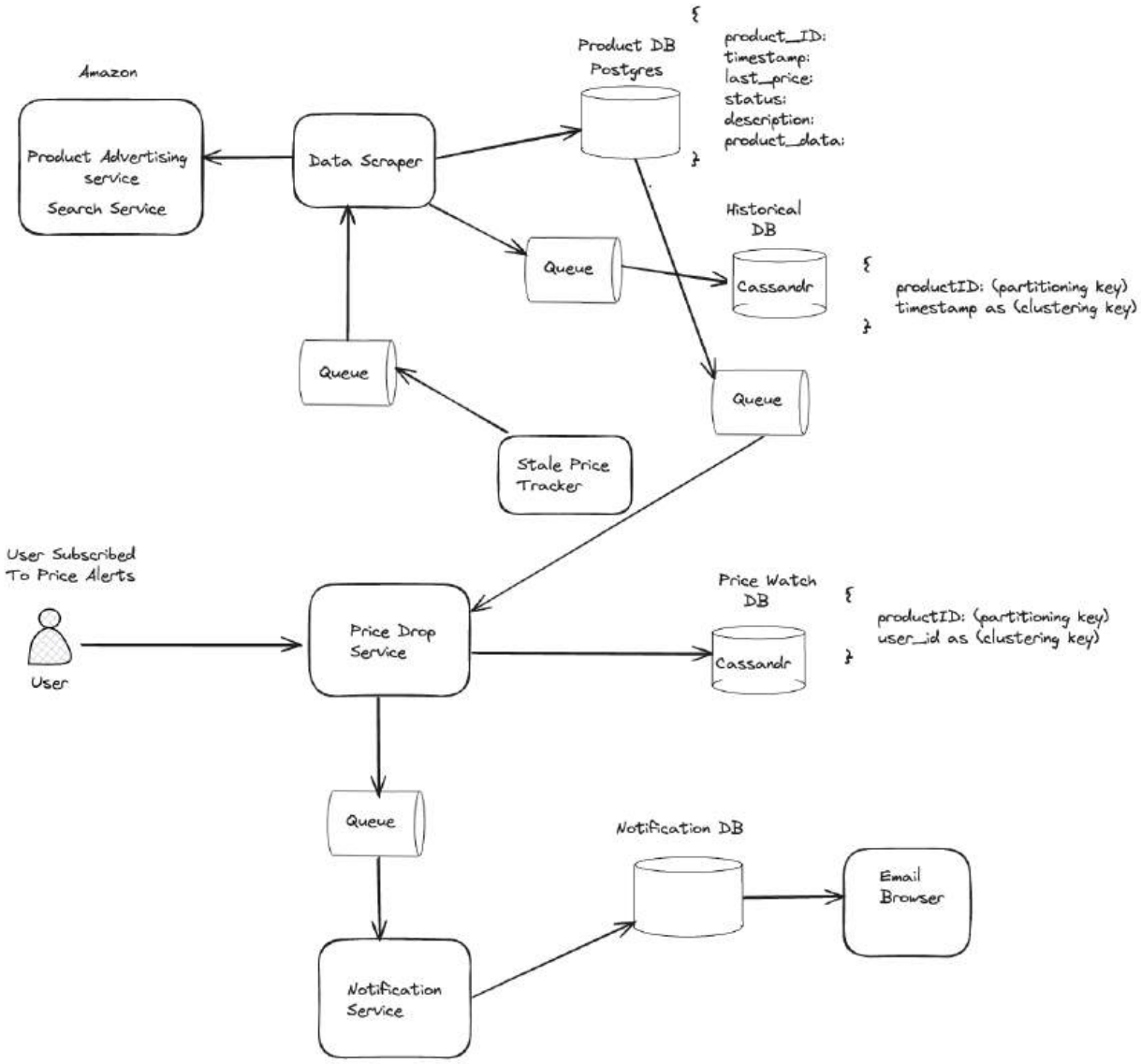
## Chapter Summary




# 20 - PRICE DROP TRACKER

## FINAL DESIGN

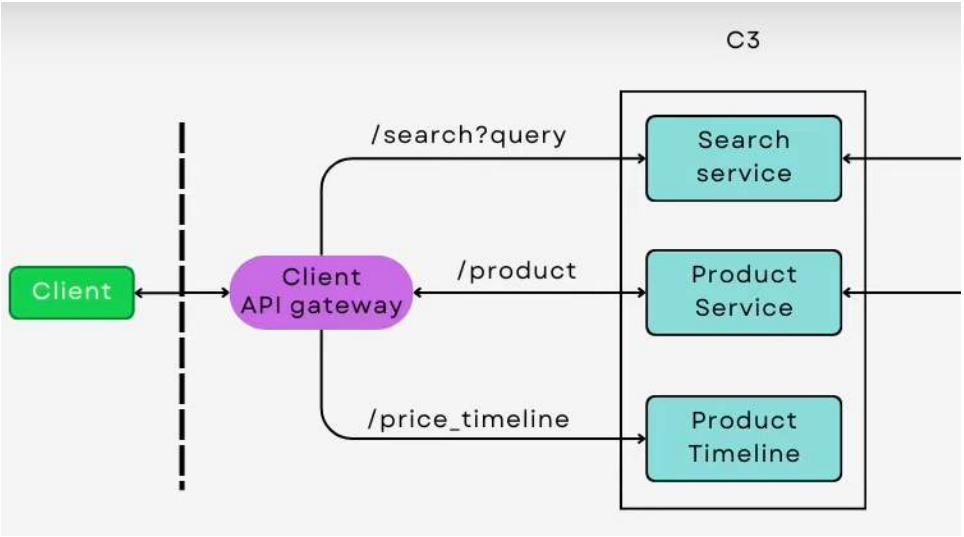

Amazon
Stale Price Tracker
Kafka Queue
Search service
SQS service
Product advertising service
Historical Pricing DB (Cassandra)
Product DB (Postgres)
Data Scraper
Kafka Queue
Sub System 1
Sub System 2
Price Watch DB
Price Drop Service
Kafka Queue
Notification Service
Email Browser WS Other
Notification DB
Users Subscribed To Price Alerts
Price Watch Table
product_ID: UUID
price: long
user_id: UUID
timestamp: long
status: type
description: string
product_data: JSON



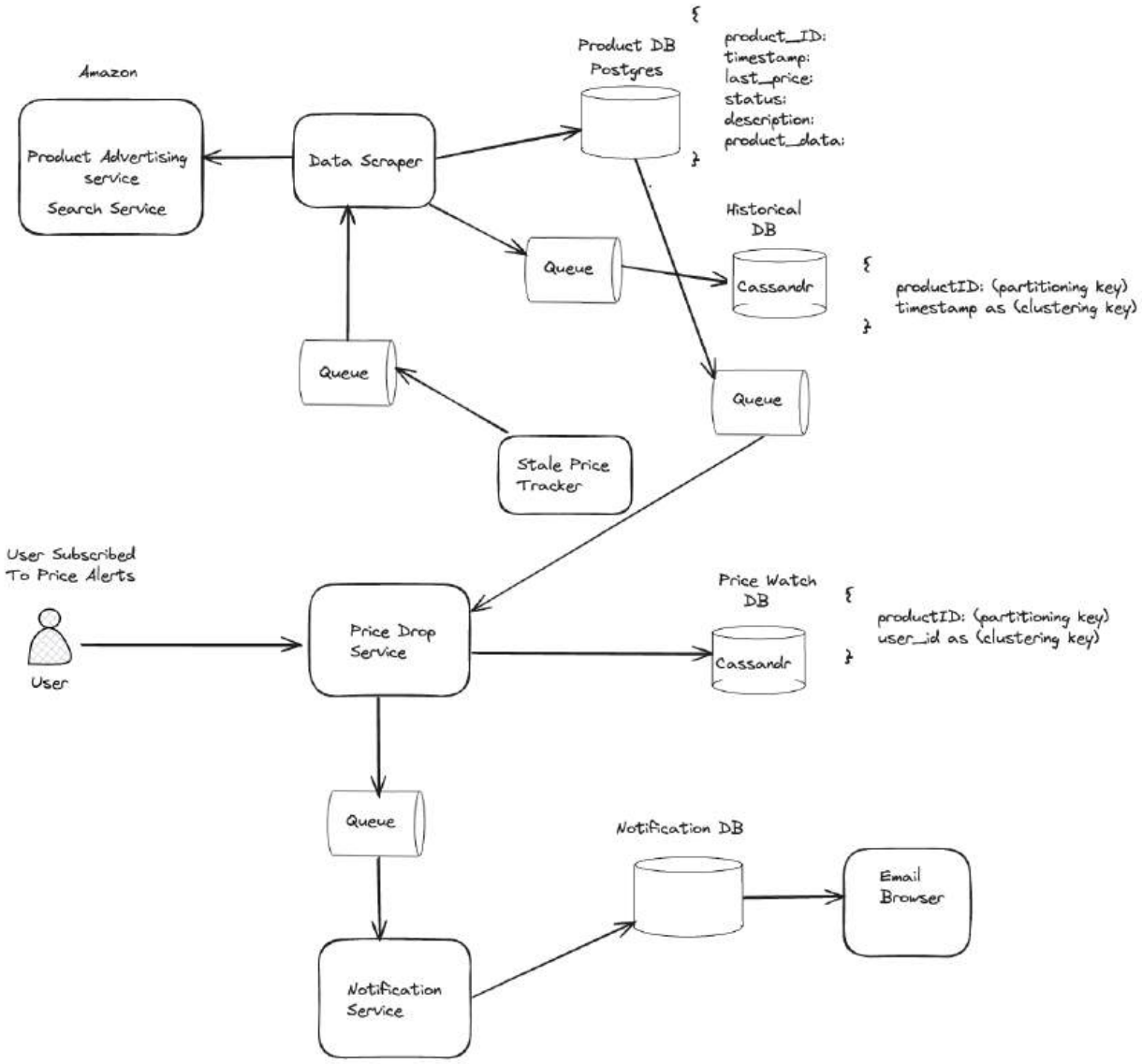

## STEP 1: UNDERSTANDING & QUESTIONS

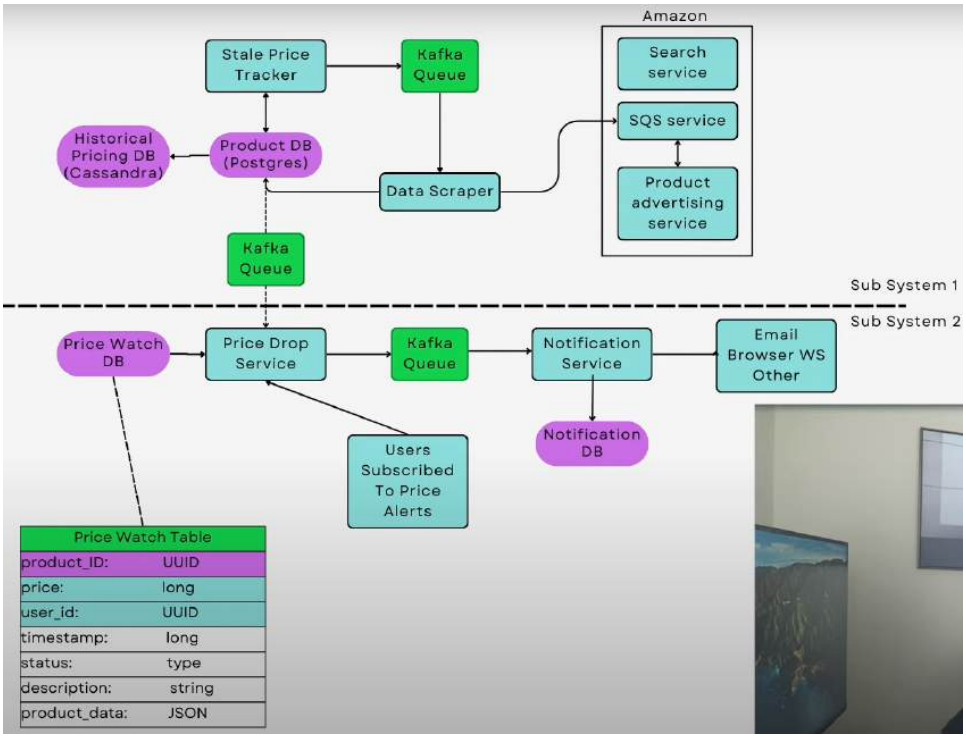
- What kind of website
- What kind of products?
- Estimated no.of users
- Should the users be notified about the drop

## STEP 2: API & DATA MODEL




## STEP 3: UNDERSTANDING & QUESTIONS


Amazon
Product Advertising service
Search Service
Data Scraper
Product DB
Postgres
{
product_ID:
timestamp:
last_price:
status:
description:
product_data:
}
Historical DB
Queue
Cassandr
{
productID: (partitioning key)
timestamp as (clustering key)
}
Queue
Queue
Stale Price Tracker
User Subscribed To Price Alerts
User
Price Drop Service
Price Watch DB
Cassandr
{
productID: (partitioning key)
user_id as (clustering key)
}
Queue
Notification DB
Email Browser
Notification Service




- Users are subscribed to few products at certain price
- In the backend, we have scrapers that scrape the website like amazon for the product and its price
- These requests are pushed to Kafka queue from where the scrapers process the data
- This data is stored in Product DB. SQL DB in this case
- Whenever there is an update in the timestamp of a particular product, we can have a trigger or place the event in the queue so that the same is updated in historical price DB
- This would be Cassandra DB with product id as partition key. Timestamp as clustering key
- At the same time, whenever there is a change from previous time, these events are sent to Kafka Queue
- We have Price Drop Service which polls the data and compares it with the Price Watch DB
- This would be Cassandra DB with product id as partition key. User_Id as clustering key
- If any user needs to be notified regarding this depending upon his initial price, then notification is sent using queue and notification service

# 21 - TOP K

## STEP 1: UNDERSTANDING & QUESTIONS

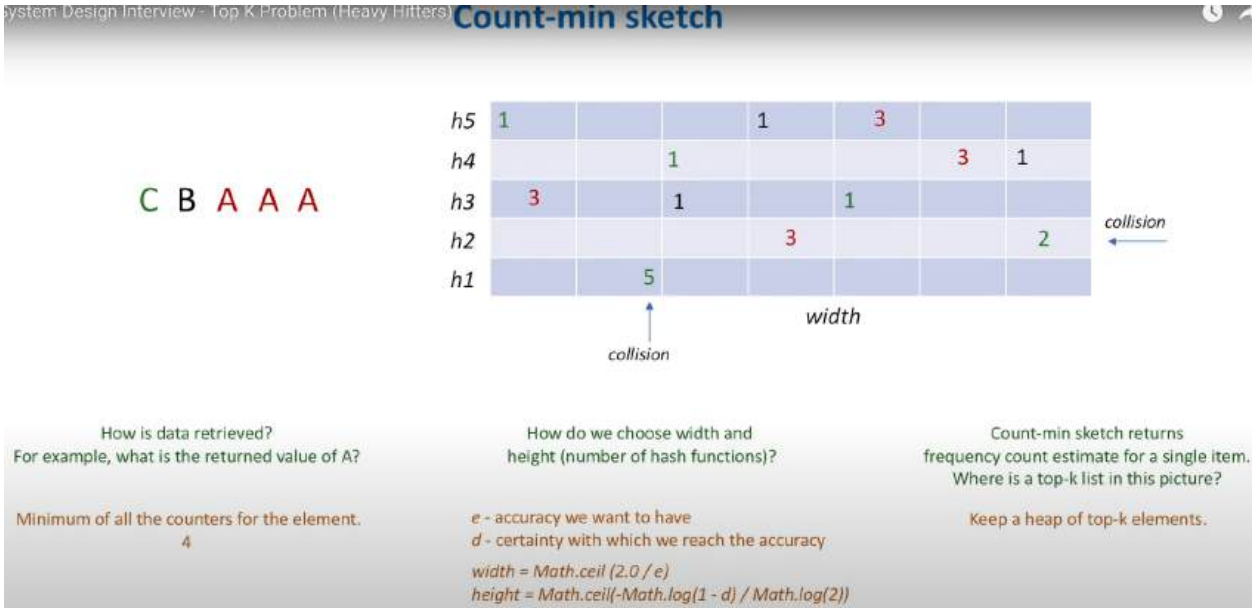
### Functional

- topK(k, startTime, endTime)

### Non-Functional

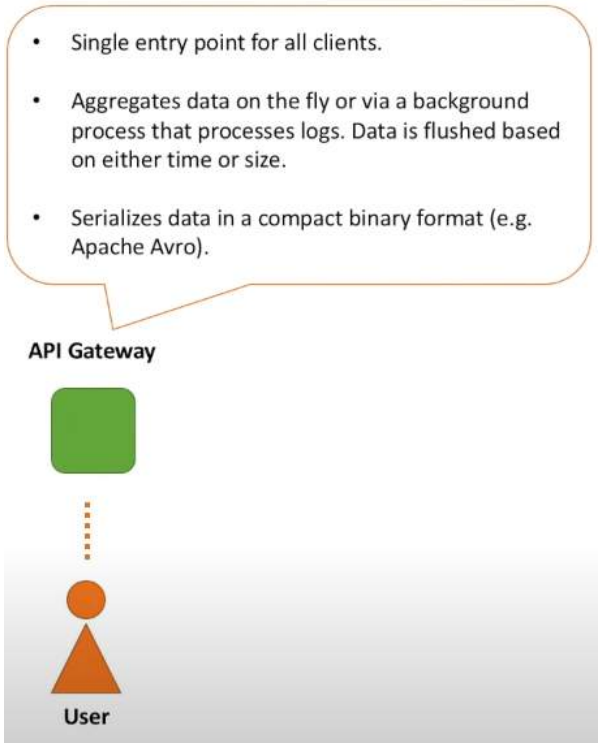
- Scalable (scales out together with increasing amount of data: videos, tweets, posts, etc.)
- Highly Available (survives hardware/network failures, no single point of failure)
- Highly Performant (few tens of milliseconds to return top 100 list)
- Accurate (as accurate as we can get)

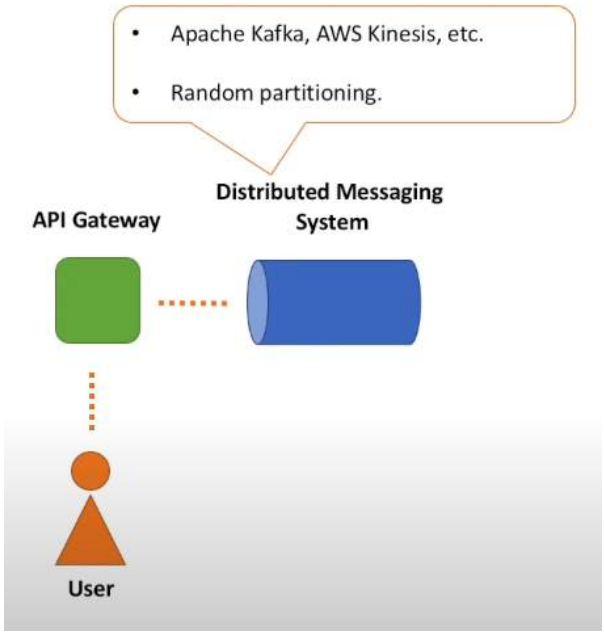
## USEFUL SOLUTION

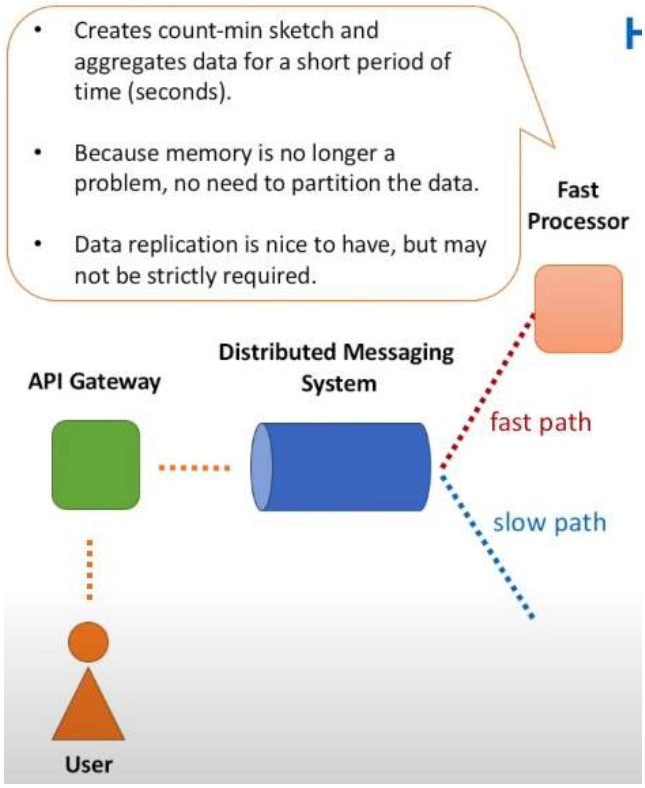


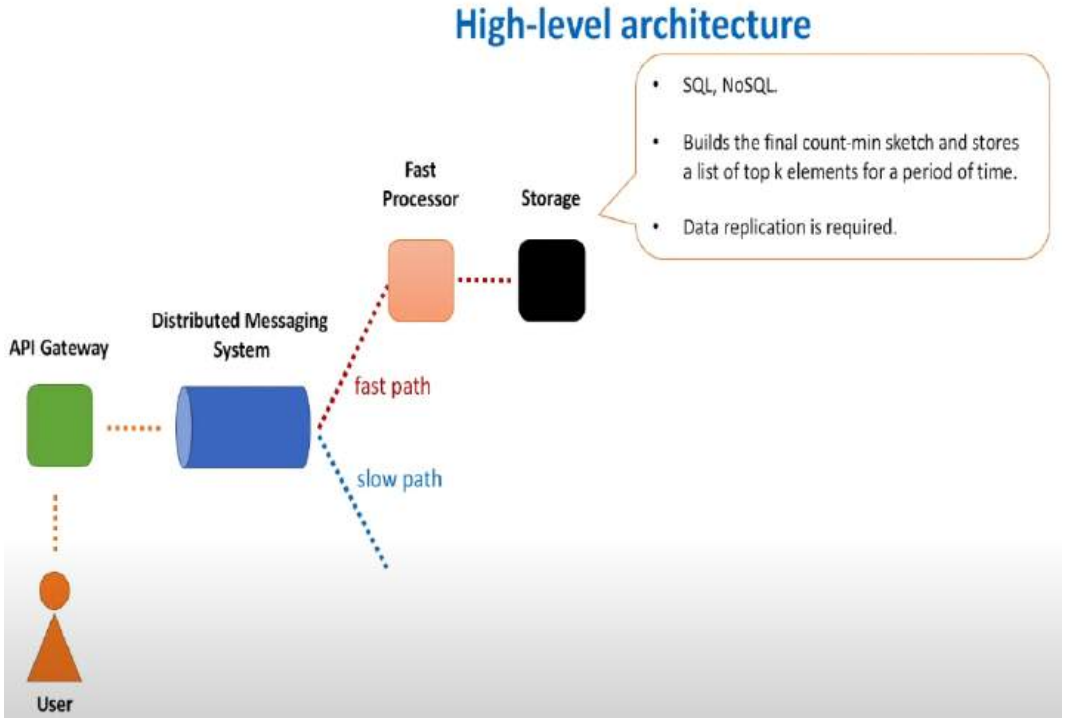

- We'll be using 2 paths. One is Fast Path using count min sketch and other is Slow path using Map Reduce jobs

## DESIGN

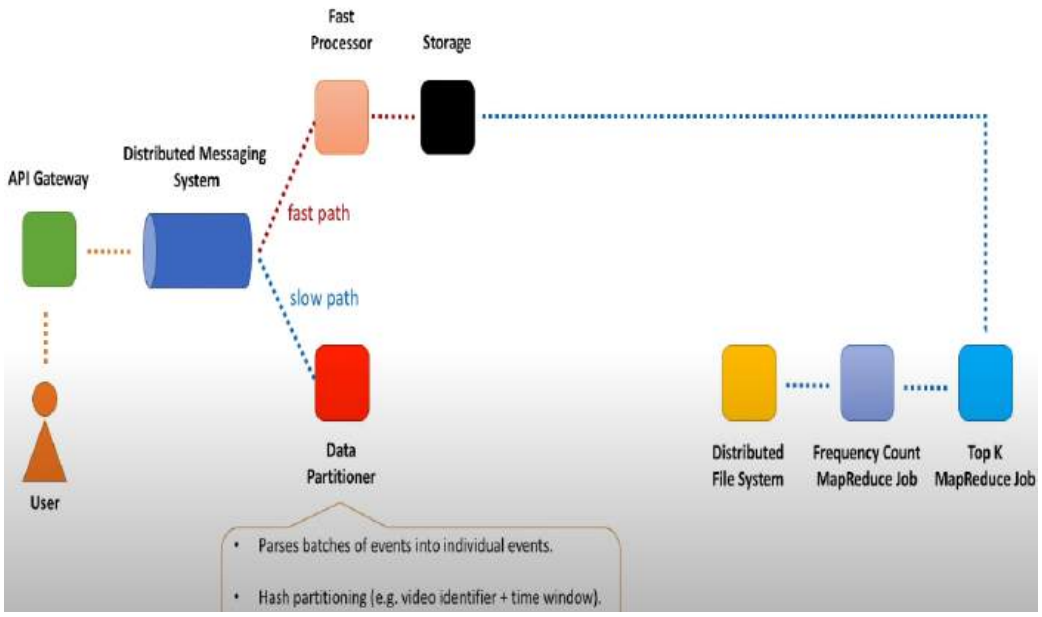

Single entry point for all clients.
Aggregates data on the fly or via a background process that processes logs. Data is flushed based on either time or size.
Serializes data in a compact binary format (e.g. Apache Avro).
API Gateway
User

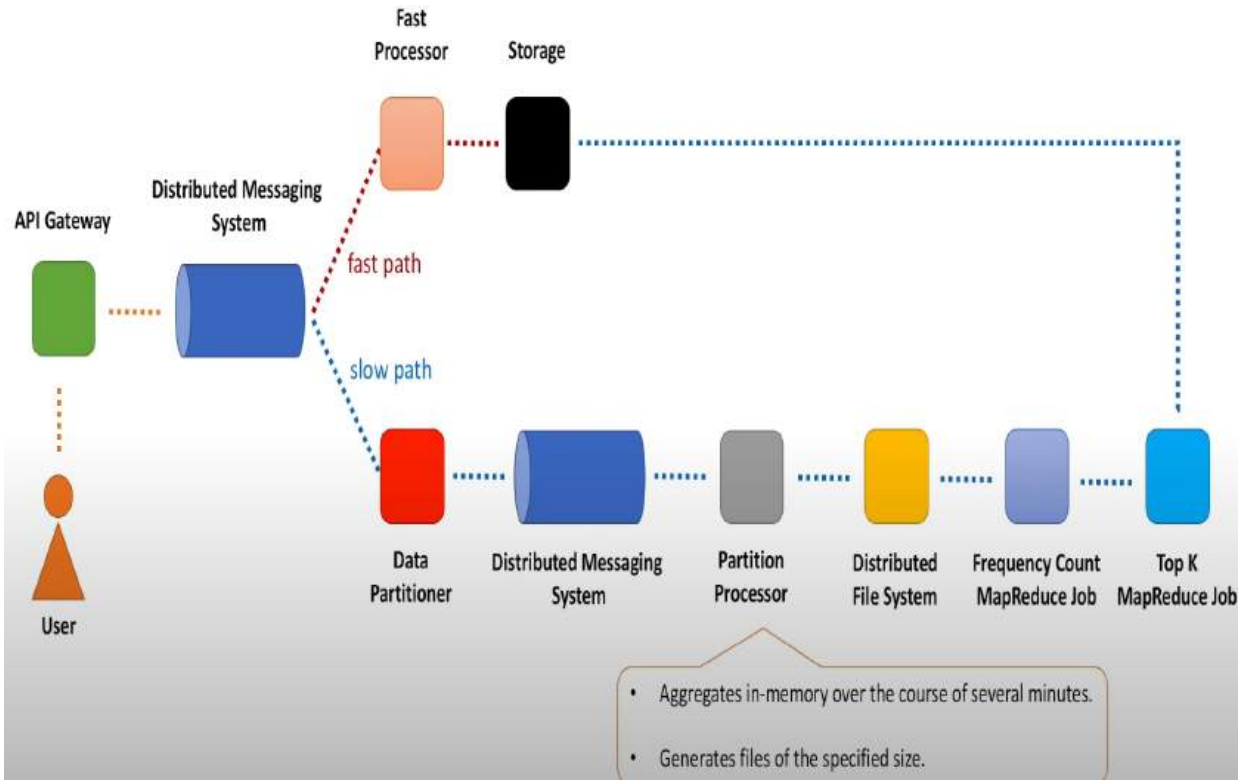

Apache Kafka, AWS Kinesis, etc.
Random partitioning.
Distributed Messaging System
API Gateway
User


Creates count-min sketch and aggregates data for a short period of time (seconds).
Because memory is no longer a problem, no need to partition the data.
Data replication is nice to have, but may not be strictly required.
Fast Processor
Distributed Messaging System
API Gateway
fast path
slow path
User


High-level architecture
SQL, NoSQL.
Builds the final count-min sketch and stores a list of top k elements for a period of time.
Data replication is required.
Fast Processor
Storage
Distributed Messaging System
API Gateway
fast path
slow path
User

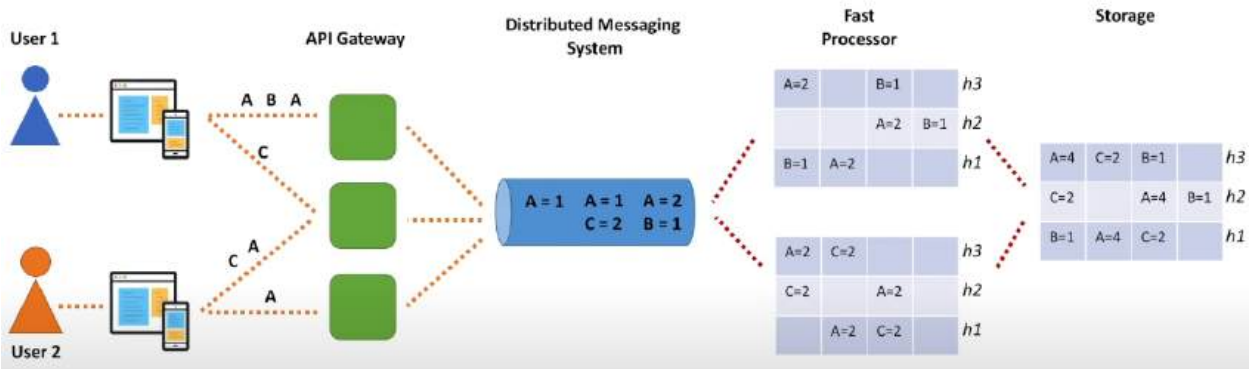
## High-level architecture



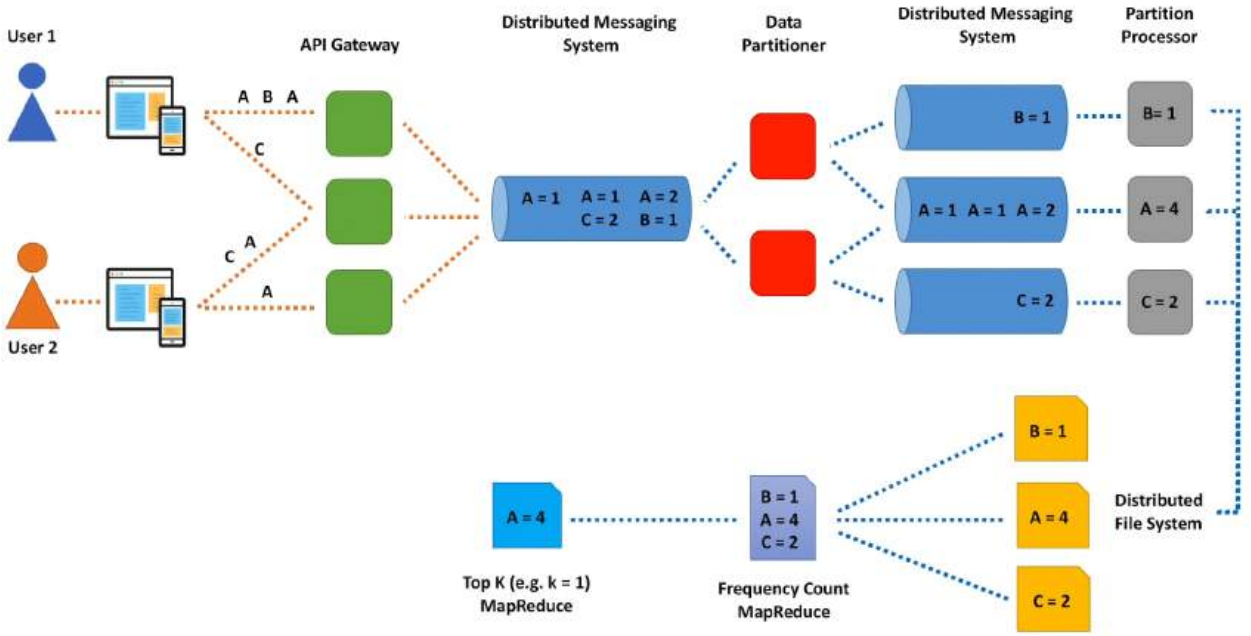




[Examples]

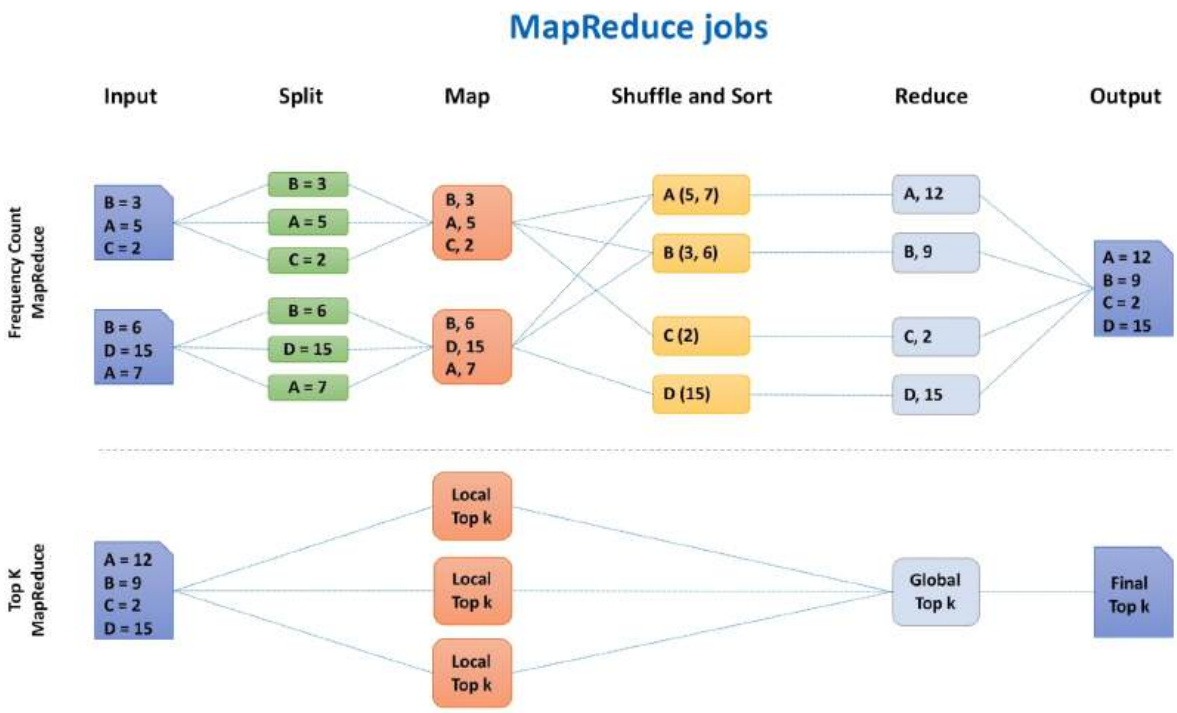
## Data flow, fast path




## Data flow, slow path




## EXPLANATION OF MAP REDUCE JOBS



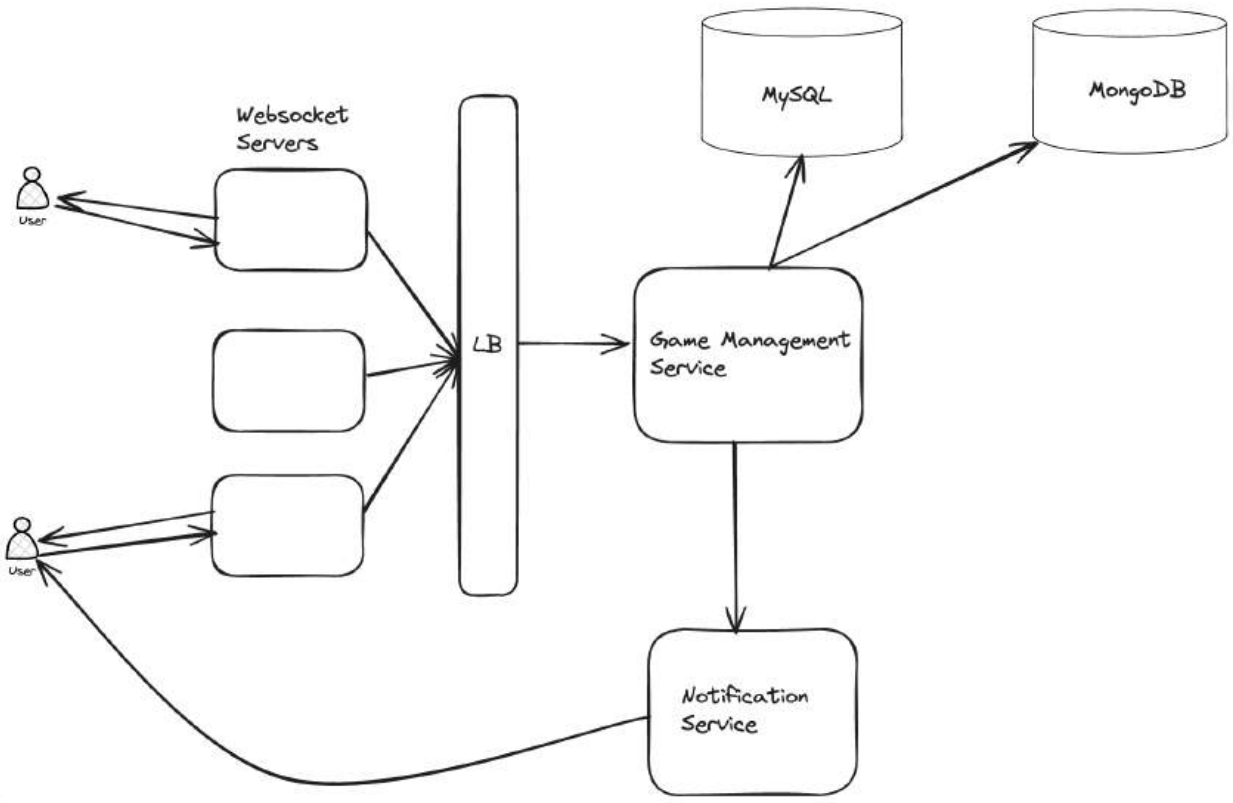

[SOURCE]

[System Design Interview - Top K Problem (Heavy Hitters)]


# 22 - CHESS GAME SERVICE

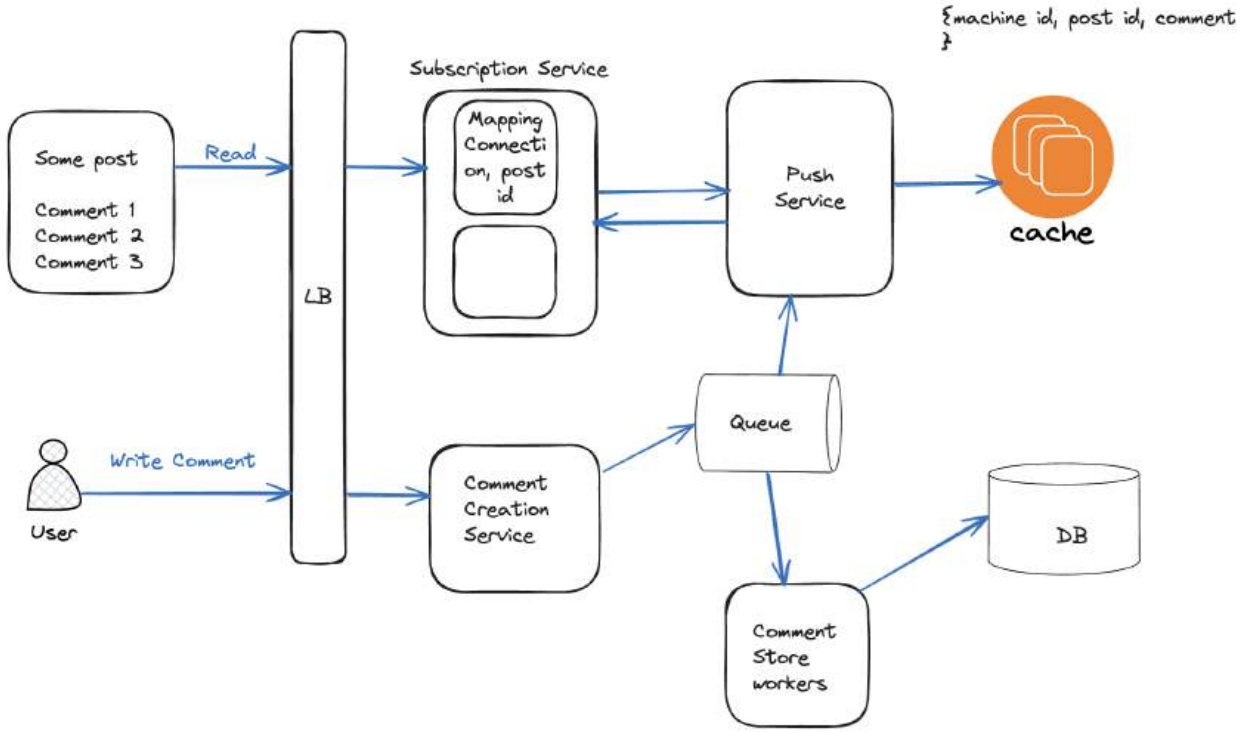
- Web socket servers for communication and chat
- RDBMS for user details

- **NoSQL for game details and the moves**




# 23 - BUILD LIVE COMMENTS

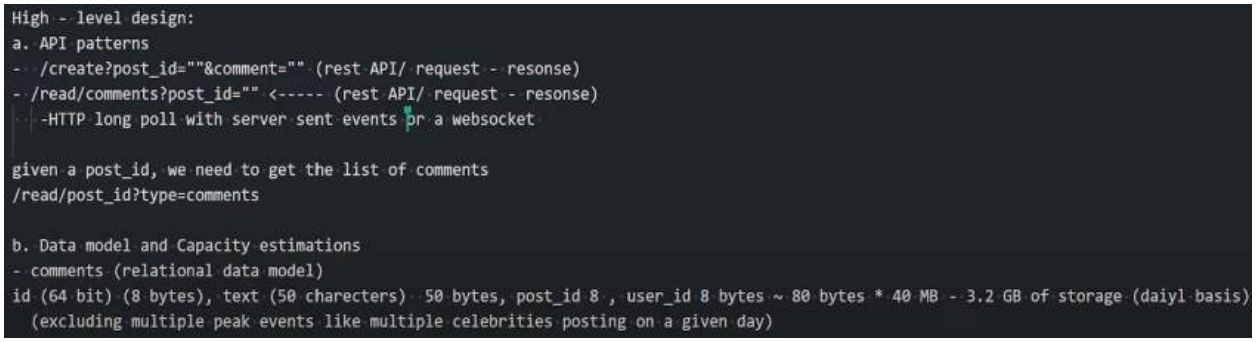
## FINAL DESIGN




## Portal o

## STEP 1: UNDERSTANDING & QUESTIONS

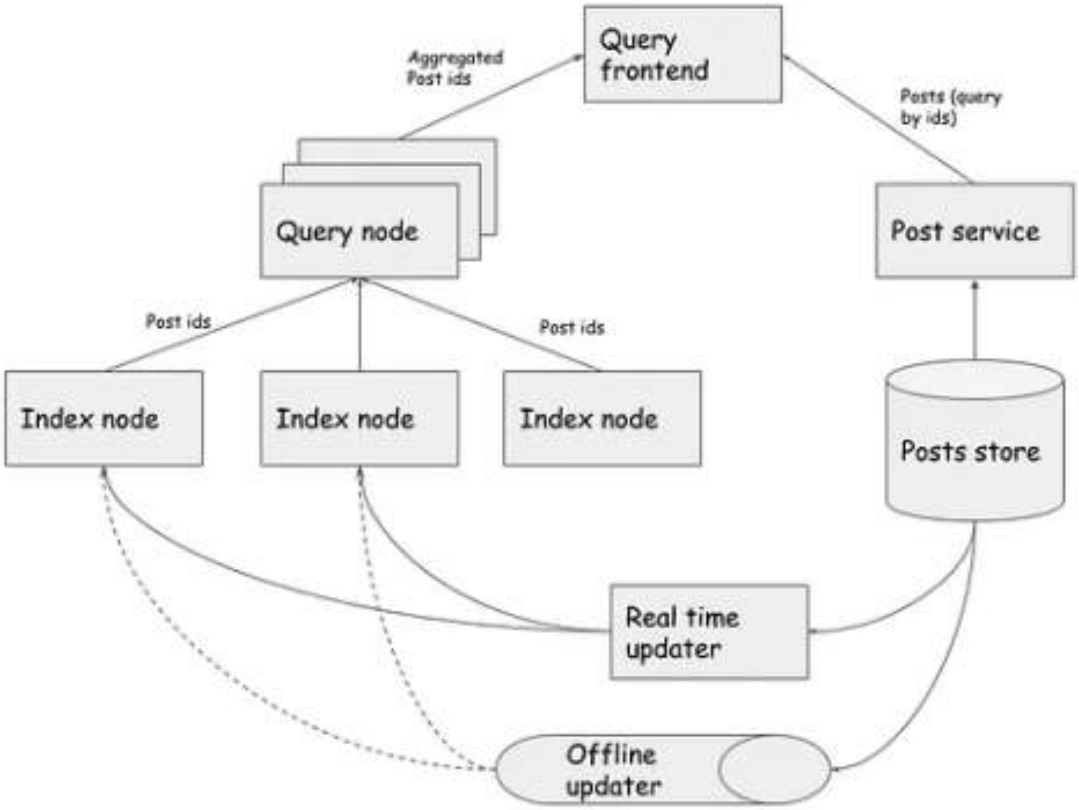
```
Functional Requirements:
1. When the comments show, should we do any sort of filtering? Filter for bad words, etc. Short answer is no
2. CRUD API: Update the comment, delete the live comment? Create and read
3. Is the comments just text based or do we support media (photos, video)? just text based for now
4. Is reactions to the comment in scope? Not in scope

What is in scope:
1. Creating and reading comments
2. Support for text based comments
3. Ordering comments by time
4. "Someone is typing"

Non Functional requirements:
1. Number of people who comments on a video/photo post -> (exlcuding celebrities) 25 -30 live comments
2. 1 billion users -> 1% of these users comments on a post -> 10,000,000 * 4 comments -> 40,000,000 comments per day (globally) and exlucding
   spikes due to celebrities
3. High consistency or High availablity (trade off) -> High availability with eventual consistency -> PErson A reading comments (10) while
   PErson B would see (15) and evenetual the information becomes consistent (CAP)
4. Read quorum would be lower
latency requirement - low latency
is the comment going to be persistent - yes (replay feature)
```




- Above, Websockets are preferred to Long polling because
  - Websockets offer bi-directional communication
  - If we go with long polling, Server can send events but again for the client to communicate to server, we will have to depend on another call

- User writes a comment. It gets to the LB and then gets forwarded to comment creation service
- All the comments are placed in a Kafka Queue
- From there, workers poll the events and place the comments in DB
- For the read, a user has already established a handshake. Using that itself, it waits for response
- For each user, websocket connection is established. And in the subscription service, information about which connection is used for which post id is stored
- Now, there is a push/pull service which has 2 jobs
  - When a user comments, from that Queue this service also polls and checks for each post all the connections that are subscribed to. Populates their pipeline accordingly in a Cache
  - At the same time, When the subscription service requests for information, it fetches the data from the cache and sends it back.

# 24 - STATUS SEARCH




an architecture for a search backend

***POST***

- Whenever a post is made, it is sent to the post service --> post DB etc
- At this time, the data is also pushed to a Kafka Queue
- We can have Index layer or some workers polling for updates and then feeding this data to the index layer.
- This is where the indexing can be done
    - Remove unnecessary words ex: of, the, and
    - Tokenize the words
    - Indexing the words along with a list of post ids
    - These need to be sharded in case of distributed systems
    - Index nodes store the indexed data locally

***SEARCH***

- Whenever the user searches for a string, the query is sent to query layer

- The raw keyword typed by user might be processed through a bunch of steps, like text normalization, query expansion, etc.
- Query node layer may also contact other services, like the service that provides user preferences, geo-location, or search history, which are all signals that help the translation.
- Response from index layer contains a list of ranked post ids. A query node is also responsible for aggregating the partial results to present a final result to the users.

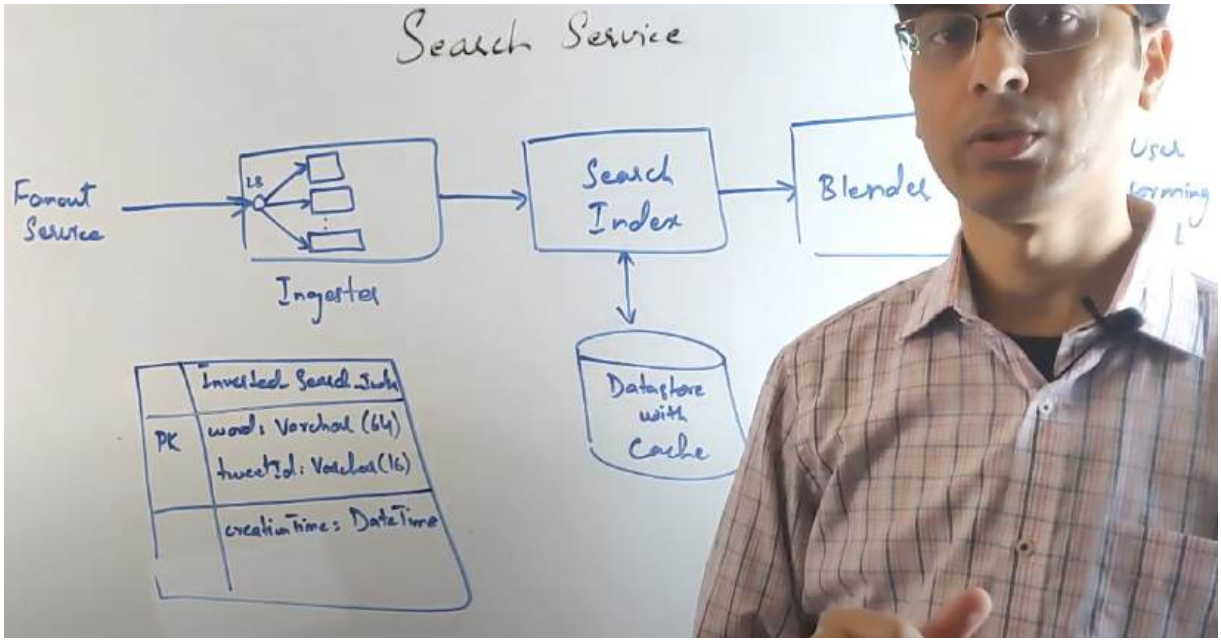
Along with real time updation of the index store, we might also need to batch updates so that all the posts are fed to the index layer properly

**Part 1: Posting Statuses and Indexing**

1. **Posting Statuses:**
   - When a user or a page posts a status on their timeline, the content of the status is captured along with metadata such as user ID (or page ID), timestamp, and any relevant tags or hashtags.
   - This status data is then passed to the backend for processing.
2. **Indexing Process:**
   - The backend system responsible for indexing statuses receives the posted status data.
   - The content of the status is parsed and analyzed to extract keywords, phrases, and any relevant metadata.
   - These parsed elements are then indexed using an indexing store (such as Apache Solr or Elasticsearch) for efficient retrieval during searches.
   - Additionally, the status data is stored in a database (such as Apache Cassandra or DynamoDB) for persistence.
3. **Storage:**
   - In the database, the status is stored along with its associated metadata (user ID, timestamp, etc.).
   - The indexing store contains documents representing each status, with fields for content, user ID, timestamp, and any other relevant metadata.
   - These documents are indexed using inverted indexes, where each indexed term points to the documents containing that term.
   - The indexing store is continuously updated with new statuses to ensure real-time search functionality.

**Part 2: Searching for Statuses**

1. **User Search Request:**
   - When a user initiates a search request, the search service receives the search query along with the user's ID.
2. **Query Processing:**
   - The search service parses the search query and identifies the user's friends and the pages they follow.
   - It constructs a search query that includes filters based on the user's relationships and subscriptions.
   - These filters limit the search scope to statuses authored by the user's friends and the pages they follow.
3. **Data Retrieval:**
   - The search service queries the indexing store with the constructed search query, applying filters based on the user's relationships and subscriptions.
   - Only statuses that meet the filtering criteria (authored by friends or from followed pages) are retrieved from the indexing store.
   - The retrieved statuses are aggregated and ranked based on relevance and other factors, such as recency or engagement.
4. **Presentation to User:**
   - The search results, consisting of relevant statuses authored by the user's friends and followed pages, are formatted and presented to the user interface for display.
   - Pagination and infinite scrolling techniques may be applied to efficiently handle large result sets.


Search Service
Fanout Service
LB
Ingester
Search Index
Blender
Datastore with Cache
Inverted Search Index
PK
word: Varchar (64)
tweetId: Varchar(16)
creationTime: DateTime