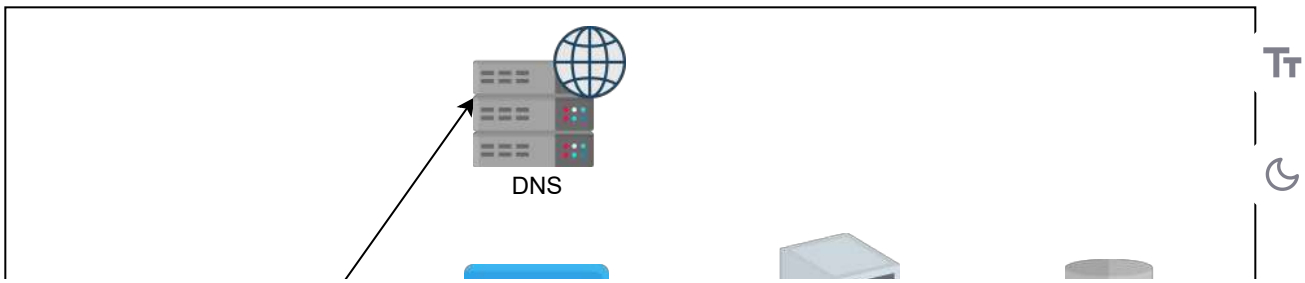
High-level Design of Twitter

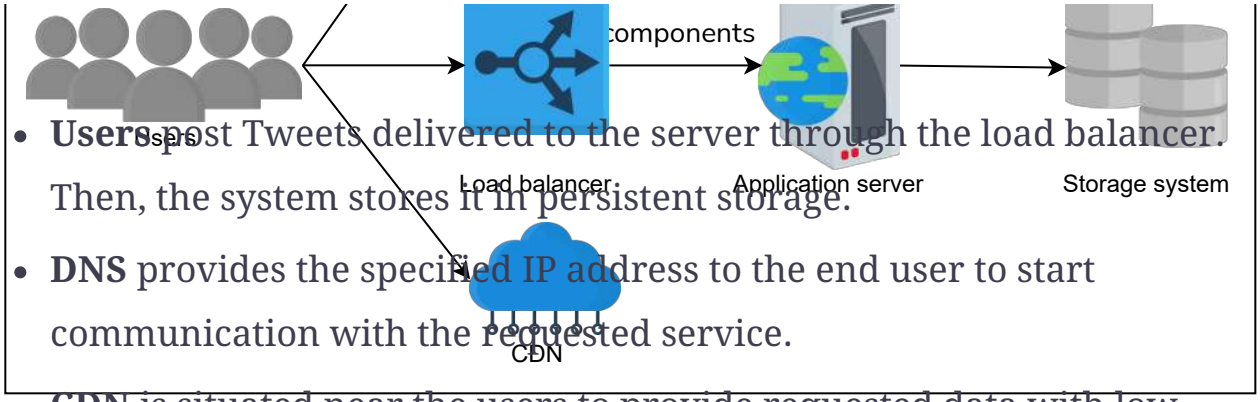
Understand the high-level design of the Twitter service.

We'll cover the following

User-system interaction

Let’s begin with the high-level design of our Twitter system. We’ll initially highlight and discuss the building blocks, as well as other components, in the context of the Twitter problem briefly. Later on, we’ll dive deep into a few components in this chapter.





- **Users** post Tweets delivered to the server through the load balancer. Then, the system stores it in persistent storage.
- **DNS** provides the specified IP address to the end user to start communication with the requested service.
- **CDN** is situated near the users to provide requested data with low latency. When users search for a specified term or tag, the system first searches in the CDN proxy servers containing the most frequently requested content.
- **Load balancer** chooses the operational application server based on traffic load on the available servers and the user requests.
- **Storage system** represents the various types of storage (SQL-based and NoSQL-based) in the above illustration. We'll discuss significant storage systems later in this chapter.
- **Application servers** provide various services and have business logic to orchestrate between different components to meet our functional requirements.

We have detailed chapters on DNS, CDN, specified storage systems (Databases, Key-value store, Blob store), and Load balancers in our building blocks section. We'll focus on further details specific to the Twitter service in the coming lessons. Let's first understand the service API.

API design

This section will focus on designing various APIs regarding the functionalities we are providing. We learn how users request various services through APIs. We'll only concentrate on the significant parameter of the APIs that are relevant to our design. Although the front-end server can call another API or add more parameters in the API received from the end

users to fulfill the given request, we consider all relevant arguments specified for the particular request in a single API. Let's develop APIs for each of the following features:

- Post Tweet
- Like or dislike Tweet
- Reply to Tweet
- Search Tweet
- View user or home timeline
- Follow or unfollow the account
- Retweet a Tweet

Post Tweet

The POST method is used to send the Tweet to the server from the user through the `/postTweet` API.

```
postTweet(user_id, access_type, tweet_type, content, tweet_length, media_field, po
st_time, tweet_location, list_of_used_hashtags, list_of_tagged_people)
```

Let's discuss a few of the parameters:

Parameter	Description
`user_id`	It indicates the unique ID of the user who posted the Tweet.
`access_type`	It tells us whether the Tweet is protected (that is, only visible to follc public.
`tweet_type`	It indicates whether the Tweet is text-based, video-clip based, im consisting of different types.
`content`	It specifies the Tweet's actual content (text).
`tweet_length`	It represents the text length in the Tweet. In the case of video, it duration and size of a video.

`media_field`	It specifies the type of media (image, video, GIF, and so on) delivered Tweet.

The rest of the parameters are self-explanatory.

> **Note:** Twitter uses the **Snowflake** service to generate unique IDs for Tweets. We have a detailed chapter (Sequencer) that explains this service.

Points to Ponder

Question 1

At most, how many hashtags can a Tweet have?

Show Answer

1 of 2

Like or dislike Tweet

The `/likeTweet` API is used when users like public Tweets.

```
likeTweet(user_id, tweet_id, tweeted_user_id, user_location)
```

Parameter	Description
`user_id`	It indicates the unique ID of the user who liked the Tweet.
`tweet_id`	It indicates the Tweet's unique ID.
`tweeted_user_id`	This is the unique ID of the user who posted the Tweet.
`user_location`	It denotes the location of the user who liked the Tweet.

The parameters above are also used in the `/dislikeTweet` API when users dislike others' Tweets.

Reply to Tweet

The `/replyTweet` API is used when users reply to public Tweets.

```
replyTweet(user_id, tweet_id, tweeted_user_id, reply_type, reply_length)
```

The `reply_type` and `reply_length` parameters are the same as `tweet_type` and `tweet_length` respectively.

Search Tweet

When the user searches any keyword in the home timeline, the GET method is used. The following is the `/searchTweet` API:

```
searchTweet(user_id, search_term, max_result, exclude, media_field, expansions, sort_order, next_token, user_location)
```

Some new parameters introduced in this case are:

Parameter	Description

`search_term`	It is a string containing the search keyword or phrase.
`max_result`	It is the number of Tweets returned per response page. By default, t response is 10.
`exclude`	It specifies what to exclude from the returned Tweets, that is, replie The maximum limit on returned Tweets is 3200, but when we exclu maximum limit is reduced to 800 Tweets.
`media_field`	It specifies the media (image, video, GIF) delivered in each returned
`expansions`	It enables us to request additional data objects in the returned Twee mentioned user, referenced Tweet, attached media, attached places on.
`sort_order`	It specifies the order in which Tweets are returned. By default, it wi recent Tweets first.
`next_token`	It is used to get the next page of results. For instance, if `max_result` i Tweets, and the result set contains 200 Tweets, then the value of `nex` pulled from the response to request the next page containing the fol Tweets. The last result (page) will not have a `next_token`.

Response

Let's look at a sample response in JSON format. The **id** is the user's unique ID who posted the Tweet and the **text** is the Tweet's content. The **result_count** is the count of the returned Tweet, which we set in the `max_result` in the request. Here, we're displaying the default fields only.

Click to see response in JSON

> **Note:** Twitter performs various types of searches. The following are two of them:
>
> - One search type returns the result of the last seven days, which all registered users usually use.
> - The other type returns all matching results on all Tweets ever posted (remind that service does not delete a posted Tweet). Indeed, matches can contain the first Tweet on Twitter. This search is usually used for academic research.

View home_timeline

The GET method is suitable when users view their home timelines through the `/viewHome_timeline` API.

```
viewHome_timeline(user_id, tweets_count, max_result, exclude, next_token, user_loc
ation)
```

In the `/viewUser_timeline` API, we'll exclude the `user_location` to get the user timeline.

The `max_rsult` parameter determines the number of tweets a client application can show the user. The server sends the `max_result` number of tweets in each response. Further, the server will also send a paginated `list_of_followers` to reduce the client latency.

Point to Ponder

Question

Which parameter in the `viewHome_timeline` method is the most relevant when deciding which promoted ads (Tweets) to be returned in response?

Show Answer

Follow the account

The `/followAccount` API is used when users follow someone's account on Twitter.

```
followAccount(account_id, followed_account_id)
```

Parameter	Description
`account_id`	It specifies the unique ID of a user who follows that account on Twit
`followed_account_id`	It indicates the unique ID of the account that the user follows.

The `/unfollowAccount` API will use the same parameters when a user unfollows someone's account on Twitter.

Retweet a Tweet

When a registered user Retweets (re-posts) someone's Tweet on Twitter, th following `/retweet` API is called:

```
retweet(user_id, tweet_id, retweet_user_id)
```

The same parameters will be required in the `/undoRetweet` API when users undo a Retweet of someone's Tweet.

Back

Requirements of Twitter's Design

Next

Detailed Design of Twitter

Completed

Detailed Design of Twitter

Take a deep dive into the detailed design of Twitter.

We'll cover the following

- Storage system
- Cache
- Observability
- Real-world complex problems
- The complete design overview

Storage system

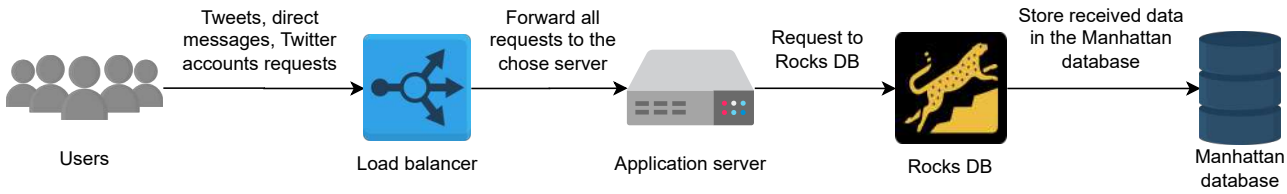
Storage is one of the core components in every real-time system. Although we have a detailed chapter on storage systems, here, we'll focus on the storage system used by Twitter specifically. Twitter uses various storage models for different services to take full advantage of each model. We'll discuss each storage model and see how Twitter shifted from various databases, platforms, and tools to other ones and how Twitter benefits from all of these.

The content in this lesson is primarily influenced by Twitter's technical blogs, though the analysis is ours.

- **Google Cloud**: In Twitter, HDFS (Hadoop Distributed File System) consists of tens of thousands of servers to host over 300PB data. The data stores in HDFS are mostly compressed by the LZO (data

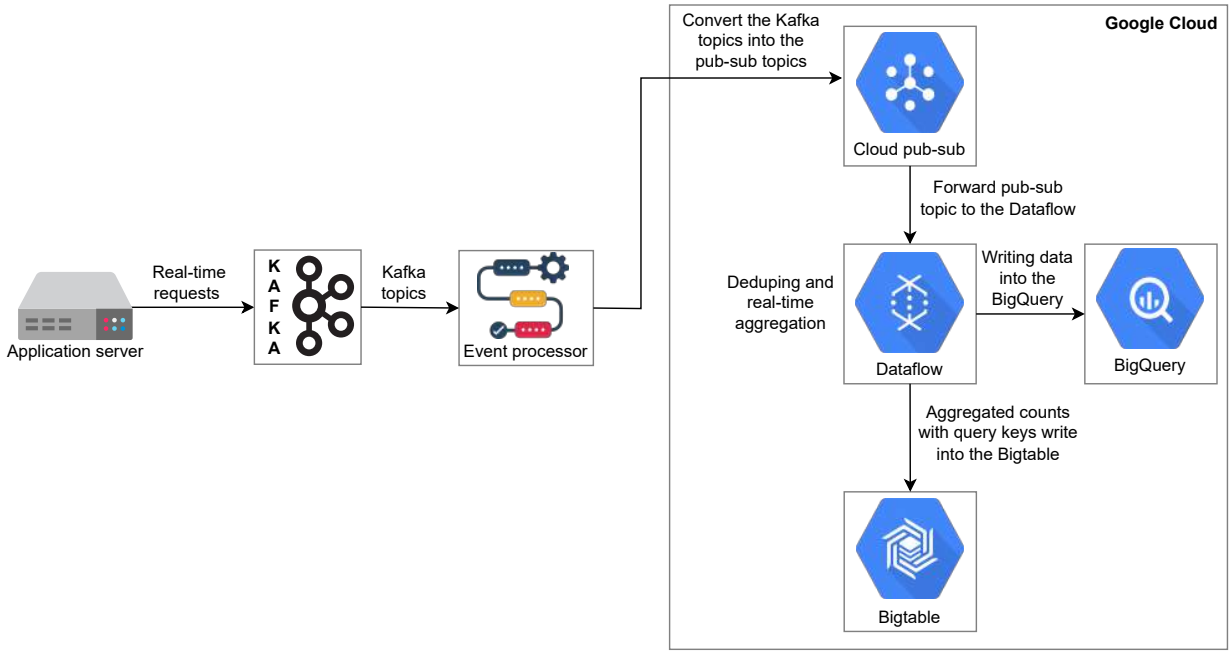
compression algorithm) because LZO works efficiently in Hadoop. This data includes logs (client events, Tweet events, and timeline events), MySQL and Manhattan (discussed later) backups, ad targeting and analytics, user engagement predictions, social graph analysis, and so on. In 2018, Twitter decided to shift data from Hadoop clusters to the Google Cloud to better analyze and manage the data. This shift is named a **partly cloudy** strategy. Initially, they migrated Ad-hoc clusters (occasional analysis) and cold storage clusters (less accessed and less frequently used data), while the real-time and production Hadoop clusters remained. The big data is stored in the BigQuery (Google cloud service), a fully managed and highly scalable serverless data warehouse. Twitter uses the Presto (distributed SQL query engine) to access data from Google Cloud (BigQuery, Ad-hoc clusters, Google cloud storage, and so on).

- **Manhattan**: On Twitter, users were growing rapidly, and it needed a scalable solution to increase the throughput. Around 2010, Twitter used **Cassandra** (a distributed wide-column store) to replace MySQL but could not fully replace it due to some shortcomings in the Cassandra store. In April 2014, Twitter launched its own general-purpose real-time distributed key-value store, called Manhattan, and deprecated Cassandra. Manhattan stores the backend for Tweets, Twitter accounts, direct messages, and so on. Twitter runs several clusters depending on the use cases, such as smaller clusters for non-common or read-only and bigger for heavy read/write traffic (millions of QPS). Initially, Manhattan had also provided the time-series (view, like, and so on.) counters service that the MetricsDB now provides. Manhattan uses RocksDB as a storage engine responsible for storing and retrieving data within a particular node.



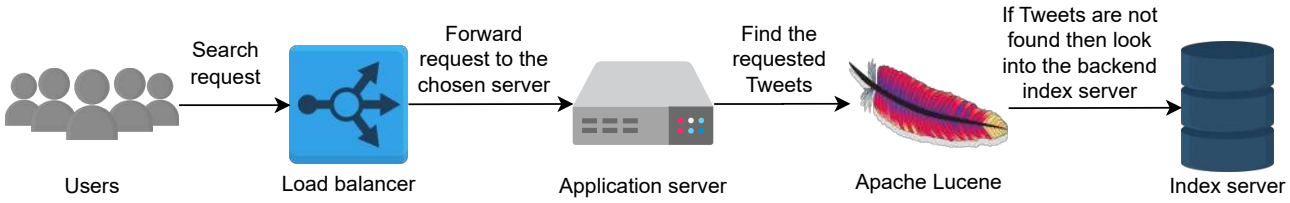
Twitter's data move to the Manhattan database

- **Blobstore**: Around 2012, Twitter built the Blobstore storage system to store photos attached to Tweets. Now, it also stores videos, binary files, and other objects. After a specified period, the server checkpoints the in-memory data to the Blobstore as durable storage. We have a detailed chapter on the Blob Store, which can help you understand what it is and how it works.
- **SQL-based databases**: Twitter uses MySQL and PostgreSQL, where it needs strong consistency, ads exchange, and managing ads campaigns. Twitter also uses Vertica to query commonly aggregated datasets and Tableau dashboards. Around 2012, Twitter also built the Gizzard framework on top of MySQL for sharding, which is done by partitioning and replication. We have a detailed discussion on relational stores in our Databases chapter.
- **Kafka and Cloud dataflow**: Twitter evaluates around 400 billion real-time events and generates petabytes of data every day. For this, it processes events using Kafka on-premise and uses Google Dataflow jobs to handle deduping and real-time aggregation on Google Cloud. After aggregation, the results are stored for ad-hoc analysis to BigQuery (data warehouse) and the serving system to the Bigtable (NoSQL database). Twitter converts Kafka topics into Cloud Pub-sub topics using an event processor, which helps avoid data loss and provides more scalability.



Real-time processing of billions of requests

- **FlockDB**: A relationship refers to a user's followers, who the user follows, whose notifications the user has to receive, and so on. Twitter stores this relationship in the form of a graph. Twitter used FlockDB, a graph database tuned for huge adjacency lists, rapid reads and writes, and so on, along with graph-traversal operations. We have a chapter on Databases and Newsfeed that discuss graph storage in detail.
- **Apache Lucene**: Twitter constructed a search service that indexes about a trillion records and responds to requests within 100 milliseconds. Around 2019, Twitter's search engine had an indexing latency (time to index the new tweets) of roughly 15 seconds. Twitter uses Apache Lucene for real-time search, which uses an inverted index. Twitter stores a real-time index (recent Tweets during the past week) i RAM for low latency and quick updates. The full index is a hundred times larger than the real-time index. However, Twitter performs batc processing for the full indexes. See the Distributed Search chapter to deep dive into how indexing works.

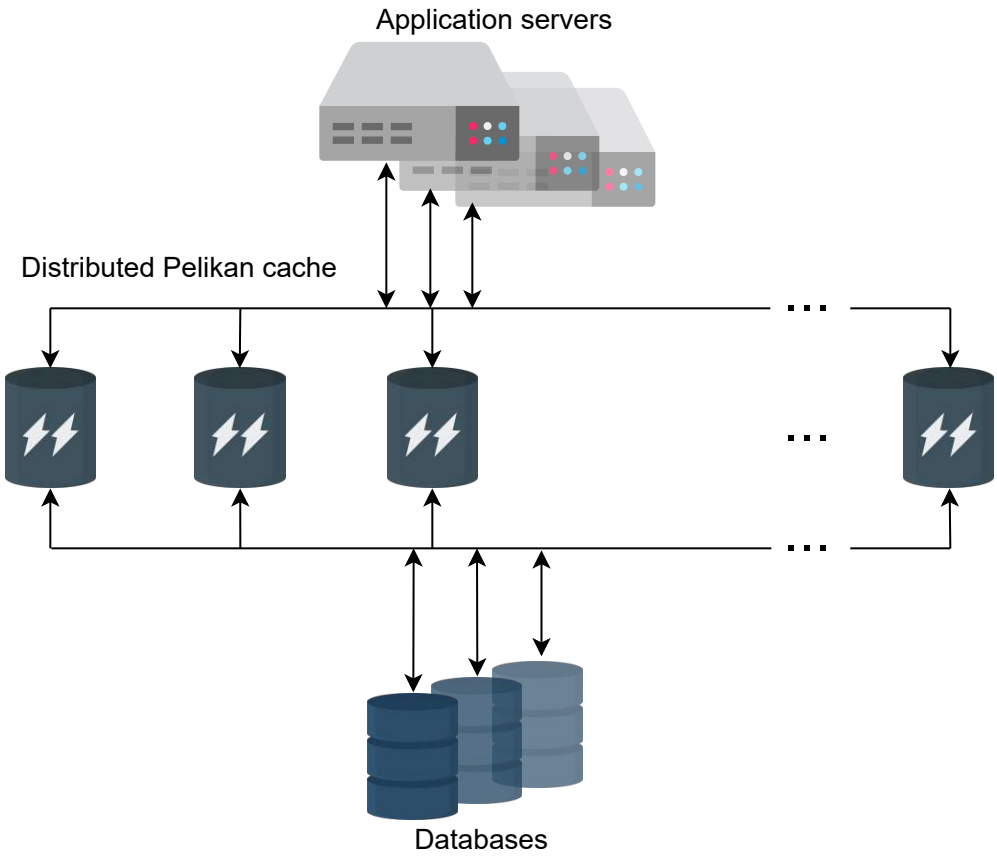


Processing the search request

The solution based on the “one size fits all” approach is infrequently effective. Real-time applications always focus on providing the right tool for the job, which needs to understand all possible use cases. Lastly, everything has upsides and downsides and should be applied with a sense of reality.

Cache

As we know, caches help to reduce the latency and increase the throughput. Caching is mainly utilized for storage (heavy read traffic), computation (real-time stream processing and machine learning), and transient data (rate limiters). Twitter has been used as multi-tenant (multiple instances of an application have the shared environment) Twitter Memcached (Twemcache) and Redis (Nighthawk) clusters for caching. Due to some issues such as unexpected performance, debugging difficulties, and other operational hassles in the existing cache system (Twemcache and Nighthawk), Twitter has started to use the **Pelikan** cache. This cache gives high-throughput and low latency. Pelikan uses many types of back-end servers such as the `peliken_twemcache` replacement of Twitter’s Twemcache server, the `peliken_slimcache` replacement of Twitter’s Memcached/Redis server, and so on. To dive deep, we have a detailed chapter on an In-memory Cache. Let’s have a look at the below illustration representing the relationship of application servers with distributed Pelikan cache.



Distributed Pelikan cache

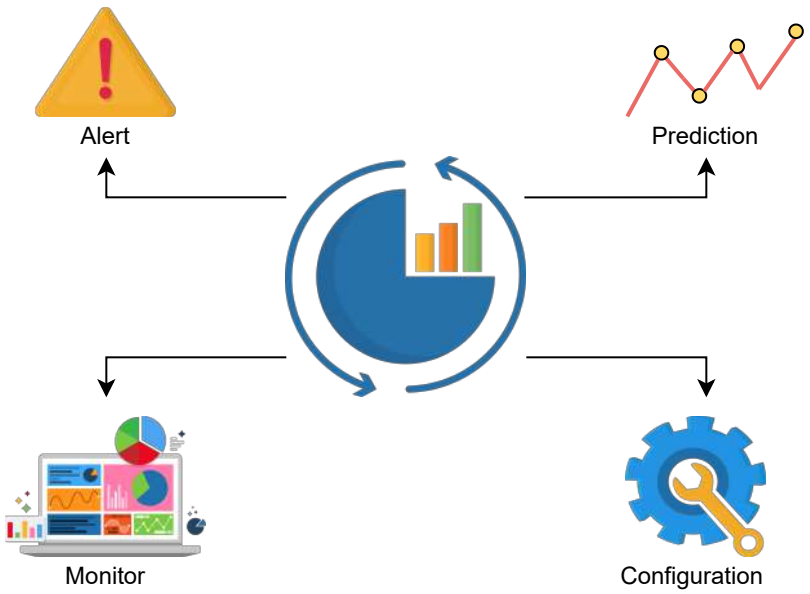
> **Note:** Pelikan also introduced another back-end server named **Segcache**, which is extremely scalable and memory-efficient for small objects. Typically, the median size of the small object is between 200 to 300 bytes in a large-scale application's cache. Most solutions (Memcache and Redis) have a high size (56 bytes) of metadata with each object. This signifies that the metadata takes up more than one-third of the memory. Pelikan reduced metadata size per object to 38 bytes. Segcache also received NSDI Community Award and is used as an experimental server as of 2021.

Observability

Real-time applications use thousands of servers that provide multiple services. Monitoring resources and their communication inside or outside

the system is complex. We can use various tools to monitor services' health, such as providing alerts and support for multiple issues. Our alert system notifies broken or degraded services triggered by the set metrics. We can also use the dynamic configuration library that deploys and updates the configuration for multiple services without restarting the service. This library leverages the ZooKeeper (discussed later) for the configuration as a source of truth. Twitter used the LonLens service that delivers visualization and analytics of service logs. Later, it was replaced by Splunk Enterprise, a central logging system.

Tracing billions of requests is challenging in large-scale real-time applications. Twitter uses Zipkin, a distributed tracing system, to trace each request (spent time and request count) for multiple services. Zipkin selects a portion of all the requests and attaches a lightweight trace identifier. This sampling also reduces the tracing overhead. Zipkin receives data through the **Scribe** (real-time log data aggregation) server and stores it in the key-value stores with few indexes.



Observability

Most real-time applications use ZooKeeper to store critical data. It can also provide multiple services such as distributed locking and leader election in the distribution system. Twitter uses ZooKeeper to store service registry, Manhattan clusters' topology information, metadata, and so on. Twitter also uses it for the leader election on the various systems.

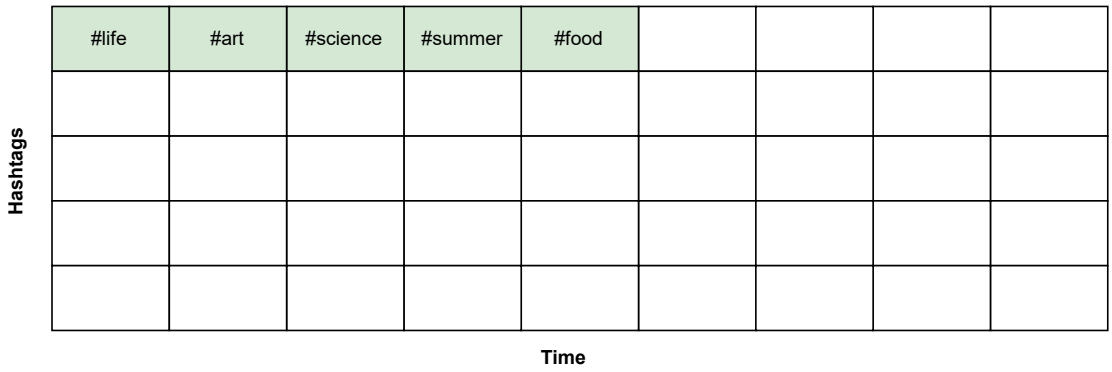
Real-world complex problems

Twitter has millions of accounts, and some accounts (public figures) have millions of followers. When these accounts post Tweets, millions of followers of the respective account engage with their Tweets in a short time. The problem becomes big when the system handles the billions of interactions (such as views and likes) on these Tweets. This problem is also known as the **heavy hitter** problem. For this, we need millions of counters to count various operations on Tweets.

Moreover, a single counter for each specific operation on the particular Tweet is not enough. It's challenging to handle millions of increments or decrements requests against a particular Tweet in a single counter. Therefore, we need multiple distributed counters to manage burst write requests (increments or decrements) against various interactions on celebrities' Tweets. Each counter has several shards working on different computational units. These distributed counters are known as **sharded counters**. These counters also help in another real-time problem named the **Top-k** problem. Let's discuss an example of Twitter's Top-k problems: trends and timeline.

Trends: Twitter shows Top-k trends (hashtags or keywords) locally and globally. Here, "locally" refers to when a topic or hashtag is used within the exact location where the requested user is active. Alternatively, "globally" refers to when the particular hashtag is used worldwide. There is a possibility that users from some regions are not using a specific hashtag in

their Tweets but get this hashtag in their trends timeline. Hashtags with the maximum frequency (counts) become trends both locally and globally. Furthermore, Twitter shows various promoted trends (known as “paid trends”) in specified regions under trends. The below slides represent hashtags in the sliding window selected as Top-k trends over time.



Top-k trends in a specified time frame

1 of 5

Timeline: Twitter shows two types of timelines: home and user timelines. Here, we’ll discuss the home timeline that displays a stream of Tweets posted by the followed accounts. The decision to show Top-k Tweets in the timeline includes followed accounts Tweets and Tweets that are liked or Retweeted by the followed accounts. There’s also another category of promoted Tweets displayed in the home timeline.

Sharded counters solve the discussed problems efficiently. We can also place shards of the specified counter near the user to reduce latency and increas overall performance like **CDN**. Another benefit we can get is a frequent response to the users when they interact (like or view) with a Tweet. The nearest servers managing various shards of the respective counters are

continuously updating the like or view counts with short refresh intervals. We should note, however, that the near real-time counts will update on the Tweets with a long refresh interval. The reason is the application server waits for multiple counts submitted by the various servers placed in different regions. We have a detailed chapter on Sharded Counters, explaining how it works in real-time applications.

The complete design overview

This section will discuss what happens in the back-end system when the end users generate multiple requests. The following are the steps:

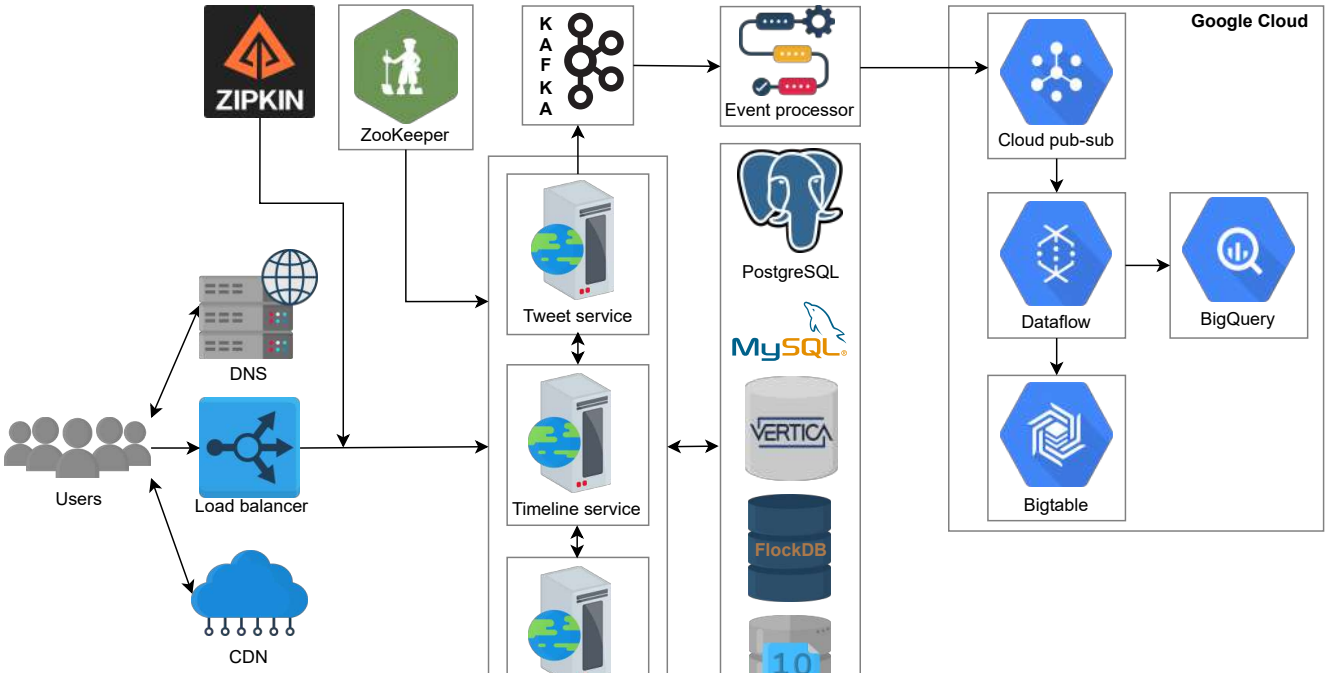
- First, end users get the address of the nearest load balancer from the local DNS.
- Load balancer routes end users' requests to the appropriate servers according to the requested services. Here, we'll discuss the Tweet, timeline, and search services.
    - **Tweet service**: When end users perform any operation, such as posting a Tweet or liking other Tweets, the load balancers forward these requests to the server handling the Tweet service. Consider an example where users post Tweets on Twitter using the `/postTweet` API. The server (Tweet service) receives the requests and performs multiple operations. It identifies the attachments (image, video) in the Tweet and stores them in the Blobstore. Text in the Tweets, user information, and all metadata are stored in the different databases (Manhattan, MySQL, PostgreSQL, Vertica). Meanwhile, real-time processing, such as pulling Tweets, user interactions data, and many other metrics from the real-time streams and client logs, is achieved in the Apache Kafka.

      Later, the data is moved to the cloud pub-sub through an event processor. Next, data is transferred for deduping and aggregation

to the BigQuery through Cloud Dataflow. Finally, data is stored in the Google Cloud Bigtable, which is fully managed, easily scalable, and sorted keys.

- **Timeline service**: Assume the user sends a home timeline request using the `/viewHome_timeline` API. In this case, the request is forwarded to the nearest CDN containing static data. If the requested data is not found, it's sent to the server providing timeline services. This service fetches data from different databases or stores and returns the Top-k Tweets. This service collects various interactions counts of Tweets from different sharded counters to decide the Top-k Tweets. In a similar way, we will obtain the Top-k trends attached in the response to the timeline request.
- **Search service**: When users type any keyword(s) in the search bar on Twitter, the search request is forwarded to the respective server using the `/searchTweet` API. It first looks into the RAM in Apache Lucene to get real-time Tweets (Tweets that have been published recently). Then, this server looks up in the index server and finds all Tweets that contain the requested keyword(s). Next, it considers multiple factors, such as time, or location, to rank the discovered Tweets. In the end, it returns the top Tweets.

- We can use the Zipkin tracing system that performs sampling on requests. Moreover, we can use ZooKeeper to maintain different data, including configuration information, distributed synchronization, naming registry, and so on.


ZIPKIN
ZooKeeper
KAFKA
Event processor
Google Cloud
Cloud pub-sub
Dataflow
BigQuery
Bigtable
PostgreSQL
MySQL
VERTICA
FlockDB
Tweet service
Timeline service
DNS
Users
Load balancer
CDN

Requirements of a Newsfeed System's Design

Get introduced to the requirements and estimation to design a newsfeed system.

We'll cover the following

Requirements

To limit the scope of the problem, we'll focus on the following functional and non-functional requirements:

Functional requirements

- **Newsfeed generation:** The system will generate newsfeeds based on pages, groups, and followers that a user follows. A user may have many friends and followers. Therefore, the system should be capable of generating feeds from all friends and followers. The challenge here is that there is potentially a huge amount of content. Our system needs to decide which content to pick for the user and rank it further to decide which to show first.

- **Newsfeed contents:** The newsfeed may contain text, images, and videos.
- **Newsfeed display:** The system should affix new incoming posts to the newsfeed for all active users based on some ranking mechanism. Once ranked, we show content to a user with higher-ranked first.

Non-functional requirements

- **Scalability:** Our proposed system should be highly scalable to support the ever-increasing number of users on any platform, such as Twitter, Facebook, and Instagram.
- **Fault tolerance:** As the system should be handling a large amount of data; therefore, partition tolerance (system availability in the events of network failure between the system's components) is necessary.
- **Availability:** The service must be highly available to keep the users engaged with the platform. The system can compromise strong consistency for availability and fault tolerance, according to the PACELC theorem.
- **Low latency:** The system should provide newsfeeds in real-time. Hence, the maximum latency should not be greater than 2 seconds.

Resource estimation

Let's assume the platform for which the newsfeed system is designed has 1 billion users per day, out of which, on average, 500 million are daily active users. Also, each user has 300 friends and follows 250 pages on average. Based on the assumed statistics, let's look at the traffic, storage, and servers estimation.

Traffic estimation

Let's assume that each daily active user opens the application (or social media page) 10 times a day. The total number of requests per day would be:

$500M \times 10 = 5$ billions request per day $\approx 58K$ requests per second.

58K Requests / Second

Traffic estimation for the newsfeed system

Storage estimation

Let's assume that the feed will be generated offline and rendered upon a request. Also, we'll precompute the top 200 posts for each user. Let's calculate storage estimates for users' metadata, posts containing text, and media content.

$1B \times 50KB = 50TB.$

We can tweak the estimated numbers and calculate the storage for our desired numbers in the following calculator:

Storage Estimation for the Users' Metadata.

Number of users (in billion)	1
Required storage for one users' metadata (in KBs)	50
Total storage required for all users (in TBs)	f 50

2. **Textual post's storage estimation:** All posts could contain some text, we assume it's 50KB on average. The storage estimation for the top 200 posts for 500 million users would be:

   $200 \times 500M \times 50KB = 5PB$

3. **Media content storage estimate:** Along with text, a post can also contain media content. Therefore, we assume that $1/5th$ posts have videos and $4/5th$ include images. The assumed average image size is 200KB and the video size is 2MB.

   Storage estimate for 200 posts of one user:
   $(200 \times 2MB \times \frac{1}{5}) + (200 \times 200KB \times \frac{4}{5}) = 80MB + 32MB = 112MB$

   Total storage required for 500 million users' posts:
   $112MB \times 500M = 56PB$

   So we'll need at least 56PB of blob storage to store the media content.

Storage required for 500 million active users per day (each with approx. 200 posts) by newsfeed system

Storage Estimation of Posts Containing Text and Media Content.

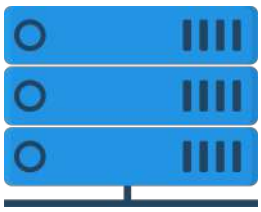
Number of active users (in million)	50(
Maximum allowed text storage per	5(

post (in KBs)	
Number of precomputed posts per user (top N)	200
Storage required for textual posts (in PBs)	f 5
Total required media content storage for active users (in PBs)	f 56

Number of servers estimation

Considering the above traffic and storage estimation, let's estimate the required number of servers for smooth operations. Recall that a single typical server can serve 8000 requests per second (RPS). Since our system will have approximately 500 million daily active users (DAU). Therefore, according to estimation in Back-of-the-Envelope Calculations chapter, the number of servers we would require is:

$$\frac{DAU}{ServerRPS} = \frac{500M}{8000} = 62500 \text{ servers.}$$



Number of servers required for the newsfeed system

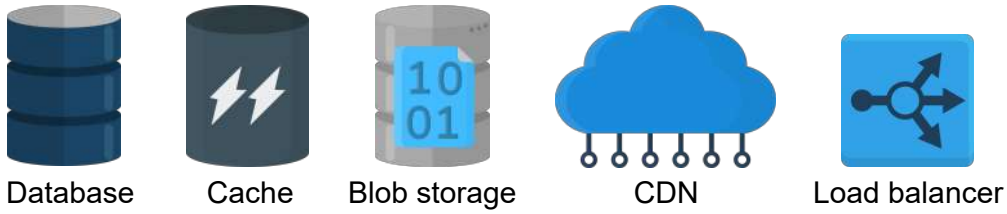
Servers Estimation

Number of active users (in million)	500
RPS of a server	8000

Number of servers required	f 62500

Building blocks we will use

The design of newsfeed system utilizes the following building blocks:



The building blocks to design a newsfeed system

- **Database(s)** is required to store the posts from different entities and the generated personalized newsfeed. It is also used to store users' metadata and their relationships with other entities, such as friends and followers.
- **Cache** is an important building block to keep the frequently accessed data, whether posts and newsfeeds or users' metadata.
- **Blob storage** is essential to store media content, for example, images and videos.
- **CDN** effectively delivers content to end-users reducing delay and burden on back-end servers.
- **Load balancers** are necessary to distribute millions of incoming clients' requests for newsfeed among the pool of available servers.

In the next lesson, we'll focus on the high-level and detailed design of the newsfeed system.

Back
System Design: Newsfeed System
Next
Design of a Newsfeed Sys
Completed

Design of a Newsfeed System

Learn how to design a newsfeed system.

We'll cover the following

Let's discuss the high-level and detailed design of a newsfeed system based on the requirements discussed in the previous lesson.

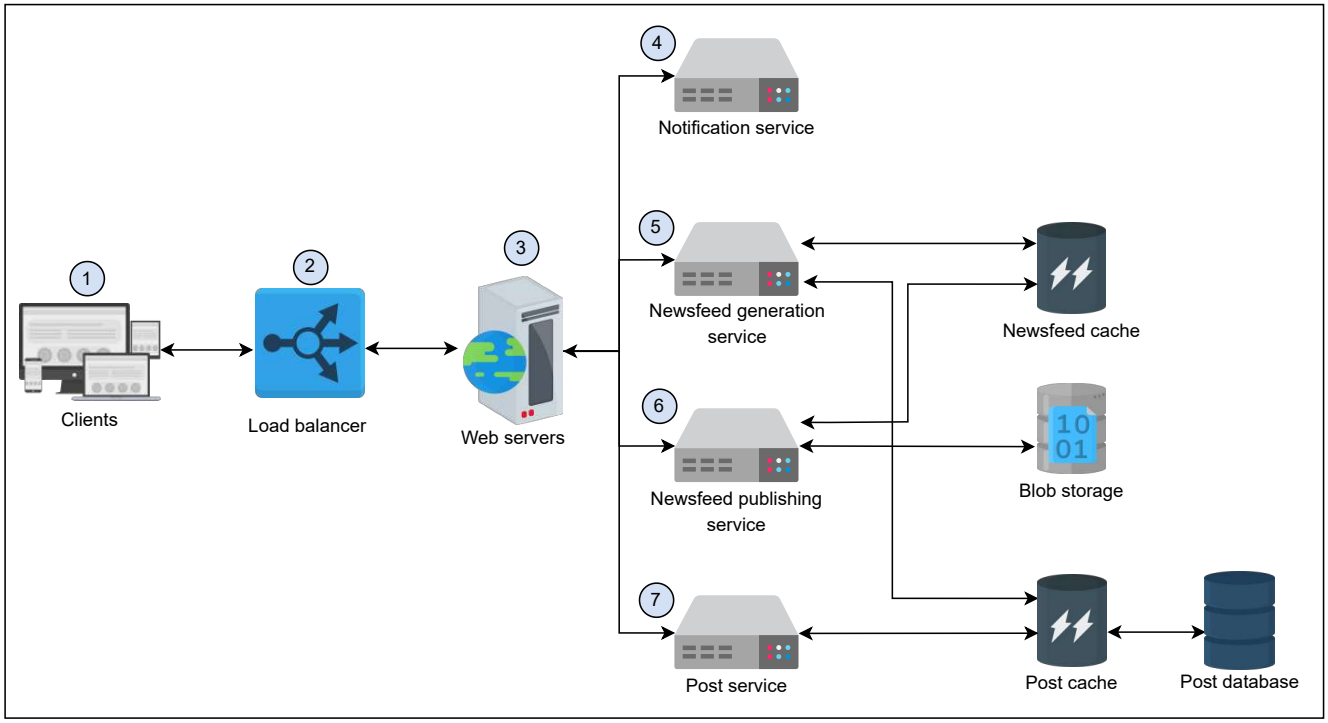
High-level design of a newsfeed system

Primarily, the newsfeed system is responsible for the following two tasks:

1. **Feed generation:** The newsfeed is generated by aggregating friends' and followers' posts (or feed items) based on some ranking mechanism

2. **Feed publishing:** When a feed is published, the relevant data is written into the cache and database. This data could be textual or any media content. A post containing the data from friends and followers is populated to a user's newsfeed.

Let's move to the high-level design of our newsfeed system. It consists of the above two essential parts, shown in the following figure:



High-level design of the Newsfeed system

Let's discuss the main components shown in the high-level design:

1. **User(s):** Users can make a post with some content or request their newsfeed.
2. **Load balancer:** It redirects traffic to one of the web servers.
3. **Web servers:** The web servers encapsulate the back-end services and work as an intermediate layer between users and various services. Apart from enforcing authentication and rate-limiting, web servers ar responsible to redirect traffic to other back-end services.

4. **Notification service:** It informs the newsfeed generation service whenever a new post is available from one's friends or followers, and sends a push notification.
5. **Newsfeed generation service:** This service generates newsfeeds from the posts of followers/friends of a user and keeps them in the newsfeed cache.
6. **Newsfeed publishing service:** This service is responsible for publishing newsfeeds to a users' timeline from the newsfeed cache. It also appends a thumbnail of the media content from the blob storage and its link to the newsfeed intended for a user.
7. **Post-service:** Whenever a user requests to create a post, the post-service is called, and the created post is stored on the post database and corresponding cache. The media content in the post is stored in the blob storage.

API design

APIs are the primary ways for clients to communicate with servers. Usually, newsfeed APIs are HTTP-based that allow clients to perform actions, including posting a status, retrieving newsfeeds, adding friends, and so on. We aim to generate and get a user's newsfeed; therefore, the following APIs are essential:

Generate user's newsfeed

The following API is used to generate a user's newsfeed:

```
generateNewsfeed(user_id)
```

This API takes users' IDs, and determines their friends and followers. This API generates newsfeeds that consist of several posts. Since internal system components use this API, therefore, it can be called offline to pre-generate

newsfeeds for users. The pre-generated newsfeeds are stored on persistent storage and associated cache.

The following parameter is used in this API call:

Parameter	Description
`user_id`	A unique identification of the user for whom the newsfeed is generated.

Get user's newsfeed

The following API is used to get a user's newsfeed:

```
getNewsfeed(user_id, count)
```

The `getNewsfeed(.)` API call returns a JSON object consisting of a list of posts.

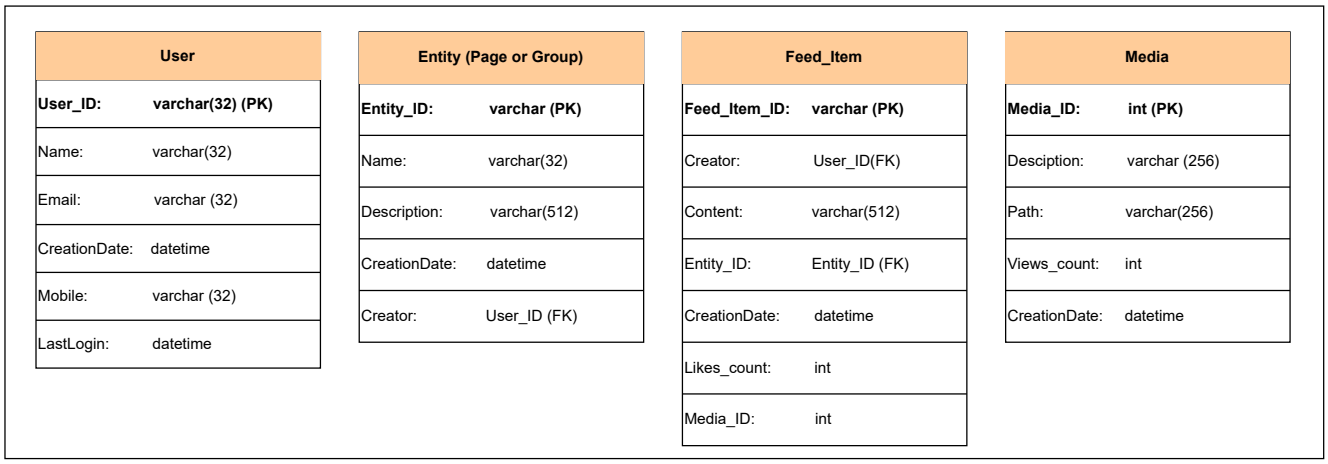
The following parameters are used for this API:

Parameter	Description
`user_id`	A unique identification of the user for whom the system will fetch the newsfeed.
`count`	The number of feed items (posts) that will be retrieved per request.

Storage schema

The database relations for the newsfeed system are as follows:

- **User:** This relation contains data about a user. A user can also be a follower or friend of other users.
- **Entity:** This relation stores data related to any entity, such as pages, groups, and so on.
- **Feed_item:** The data about posts created by users is stored in this relation.
- **Media:** The information about the media content is stored in this relation.



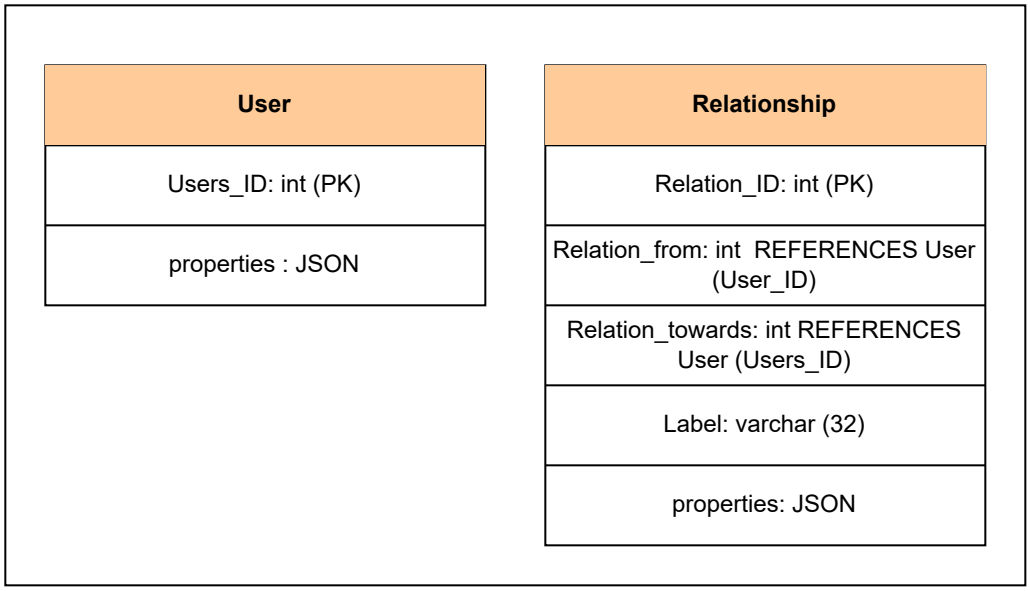
The database schema for the newsfeed system

We use a graph database to store relationships between users, friends, and followers. For this purpose, we follow the property graph model. We can think of a graph database consisting of two relational tables:

1. For vertices that represent users
2. For edges that denotes relationships among them

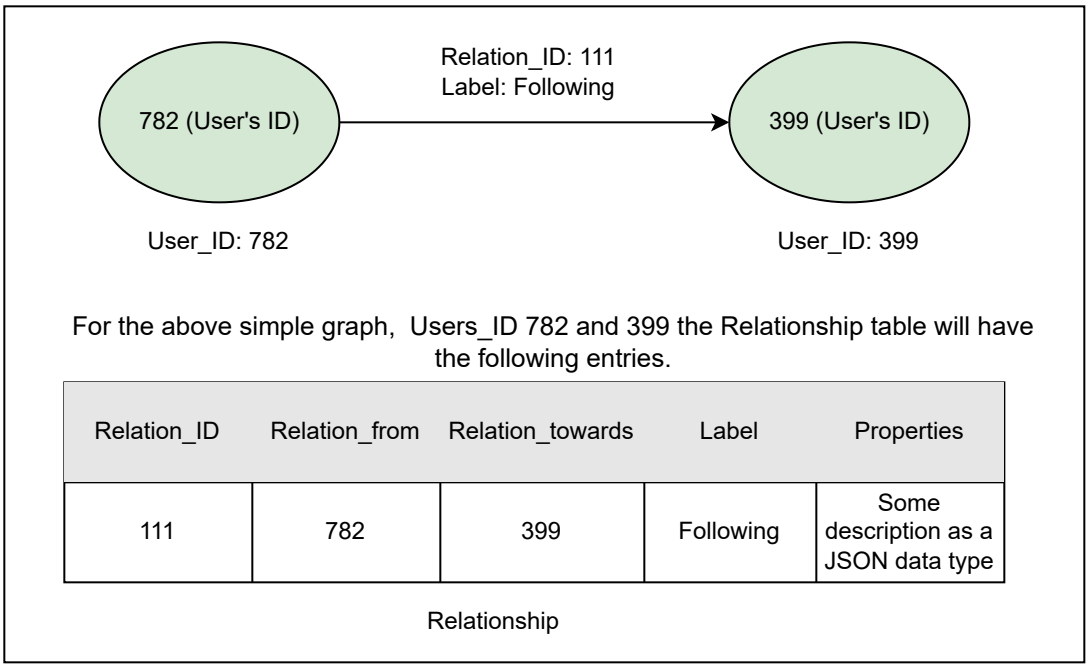
Therefore, we follow a relational schema for the graph store, as shown in the following figure. The schema uses the PostgreSQL JSON data type to store t properties of each vertex (user) or edge (relationship).

An alternative representation of a **User** can be shown in the graph database below. Where the `Users_ID` remains the same and attributes are stored in a JSON file format.



Schema of the graph database to hold relationships between users

The following figure demonstrates how graph can be represented using the relational schema:



A graph between two users consisting of two vertices and an edge

Detailed design

Let’s explore the design of the newsfeed system in detail.

As discussed earlier, there are two parts of the newsfeed system; newsfeed publishing and newsfeed generation. Therefore, we'll discuss both parts, starting with the newsfeed generation service.

The newsfeed generation service

Newsfeed is generated by aggregated posts (or feed items) from the user's friends, followers, and other entities (pages and groups).

In our proposed design, the **newsfeed generation service** is responsible for generating the newsfeed. When a request from a user (say Alice) to retrieve a newsfeed is received at web servers, the web server either:

- Calls the newsfeed generation service to generate feeds because some users don't often visit the platform, so their feeds are generated on their request.
- It fetches the pre-generated newsfeed for active users who visit the platform frequently.

The following steps are performed in sequence to generate a newsfeed for Alice:

1. The newsfeed generation service retrieves IDs of all users and entities that Alice follows from the graph database.
2. When the IDs are retrieved from the graph database, the next step is to get their friends' (followers and entities) information from the user cache, which is regularly updated whenever the **users database** gets updated/modified.
3. In this step, the service retrieves the latest, most popular, and relevant posts for those IDs from the post cache. These are the posts that we might be able to display on Alice's newsfeed.
4. The **ranking service** ranks posts based on their relevance to Alice. This represents Alice's current newsfeed.

5. The newsfeed is stored in the the newsfeed cache from which top N posts are published to Alice's timeline. (The publishing process is discussed in detail in the following section).
6. In the end, whenever Alice reaches the end of her timeline, the next top N posts are fetched to her screen from the newsfeed cache.

The process is illustrated in the following figure:

Working of the newsfeed generation service

Point to Ponder

Question

The creation and storage of newsfeeds for each user in the cache requires an enormous amount of memory (step 5 in the above section). Is there any way to reduce this memory consumption?

Show Answer

The newsfeed publishing service

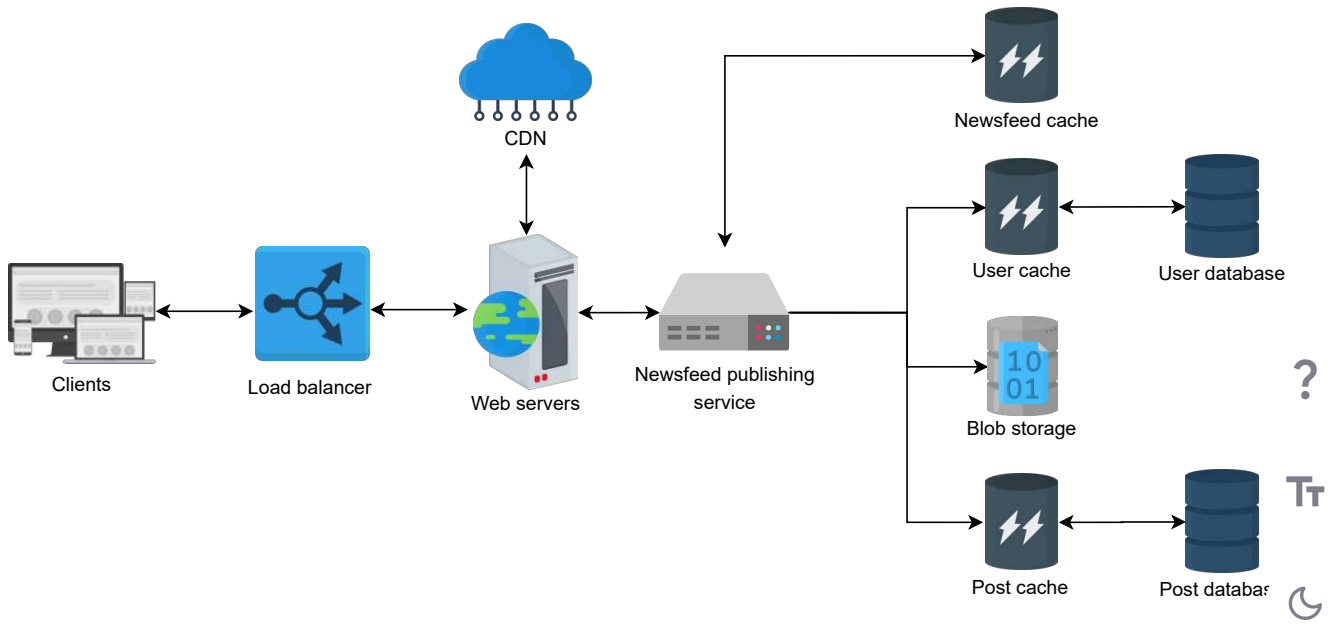
At this stage, the newsfeeds are generated for users from their respective friends, followers, and entities and are stored in the form of `<Post_ID, User_ID>` in the news feed cache.

Now the question is how the newsfeeds generated for Alice will be published to her timeline?

The **newsfeed publishing service** fetches a list of post IDs from the newsfeed cache. The data fetched from the newsfeed cache is a tuple of post and user IDs, that is., `<Post_ID, User_ID>`. Therefore, the complete data about posts and users are retrieved from the users and posts cache to create an entirely constructed newsfeed.

In the last step, the fully constructed newsfeed is sent to the client (Alice) using one of the **fan-out approaches**↴. The popular newsfeed and media content are also stored in CDN for fast retrieval.

What is the problem with generating a newsfeed upon a user's request (also called live updates)?



The newsfeed publishing service in action

Point to Ponder

Question

How is the newsfeed of the friends and followers updated when a user creates a new post?

Show Answer

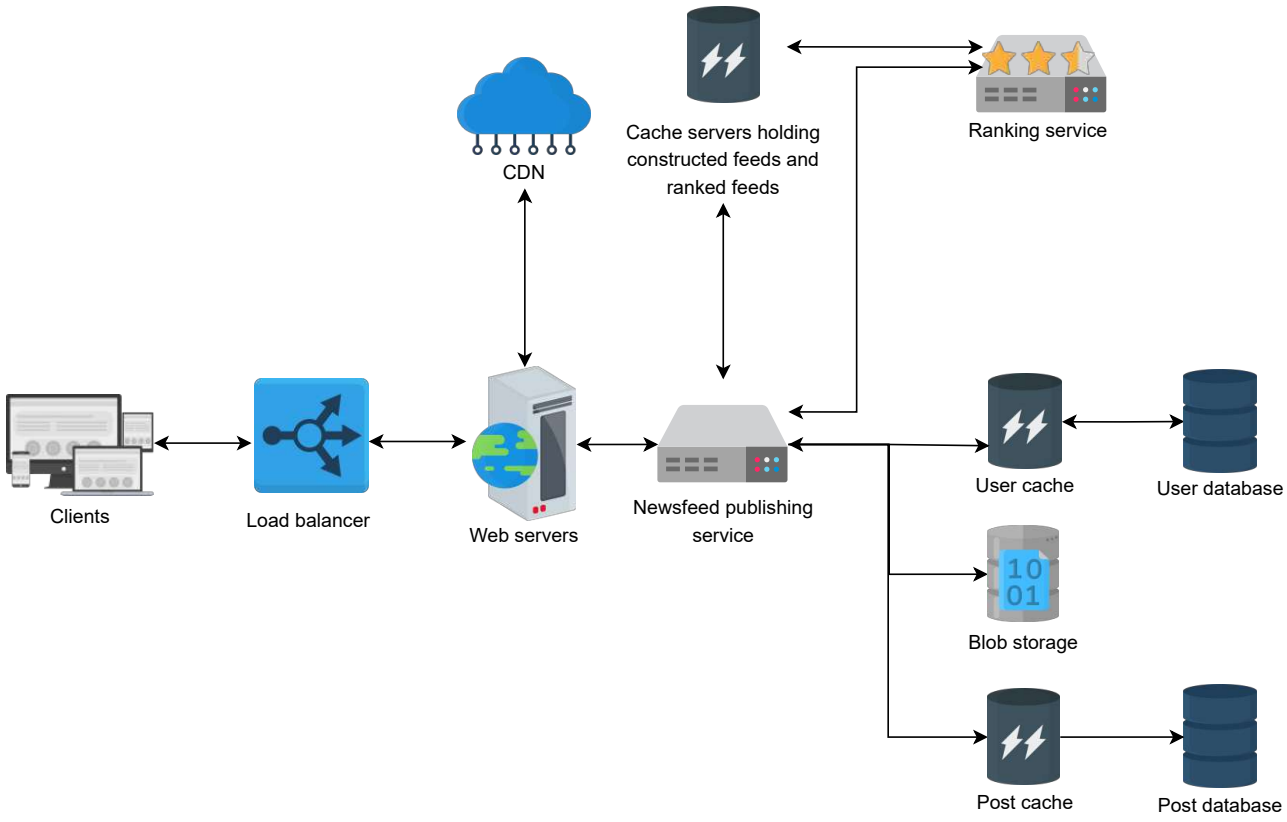
The newsfeed ranking service

Often we see the relevant and important posts on the top of our newsfeed whenever we log in to our social media accounts. This ranking involves multiple advanced ranking and recommendation algorithms.

In our design, the **newsfeed ranking service** consists of these algorithms working on various features, such as, a user's past history, likes, dislikes, comments, clicks, and many more. These algorithms also perform the following functions:

- Select "candidates" posts to show in a newsfeed.
- Eliminate posts including misinformation or clickbait from the candidate posts.
- Create a list of friends a user frequently interacts with.
- Choose topics on which a user spent more time.

The ranking system considers all the above points to predict relevant and important posts for a user.



The ranked newsfeeds are stored in the cache servers

Posts ranking and newsfeed construction

The post database contains posts published by different users. Assume that there are 10 posts in the database published by 5 different users. We aim to rank only 4 posts out of 10 for a user (say Bob) who follows those five different users. We perform the following to rank each post and create a newsfeed for Bob:

1. Various features such as likes, comments, shares, category, duration, e and so on, are extracted from each post.
2. Based on Bob's previous history, stored in the **user database**, the relevance is calculated for each post via different ranking and machine learning algorithms.

3. A relevance score is assigned, say from 1 to 5, where 1 shows the least relevant post and 5 means highly relevant post.
4. The top 4 posts are selected out of 10 based on the assigned scores.
5. The top 4 posts are combined and presented on Bob's timeline in decreasing order of the score assigned.

The following figure shows the top 4 posts published on Bob's timeline:

A newsfeed consisting of top 4 posts based on the relevance scores

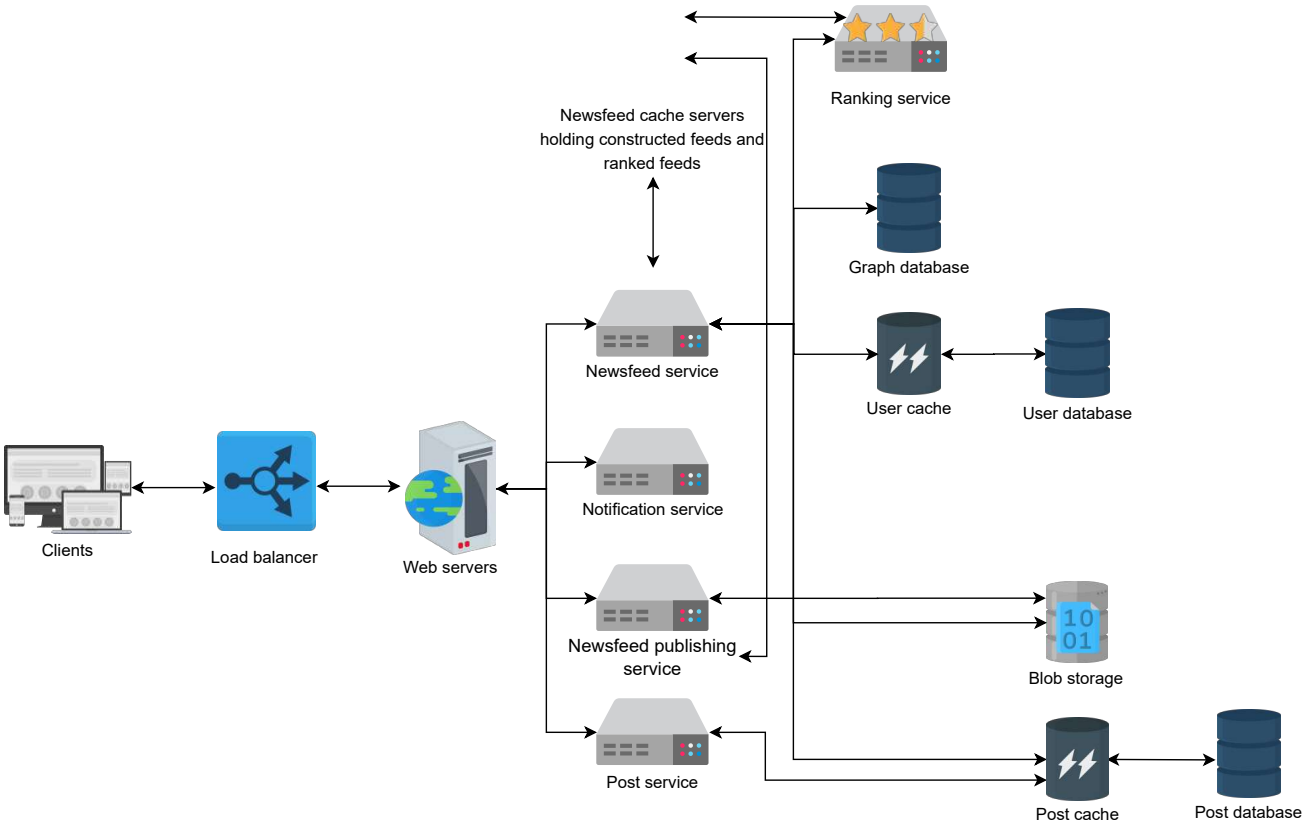
Newsfeed ranking with various machine learning and ranking algorithms is a computationally intensive task. In our design, the **ranking service** ranks posts and constructs newsfeeds. This service consists of big-data processin systems that might utilize specialized hardware like graphics processing units (GPUs) and tensor processing units (TPUs).

> **Note: According to Facebook:**

> “For each person on Facebook, we need to evaluate thousands of features to determine what that person might find most relevant to predict what each of those people wants to see in their feed.”
>
> This implies that we need enormous computational power and sophisticated learning algorithms to incorporate all the features to get good quality feed in a reasonably short time.

Putting everything together

The following figure combines all the services related to the detailed design of the newsfeed system:



The detailed design of newsfeed system

In this lesson, we discussed the design of the newsfeed system, its database schema, and the newsfeed ranking system. In the next lesson, we’ll evaluate our system’s requirements.

Requirements of Instagram's Design

Learn about the requirements and computational needs for the Instagram service.

We'll cover the following

Requirements

We'll concentrate on some important features of Instagram to make this design simple. Let's list down the requirements for our system:

Functional requirements

- **Post photos and videos**: The users can post photos and videos on Instagram.
- **Follow and unfollow users**: The users can follow and unfollow other users on Instagram.

- **Like or dislike posts**: The users can like or dislike posts of the accounts they follow.
- **Search photos and videos**: The users can search photos and videos based on captions and location.
- **Generate news feed**: The users can view the news feed consisting of the photos and videos (in chronological order) from all the users they follow. Users can also view suggested and promoted photos in their news feed.

Non-functional requirements

- **Scalability**: The system should be scalable to handle millions of users in terms of computational resources and storage.
- **Latency**: The latency to generate a news feed should be low.
- **Availability**: The system should be highly available.
- **Durability** Any uploaded content (photos and videos) should never get lost.
- **Consistency**: We can compromise a little on consistency. It is acceptable if the content (photos or videos) takes time to show in followers' feeds located in a distant region.
- **Reliability**: The system must be able to tolerate hardware and software failures.

Resource estimation

Our system is read-heavy because service users spend substantially more time browsing the feeds of others than creating and posting new content. Our focus will be to design a system that can fetch the photos and videos on time. There is no restriction on the number of photos or videos that users can upload, meaning efficient storage management should be a primary

billion users globally who share 95 million photos and videos on Instagram per day. We'll calculate the resources and design our system based on the requirements.

Let's assume the following:

- We have 1 billion users, with 500 million as daily active users.
- Assume 60 million photos and 35 million videos are shared on Instagram per day.
- We can consider 3 MB as the maximum size of each photo and 150 MB as the maximum size of each video uploaded on Instagram.
- On average, each user sends 20 requests (of any type) per day to our service.

Storage estimation

We need to estimate the storage capacity, bandwidth, and the number of servers to support such an enormous number of users and content.

The storage per day will be:

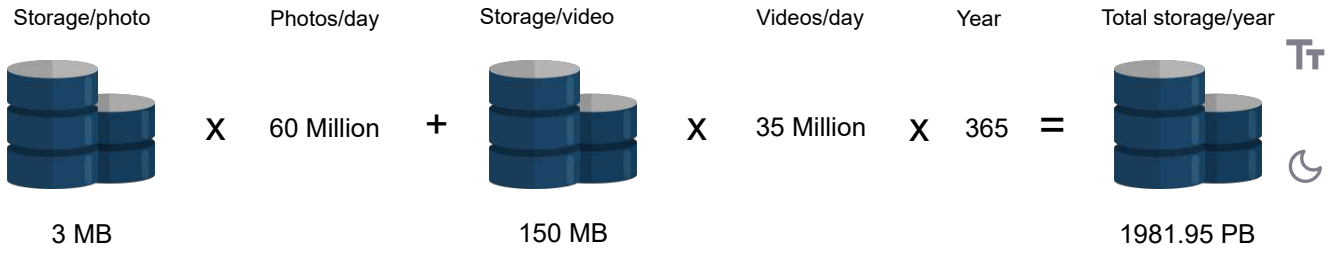
60 million photos/day * 3 MB = 180 TeraBytes / day

35 million videos/day * 150 MB = 5250 TB / day

Total content size = 180 + 5250 = 5430 TB

The total space required for a year:

5430 TB/day * 365 (days a year) = 1981950 TB = 1981.95 PetaBytes



Total storage required by the Instagram storage system for a year

Besides photos and videos, we have ignored comments, status sharing data, and so on. Moreover, we also have to store users' information and post metadata, for example, userID, photo, and so on. So, to be precise, we need more than 5430 TB/day, but to keep our design simple, let's stick to the 5430 TB/day.

Bandwidth estimation

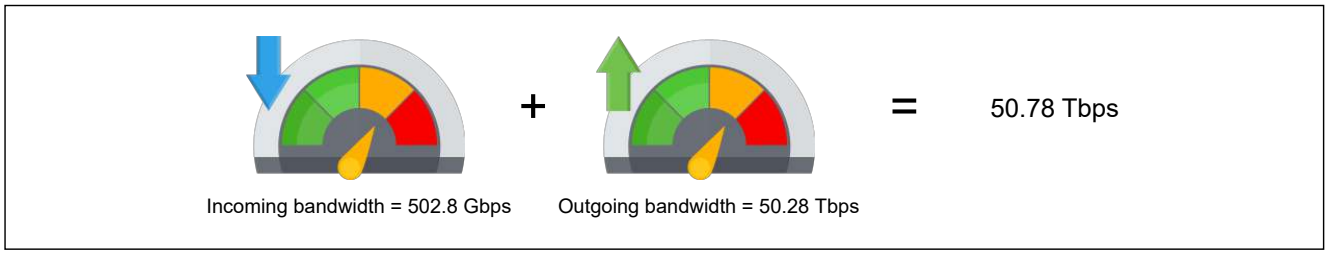
According to the estimate of our storage capacity, our service will get 5430 TB of data each day, which will give us:

5430 TB/(24 * 60* 60) = 5430 TB/86400 sec ~= 62.84 GB/s ~= 502.8 Gbps

Since each incoming photo and video needs to reach the users' followers, let's say the ratio of readers to writers is 100:1. As a result, we need 100 times more bandwidth than incoming bandwidth. Let's assume the following:

Incoming bandwidth ~= 502.8 Gbps

Required outgoing bandwidth ~= 100 * 502.8 Gbps ~= 50.28 Tbps



Total bandwidth required

Outgoing bandwidth is fairly high. We can use compression to reduce the media size substantially. Moreover, we'll place content close to the users via CDN and other caches in IXP and ISPs to serve the content at high speed and low latency.

Number of servers estimation

We need to handle concurrent requests coming from 500 million daily active users. Let's assume that a typical Instagram server handles 100 requests per second:

Requests by each user per day = 20 Queries handled by a server per second = 100

Queries handled by a server per day = 100 * 60 * 60 * 24 = 8640000

$$\frac{Number\ of\ active\ users * Requests\ by\ each\ user\ per\ day}{Queries\ handled\ per\ server\ a\ day} = 1157\ servers\ approx$$

This calculation shows that we need 1157 servers to handle queries in our Instagram system.

Number of servers required for the Instagram system

Try it yourself

Let's analyze how the number of photos per day affects the storage and bandwidth requirements. For this purpose, try to change values in the following table to compute the estimates. We have set 20 requests by each user per day in the following calculation:

Requirements	Calculations

Number of photos per day (in millions)	60
Size of each photo (in MB)	3
Number of videos per day (in millions)	35
Size of each video (in MB)	150
Number of daily active users (in millions)	500
Number of requests a server can handle per day	8640000
Storage estimation per day (in TB)	f 5430
Incoming bandwidth (Gb/s)	f 502.8
Number of servers needed	f 1157

Building blocks we will use

In the next lesson, we'll focus on the high-level design of Instagram. The design will utilize many building blocks that have been discussed in the initial chapters also. We'll use the following building blocks in our design:

Building blocks used in Instagram design

- A **load balancer** at various layers will ensure smooth requests distribution among available servers.
- A **database** is used to store the user and accounts metadata and relationship among them.
- **Blob storage** is needed to store the various types of content such as photos, videos, and so on.
- A **task scheduler** schedules the events on the database such as removing the entries whose time to live exceeds the limit.
- A **cache** stores the most frequent content related requests.
- **CDN** is used to effectively deliver content to end users which reduces delay and burden on end-servers.

In the next lesson, we'll discuss the high-level design of the Instagram system.

Back

System Design: Instagram

Next

Design of Instagram

Completed

Design of Instagram

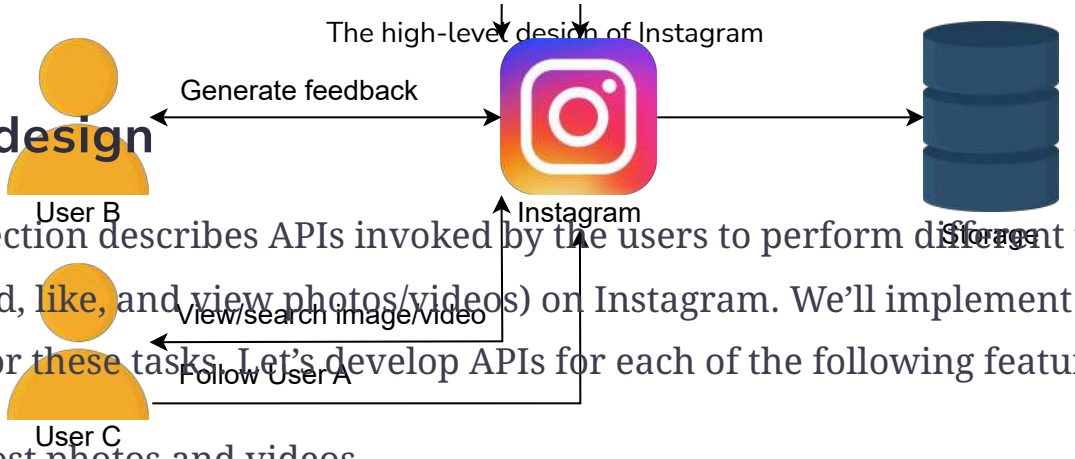
Learn the high-level design of Instagram and understand its data model.

We'll cover the following

- High-level design
- API design
  - Post photos or videos
  - Follow and unfollow users
  - Like or dislike posts
  - Search photos or videos
  - Generate news feed
- Storage schema
  - Relational or non-relational database
  - Define tables
  - Data estimation

High-level design

Our system should allow us to upload, view, and search images and videos at a high level. To upload images and videos, we need to store them, and upon



API design

This section describes APIs invoked by the users to perform different tasks (upload, like, and view photos/videos) on Instagram. We'll implement REST APIs for these tasks. Let's develop APIs for each of the following features:

- Post photos and videos
- Follow and unfollow users
- Like or dislike posts
- Search photos and videos
- Generate a news feed

All of the following calls will have a `userID`, that uniquely specifies the user performing the action. We'll only discuss new parameters in the calls.

Post photos or videos

The POST method is used to post photos/videos to the server from the user through the `/postMedia` API. The `/postMedia` API is as follows:

```
postMedia(userID, media_type, list_of_hashtags, caption)
```

Parameter	Description
`media_type`	It indicates the type of media (photo or video) in a post.
`list_of_hashtags`	It represents all hashtags (maximum limit 30 hashtags) in a post.
`caption`	It is a text (maximum limit is 2,200 characters) in a user's post.

Follow and unfollow users

The `/followUser` API is used when a user follows other users on Instagram. The `/followUser` API is as follows:

```
followUser(userID, target_userID)
```

Parameter	Description
`target_userID`	It indicates the user to be followed.

The `/unfollowUser` API uses the same parameters when a user unfollows someone on Instagram.

Like or dislike posts

The `/likePost` API is used when users like someone's post on Instagram.

```
likePost(userID, target_userID, post_id)
```

Parameter	Description
`target_userID`	It specifies the user whose post is liked.
`post_id`	It specifies the post's unique ID.

The `/dislikePost` API uses the same parameters when a user dislikes someone's post on Instagram.

Search photos or videos

The GET method is used when the user searches any photos or videos using a keyword or hashtag. The `/searchPhotos` API is as follows:

```
searchPhotos(userID, keyword)
```

Parameter	Description
`keyword`	It indicates the string (username, hashtag, and places) typed by the user in the search bar.

> **Note:** Instagram shows the posts with the highest reach (those with more likes and views) upon searching for a particular key. For example, if a user does a location-based search using “London, United Kingdom," Instagram will show the posts in order with maximum to minimum reach. Instead of showing all the posts, the data will be loaded upon scrolling.

Generate news feed

The GET method is used when users view their news feed through the `/viewNewsfeed` API. The `/viewNewsfeed` API is as follows:

```
viewNewsfeed(userID, generate_timeline)
```

Parameter	Description
`generate_timeline`	It indicates the time when a user requests news feed generation. Instagram shows the posts that are not seen by

the user between the last news feed request and the curren

Storage schema

Let's define our data model now:

Relational or non-relational database

It is essential to choose the right kind of database for our Instagram system, but which is the right choice — SQL or NoSQL? Our data is inherently relational, and we need an order for the data (posts should appear in chronological order) and no data loss even in case of failures (data durability). Moreover, in our case, we would benefit from relational queries like fetching the followers or images based on a user ID. Hence, SQL-based databases fulfill these requirements.

So, we'll opt for a relational database and store our relevant data in that database.

Define tables

On a basic level, we need the following tables:

- **Users**: This stores all user-related data such as ID, name, email, bio, location, date of account creation, time of the last login, and so on.
- **Followers**: This stores the relations of users. In Instagram, we have a unidirectional relationship, for example, if user A accepts a follow request from user B, user B can view user A's post, but vice versa is not valid.
- **Photos**: This stores all photo-related information such as ID, location, caption, time of creation, and so on. We also need to keep the user ID to determine which photo belongs to which user. The user ID is a foreign key from the users table.

- **Videos**: This stores all video-related information such as ID, location, caption, time of creation, and so on. We also need to keep the user ID to determine which video belongs to which user. The user ID is a foreign key from the users table.

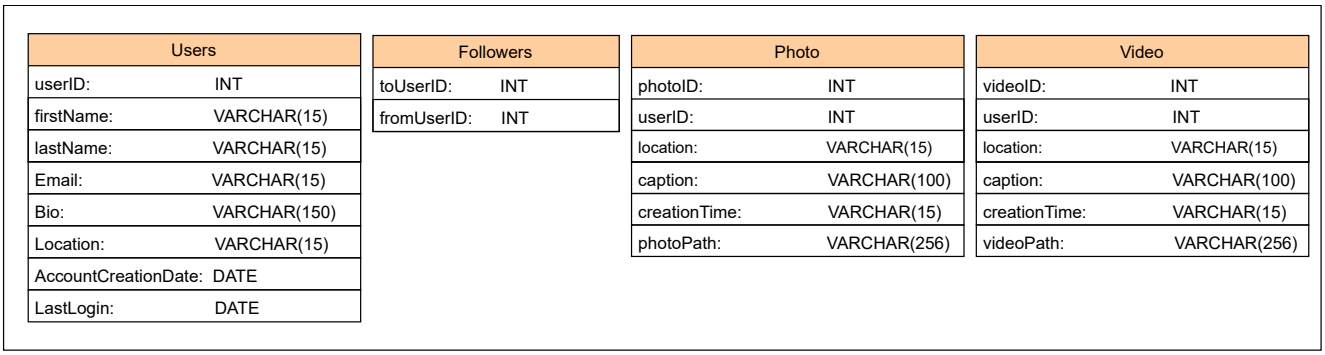
Point to Ponder

Question

Where should we store the photos and videos?

Show Answer

The following illustration visualizes the data model:



The Data model of Instagram

Data estimation

Let’s figure out how much data each table will store. The column per row size (in bytes) in the following calculator shows the data per row of each table. It also calculates the storage needed for the specified counts. For

example, the storage required for 500 million users is 111000 MB, 2000 MB for 250 followers of a single user, 23640 MB for 60 million photos, and 13790 MB for 35 million videos.

You can change the values in the calculator to observe the change in storage needed. This gives us an estimate of how fast data will increase in our tables.

Table Name	Per row size (in bytes)	Count in Millions	Storage Needed (in MBs)
Users	222	500	f 111000
Followers	8	250	f 2000
Photos	394	60	f 23640
Videos	394	35	f 13790

> **Note:** Most modern services use both SQL and NoSQL stores. Instagram officially uses a combination of SQL (PostgreSQL) and No-SQL (Cassandra) databases. The loosely structured data like timeline generation is usually stored in No-SQL, while relational data is saved in SQL-based storage.

In the next lesson, we'll identify more components to tweak our design.

← Back

Requirements of Instagram's Design

Next →

Detailed Design of Instag

Completed

Detailed Design of Instagram

Explore the design of Instagram in detail and understand the interaction of various components.

We'll cover the following

Add more components

Let's add a few more components to our design:

- **Load balancer**: To balance the load of the requests from the end users.
- **Application servers**: To host our service to the end users.
- **Relational database**: To store our data.
- **Blob storage**: To store the photos and videos uploaded by the users.

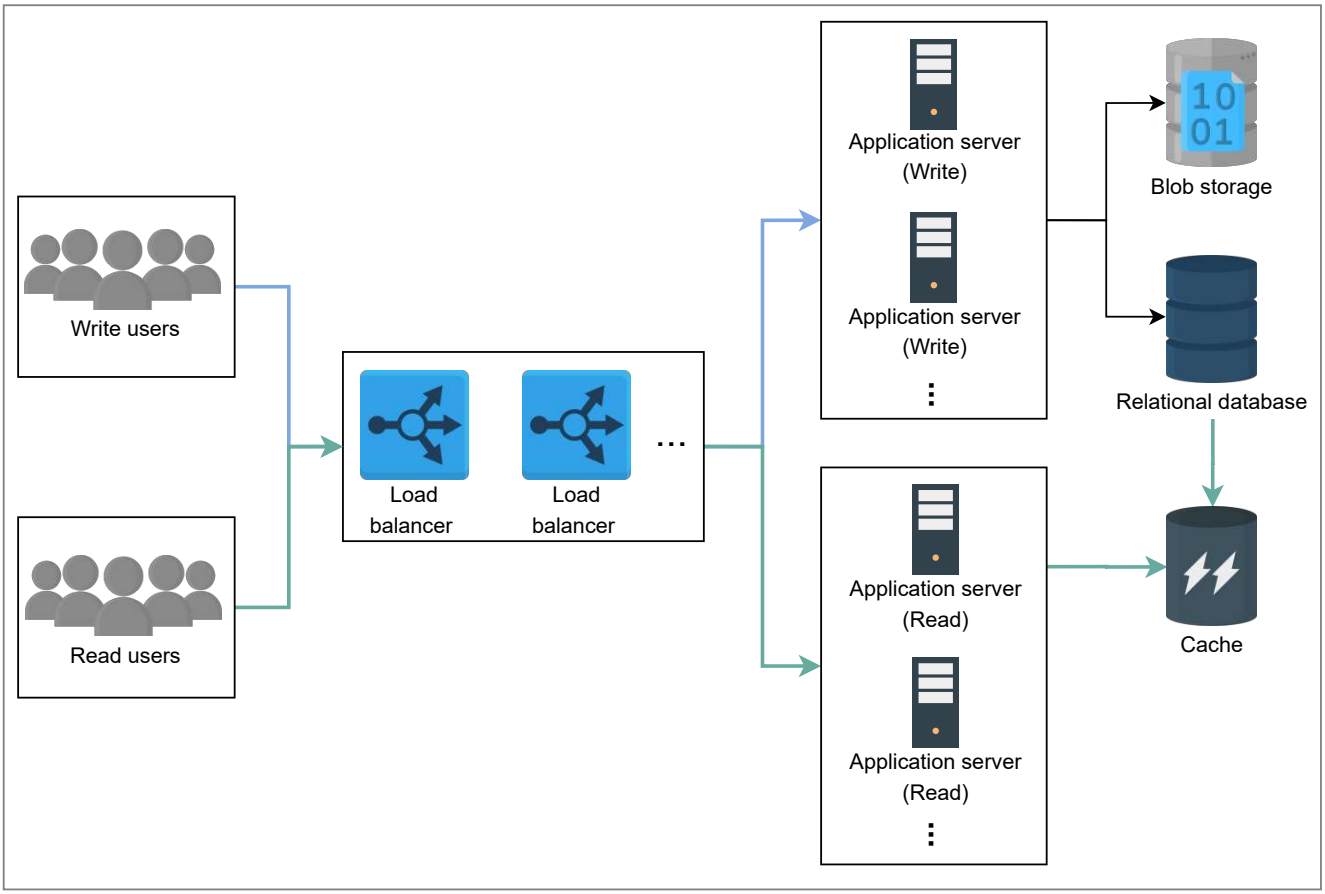
The client requests to upload the photo, load balancer passes the request to any of the application servers, which adds an entry to the database. An update that the photo is stored successfully is sent to the user. If an error is encountered, the user is communicated about it as well.

The photo viewing process is also similar to the above-mentioned flow. The client requests to view a photo, and an appropriate photo that matches the request is fetched from the database and shown to the user. The client can also provide a keyword to search for a specific image.

The read requests are more than write requests and it takes time to upload the content in the system. It is efficient if we separate the write (uploads) and read services. The multiple services operated by many servers handle the relevant requests. The read service performs the tasks of fetching the required content for the user, while the write service helps upload content to the system.

We also need to cache the data to handle millions of reads. It improves the user experience by making the fetching process fast. We'll also opt for lazy loading, which minimizes the client's waiting time. It allows us to load the content when the user scrolls and therefore save the bandwidth and focus on loading the content the user is currently viewing. It improves the latency to view or search a particular photo or video on Instagram.

The updated design is as follows:



Various operations on photos

Generate a timeline

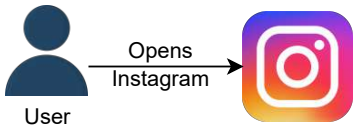
Now our task is to generate a user-specific timeline. Let's explore various approaches and the advantages and disadvantages to opt for the appropriate strategy.

The pull approach

When a user opens their Instagram, we send a request for timeline generation. First, we fetch the list of people the user follows, get the photos they recently posted, store them in queues, and display them to the user. But this approach is slow to respond as we generate a timeline every time the user opens Instagram.

We can substantially reduce user-perceived latency by generating the timeline offline. For example, we define a service that fetches the relevant

data for the user before, and as the person opens Instagram, it displays the timeline. This decreases the latency rate to show the timeline. Let's take a look at the slides below to understand the problem and its solution.



A user opens Instagram

1 of 5

Point to Ponder

Question

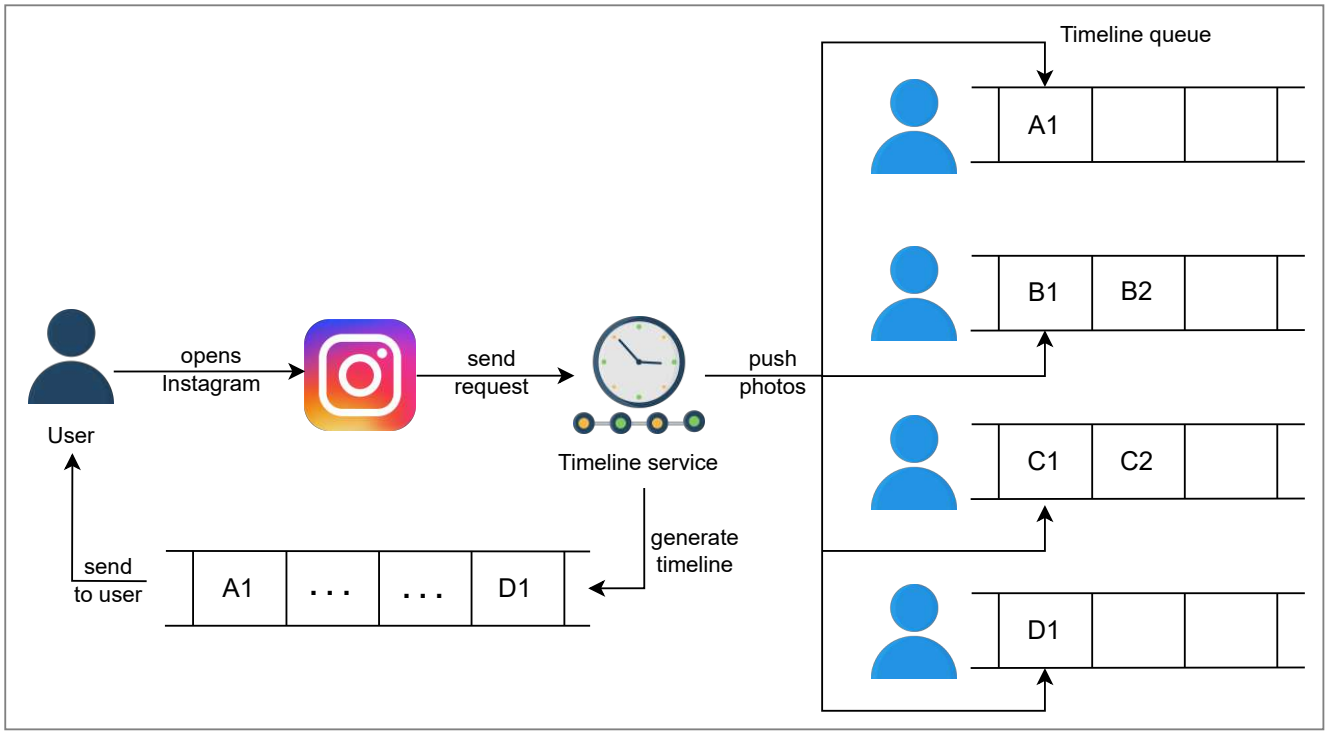
What are the shortcomings of the pull approach?

Show Answer

The push approach

In a **push approach**, every user is responsible for pushing the content they posted to the people's timelines who are following them. In the previous approach, we pulled the post from each follower, but in the current approach, we push the post to each follower.

Now we only need to fetch the data that is pushed towards that particular user to generate the timeline. The push approach has stopped a lot of requests that return empty results when followed users have no post in a specified time.



Push-based approach

Point to Ponder

Question

What are the shortcomings of the push approach?

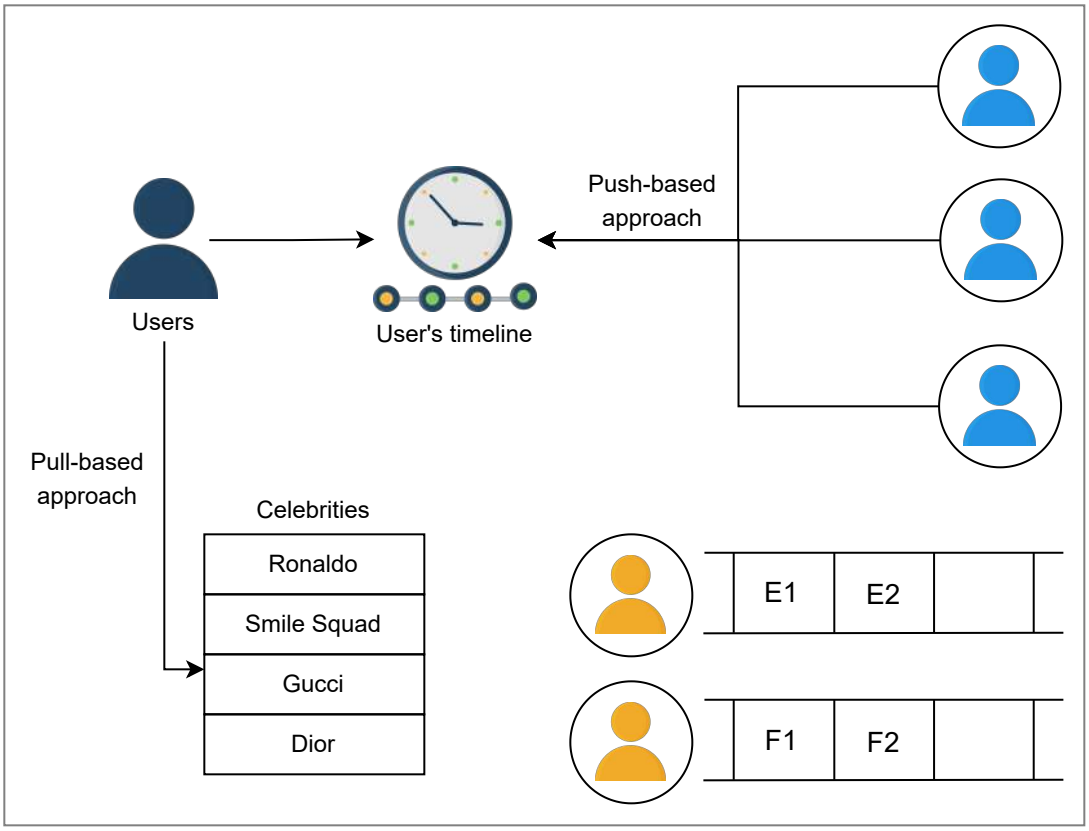
Show Answer

Hybrid approach

Let’s split our users into two categories:

- **Push-based users**: The users who have a followers count of hundreds or thousands.
- **Pull-based users**: The users who are celebrities and have followers count of a hundred thousand or millions.

The timeline service pulls the data from pull-based followers and adds it to the user’s timeline. The push-based users push their posts to the timeline service of their followers so the timeline service can add to the user’s timeline.



Hybrid approach

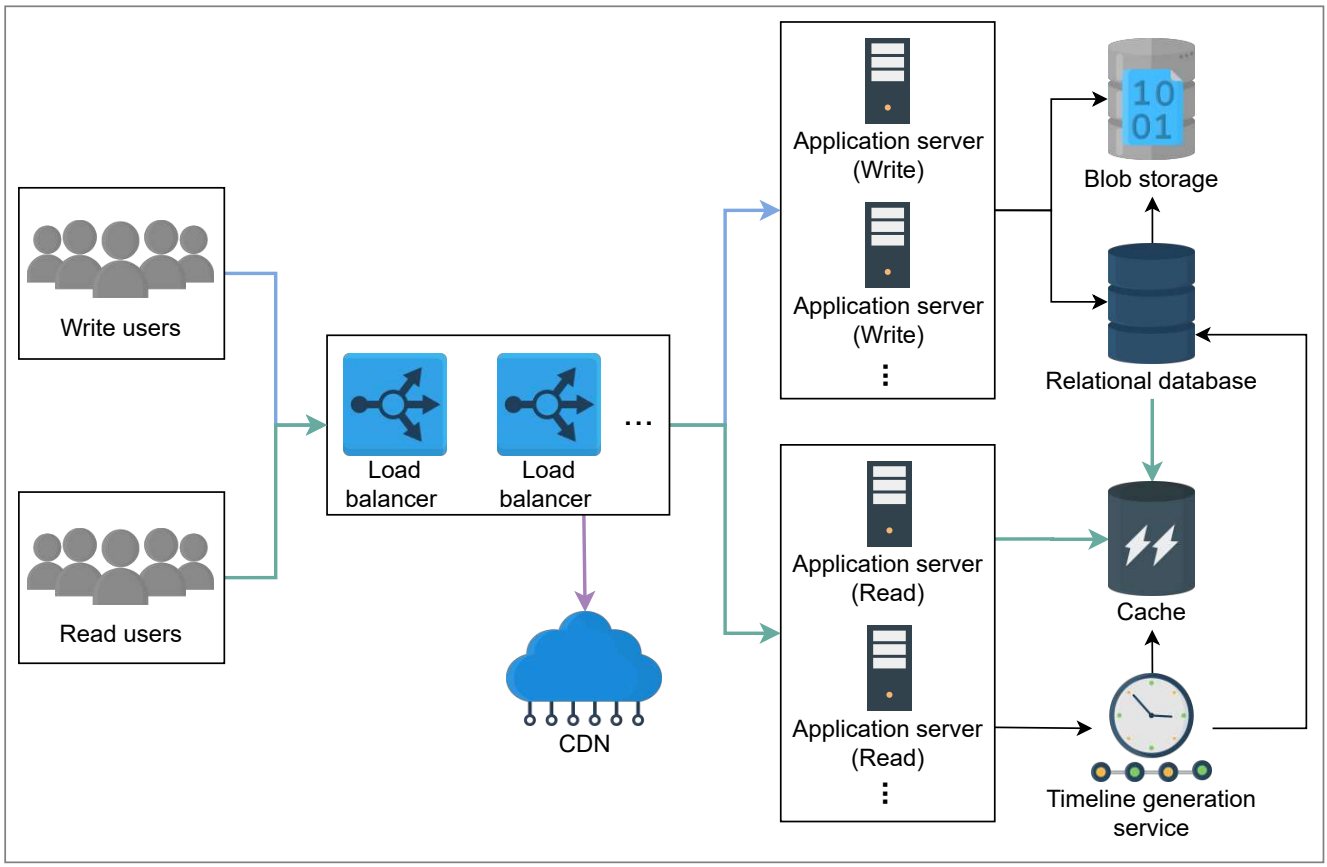
We have used the method which generates the timeline, but where do we store the timeline? We store a user's timeline against a `userID` in a key-value store. Upon request, we fetch the data from the key-value store and show it to the user. The key is `userID`, while the value is timeline content (links to photos and videos). Because the storage size of the value is often limited to a few MegaBytes, we can store the timeline data in a blob and put the link to the blob in the value of the key as we approach the size limit.

We can add a new feature called story to our Instagram. In the story feature, the users can add a photo that stays available for others to view for 24 hours only. We can do this by maintaining an option in the table where we can store a story's duration. We can set it to 24 hours, and the task scheduler deletes the entries whose time exceeds the 24 hours limit.

Finalized design

We'll also use **CDN (content delivery network)** in our design. We can keep images and videos of celebrities in CDN which make it easier for the followers to fetch them. The load balancer first routes the read request to the nearest CDN, if the requested content is not available there, then it forwards the request to the particular read application server (see the "load balancing chapter" for the details). The CDN helps our system to be available to millions of concurrent users and minimizes latency.

The final design is given below:



The final design of Instagram

Point to Ponder

Question

How can we count millions of interactions (like or view) on a celebrity post?

Show Answer

Ensure non-functional requirements

We evaluate the Instagram design with respect to its non-functional requirements:

- **Scalability**: We can add more servers to application service layers to make the scalability better and handle numerous requests from the clients. We can also increase the number of databases to store the growing users' data.
- **Latency**: The use of cache and CDNs have reduced the content fetching time.
- **Availability**: We have made the system available to the users by using the storage and databases that are replicated across the globe.
- **Durability:** We have persistent storage that maintains the backup of the data so any uploaded content (photos and videos) never gets lost.
- **Consistency**: We have used storage like blob stores and databases to keep our data consistent globally.
- **Reliability**: Our databases handle replication and redundancy, so our system stays reliable and data is not lost. The load balancing layer routes requests around failed servers.

Conclusion

This design problem highlights that we can provide major services by connecting our building blocks appropriately. The scalable and fault-tolerant building blocks enable us to concentrate on use-case-specific issues (such as the efficient formation of timelines).

Requirements of WhatsApp's Design

Learn about the functional and non-functional requirements for a chat application like WhatsApp.

We'll cover the following

Requirements

Our design of the WhatsApp messenger should meet the following requirements.

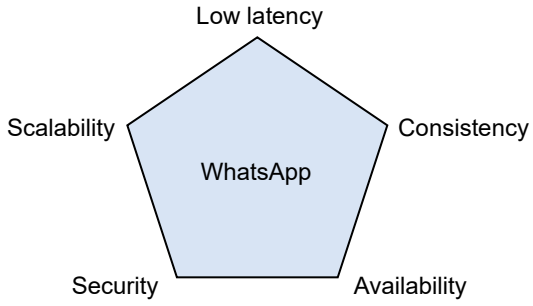
Functional requirements

- **Conversation**: The system should support one-on-one and group conversations between users.
- **Acknowledgment**: The system should support message delivery acknowledgment, such as sent, delivered, and read.

- **Sharing**: The system should support sharing of media files, such as images, videos, and audio.
- **Chat storage**: The system must support the persistent storage of chat messages when a user is offline until the successful delivery of messages.
- **Push notifications**: The system should be able to notify offline users of new messages once their status becomes online.

Non-functional requirements

- **Low latency**: Users should be able to receive messages with low latency.
- **Consistency**: Messages should be delivered in the order they were sent. Moreover, users must see the same chat history on all of their devices.
- **Availability**: The system should be highly available. However, the availability can be compromised in the interest of consistency.
- **Security**: The system must be secure via end-to-end encryption. The end-to-end encryption ensures that only the two communicating parties can see the content of messages. Nobody in between, not even WhatsApp, should have access.
- **Scalability:** The system should be highly scalable to support an ever-increasing number of users and messages per day.



The non-functional requirements of the WhatsApp system

Resource estimation

WhatsApp is the most used messaging application across the globe. According to WhatsApp, it supports more than two billion users around the world who share more than 100 billion messages each day. We need to estimate the storage capacity, bandwidth, and number of servers to support such an enormous number of users and messages.

Storage estimation

As there are more than 100 billion messages shared per day over WhatsApp

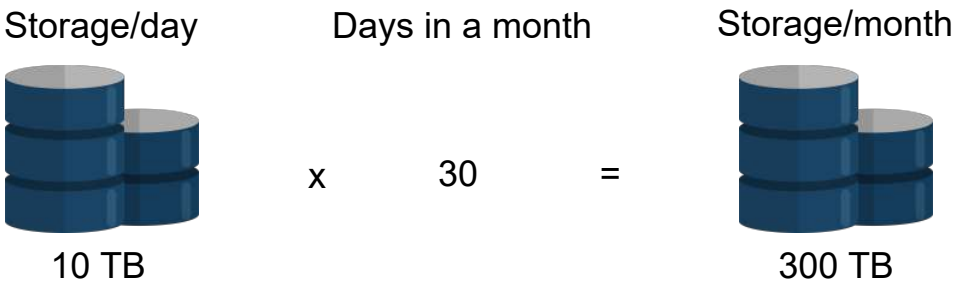
message takes 100 Bytes on average. Moreover, the WhatsApp servers keep the messages only for 30 days. So, if the user doesn't get connected to the server within these days, the messages will be permanently deleted from the server.

$$100\ billion/day * 100\ Bytes = 10\ TB/day$$

For 30 days, the storage capacity would become the following:

$$30 * 10\ TB/day = 300\ TB/month$$

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, but for the sake of simplicity, let's stick to the number 300 TB per month.



The total storage required by WhatsApp in a month

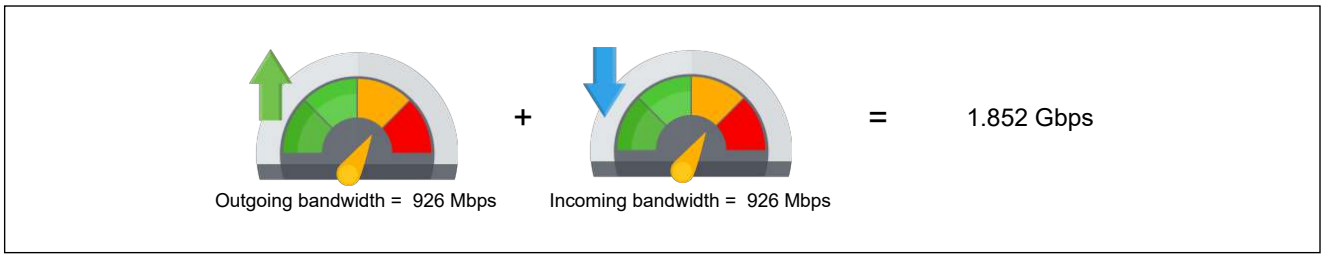
Bandwidth estimation

According to the storage capacity estimation, our service will get 10TB of data each day, giving us a bandwidth of 926 Mb/s.

$10\ TB/86400sec \approx 926\ Mb/s$

> **Note:** To keep our design simple, we've ignored the media content (images, videos, documents, and so on). So, the number 926 might seem low.

We also require an equal amount of outgoing bandwidth as the same message from the sender would need to be delivered to the receiver.



The total bandwidth required by WhatsApp

High-level Estimates

Type	Estimates
Total messages per day	100 billion
Storage required per day	10 TB
Storage for 30 days	300 TB
Incoming data per second	926 Mb/s

Outgoing data per second	926 Mb/s

Number of servers estimation

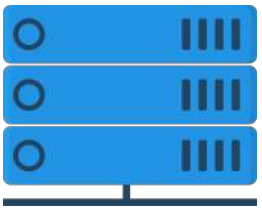
WhatsApp handles around 10 million connections on a single server, which seems quite high for a server. However, it's possible by extensive performance engineering. We'll need to know all the in-depth details of a system, such as a server's kernel, networking library, infrastructure configuration, and so on.

> **Note:** We can often optimize a general-purpose server for special tasks by careful performance engineering of the full software stack.

Let's move to the estimation of the number of servers:

$$No.\ of\ servers = \\ Total\ connections\ per\ day / No.\ of\ connections\ per\ server = \\ 2\ billion / 10\ million = 200\ servers$$

So, according to the above estimates, we require 200 chat servers.



Number of chat servers required for WhatsApp

Try it out

Let's analyze how the number of messages per day affects the storage and bandwidth requirements. For this purpose, we can change values in the following table to compute the estimates:

Number of users per day (in billions)	2
Number of messages per day (in billions)	100
Size of each message (in bytes)	100
Number of connections a server can handle (in millions)	10
Storage estimation per day (in TB)	f 10
Incoming and Outgoing bandwidth (Mb/s)	f 926.4
Number of chat servers required	f 200

Building blocks we will use

The design of WhatsApp utilizes the following building blocks that have also been discussed in the initial chapters:

The building blocks required to design WhatsApp

- **Databases** are required to store users' and groups' metadata.
- **Blob storage** is used to store multimedia content shared in messages.

- A **CDN** is used to effectively deliver multimedia content that's frequently shared.
- A **load balancer** distributes incoming requests among the pool of available servers.
- **Caches** are required to keep frequently accessed data used by various services.
- A **messaging queue** is used to temporarily keep messages in a queue on a database when a user is offline.

In the next lesson, we'll focus on the high-level design of the WhatsApp messenger.

Back

System Design: WhatsApp

Next

High-level Design of WhatsApp

Mark as Completed

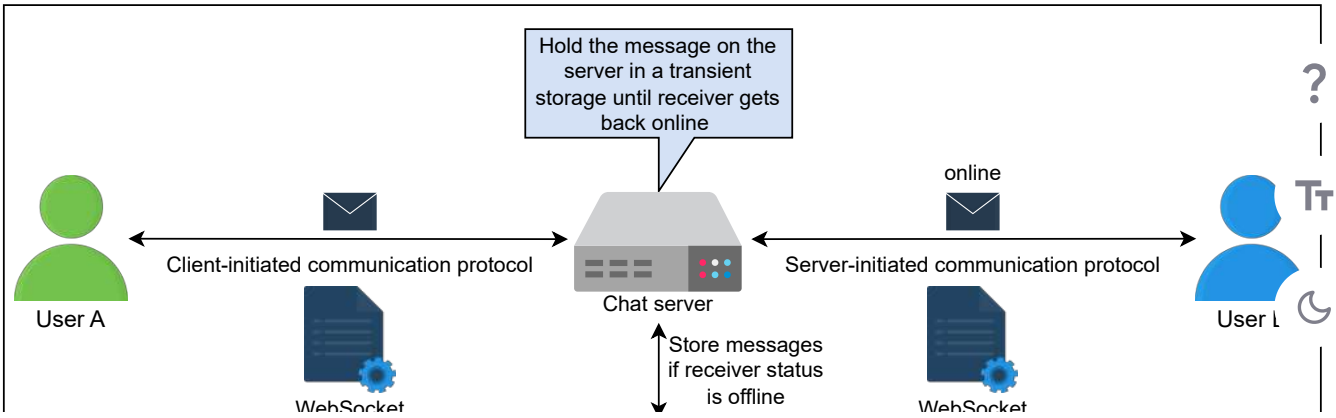
High-level Design of WhatsApp

Get introduced to the high-level design of the WhatsApp system.

We'll cover the following

High-level design

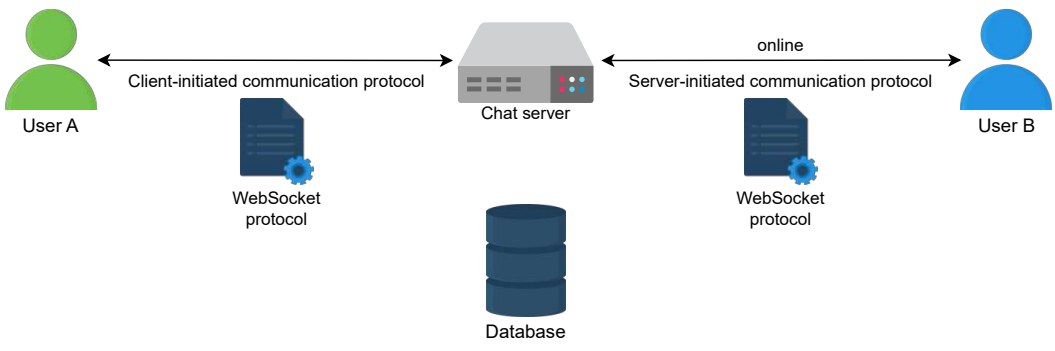
At an abstract level, the high-level design consists of a chat server responsible for communication between the sender and the receiver. When a user wants to send a message to another user, both connect to the chat server. Both users send their messages to the chat server. The chat server then sends the message to the other intended user and also stores the message in the database.



The following steps describe the communication between both clients:

1. User A and user B create a communication channel with the chat server.
2. User A sends a message to the chat server.
3. Upon receiving the message, the chat server acknowledges back to user A.
4. The chat server sends the message to user B and stores the message in the database if the receiver's status is offline.
5. User B sends an acknowledgment to the chat server.
6. The chat server notifies user A that the message has been successfully delivered.
7. When user B reads the message, the application notifies the chat server.
8. The chat server notifies user A that user B has read the message.

The process is shown in the following illustrations:



User A and user B create connections with the chat server

1 of

API design

WhatsApp provides a vast amount of features to its users via different APIs. Some features are mentioned below:

- Send message
- Get message or receive message
- Upload a media file or document
- Download document or media file
- Send a location
- Send a contact
- Create a status

However, we'll discuss essential APIs related to the first four features.

Send message

The `sendMessage` API is as follows:

```
sendMessage(sender_ID, reciever_ID, type, text=none, media_object=none, document=none)
```

This API is used to send a `text` message from a sender to a receiver by making a POST API call to the `/messages` API endpoint. Generally, the sender's and receiver's IDs are their phone numbers. The parameters used in this API call are described in the following table:

Parameter	Description
`sender_ID`	This is a unique identifier of the user who sends the message.
`reciever_ID`	This is a unique identifier of the user who receives the message.

`type`	The default message `type` is text. This represents whether the sender sends a media file or a document.
`text`	This feild contains the text that has to be sent as a message.
`media_object`	This parameter is defined based on the type parameter. It represents the media file to be sent.
`document`	This represents the document file to be sent.

Get message

The `getMessage` API is as follows:

```
getMessage(user_Id)
```

Using this API call, users can fetch all unread messages when they come online after being offline for some time.

Parameter	Description
`user_Id`	This is a unique identifier representing the user who has to fetch all unread messages.

Upload media or document file

The `uploadFile` API is as follows:

```
uploadFile(file_type, file)
```

We can upload media files via the `uploadFile` API by making a POST request to the `/v1/media` API endpoint. A successful response returns an ID that's

forwarded to the receiver. The maximum file size for media that can be uploaded is 16 MB, while the limit is 100 MB for a document.

Parameter	Description
`file_type`	This represents the type of file uploaded via the API call.
`file`	This contains the file being uploaded via the API call.

Download a document or media file

The `downloadFile` API is as follows:

```
downloadFile(user_id, file_id)
```

The parameters of this API call are explained in the following table:

Parameter	Description
`user_id`	This is a unique identifier of the user who will download a file.
`file_id`	This is a unique identifier of a file. It's generated while uploading a file via `uploadFile()` API call. The `downloadFile()` API call downloads the media file through this identifier. The client can find the `file_id` by providing the file name to the server. That API call is not shown here.

In the next lesson, we'll focus on the detailed design of the WhatsApp system.

Back

Requirements of WhatsApp's Design

Next

Detailed Design of WhatsApp

Mark as Completed

Detailed Design of WhatsApp

Learn about the design of the WhatsApp system in detail, and understand the interaction of various microservices.

We'll cover the following

Detailed design

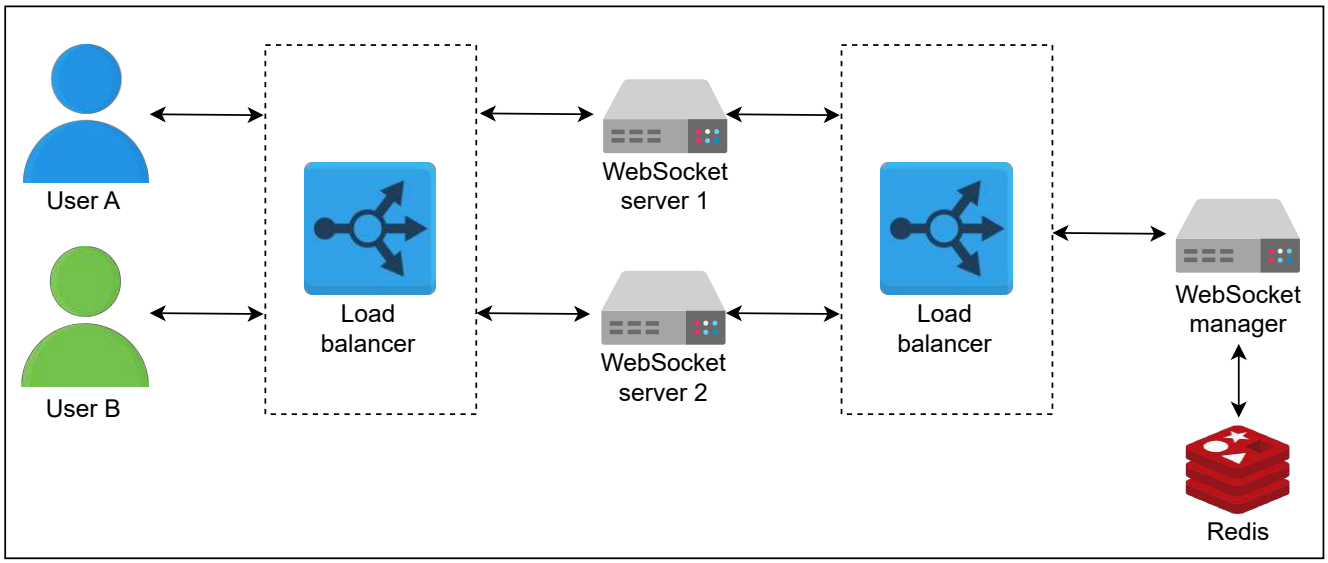
The high-level design discussed in the previous lesson doesn't answer the following questions:

- How is a communication channel created between clients and servers?
- How can the high-level design be scaled to support billions of users?
- How is the user's data stored?
- How is the receiver identified to whom the message is delivered?

explore each component in detail. Let's start with how users make connections with the chat servers.

Connection with a WebSocket server

In WhatsApp, each active device is connected with a **WebSocket server** via WebSocket protocol. A WebSocket server keeps the connection open with all the active (online) users. Since one server isn't enough to handle billions of devices, there should be enough servers to handle billions of users. The responsibility of each of these servers is to provide a port to every online user. The mapping between servers, ports, and users is stored in the WebSocket manager that resides on top of a cluster of the data store. In this case, that's Redis.



Two users are connected via WebSocket handlers

Point to Ponder

Question

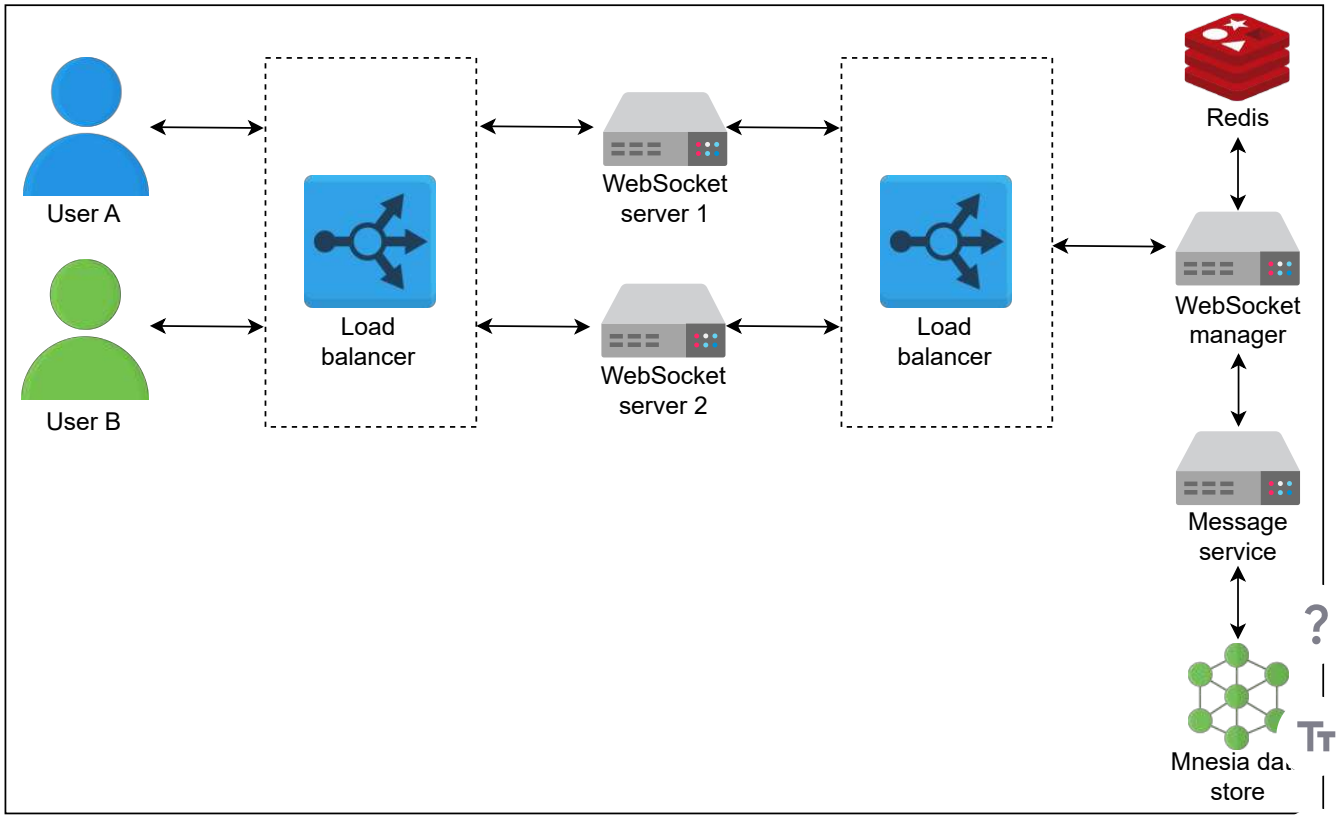
Why is WebSocket preferred over HTTP(S) protocol for client-server communication?

Show Answer

Send or receive messages

The WebSocket manager is responsible for maintaining a mapping between an active user and a port assigned to the user. Whenever a user is connected to another WebSocket server, this information will be updated in the data store.

A WebSocket server also communicates with another service called message service. **Message service** is a repository of messages on top of the **Mnesia database cluster**↴. It acts as an interface to the Mnesia database for other services interacting with the databases. It is responsible for storing and retrieving messages from the Mnesia database. It also deletes messages from the Mnesia database after a configurable amount of time. And, it exposes APIs to receive messages by various filters, such as user ID, message ID, and so on.



WebSocket communicates with message service on top of Mnesia database cluster

Now, let's assume that user A wants to send a message to user B. As shown in the above figure, both users are connected to different WebSocket servers. The system performs the following steps to send messages from user A to user B:

1. User A communicates with the corresponding WebSocket server to which it is connected.
2. The WebSocket server associated with user A identifies the WebSocket to which user B is connected via the WebSocket manager. If user B is online, the WebSocket manager responds back to user A's WebSocket server that user B is connected with its WebSocket server.
3. Simultaneously, the WebSocket server sends the message to the message service and is stored in the Mnesia database where it gets processed in the first-in-first-out order. As soon as these messages are delivered to the receiver, they are deleted from the database.
4. Now, user A's WebSocket server has the information that user B is connected with its own WebSocket server. The communication between user A and user B gets started via their WebSocket servers.
5. If user B is offline, messages are kept in the Mnesia database. Whenever they become online, all the messages intended for user B are delivered via push notification. Otherwise, these messages are deleted permanently after 30 days.

Both users (sender and receiver) communicate with the WebSocket manager to find each other's WebSocket server. Consider a case where there can be a continuous conversation between both users. This way, many calls are ma to the WebSocket manager. To minimize the latency and reduce the number of these calls to the WebSocket manager, each WebSocket server caches the following information:

- If both users are connected to the same server, the call to the WebSocket manager is avoided.
- It caches information of recent conversations about which user is connected to which WebSocket server.

Point to Ponder

Question

The data in the cache will become outdated if a user gets disconnected and connects with another server. Keeping this in mind, how long should a WebSocket server cache information?

Show Answer

Send or receive media files

So far, we've discussed the communication of text messages. But what happens when a user sends media files? Usually, the WebSocket servers are lightweight and don't support heavy logic such as handling the sending and receiving of media files. We have another service called the **asset service**, which is responsible for sending and receiving media files.

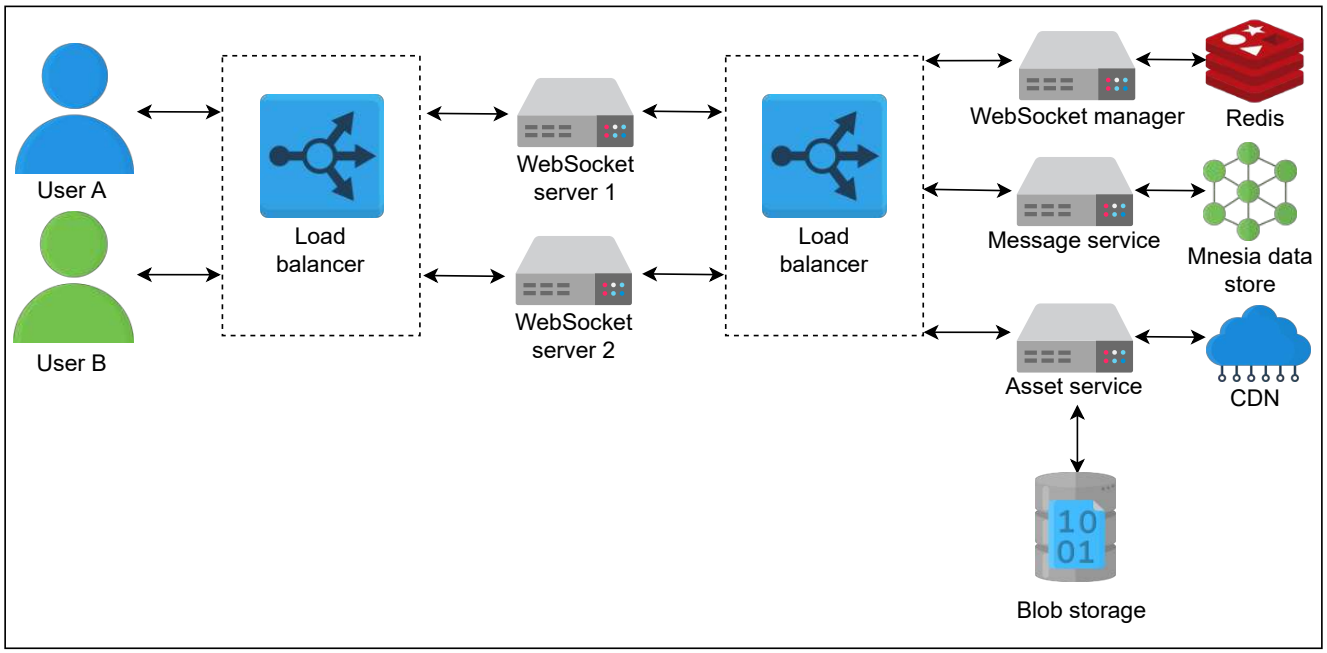
Moreover, the sending of media files consists of the following steps:

1. The media file is compressed and encrypted on the device side.
2. The compressed and encrypted file is sent to the asset service to store the file on blob storage. The asset service assigns an ID that's

communicated with the sender. The asset service also maintains a hash for each file to avoid duplication of content on the blob storage. For example, if a user wants to upload an image that's already there in the blob storage, the image won't be uploaded. Instead, the same ID is forwarded to the receiver.

3. The asset service sends the ID of media files to the receiver via the message service. The receiver downloads the media file from the blob storage using the ID.
4. The content is loaded onto a CDN if the asset service receives a large number of requests for some particular content.

The following figure demonstrates the components involved in sharing media files over WhatsApp messenger:



Sending media files via the asset service

Support for group messages

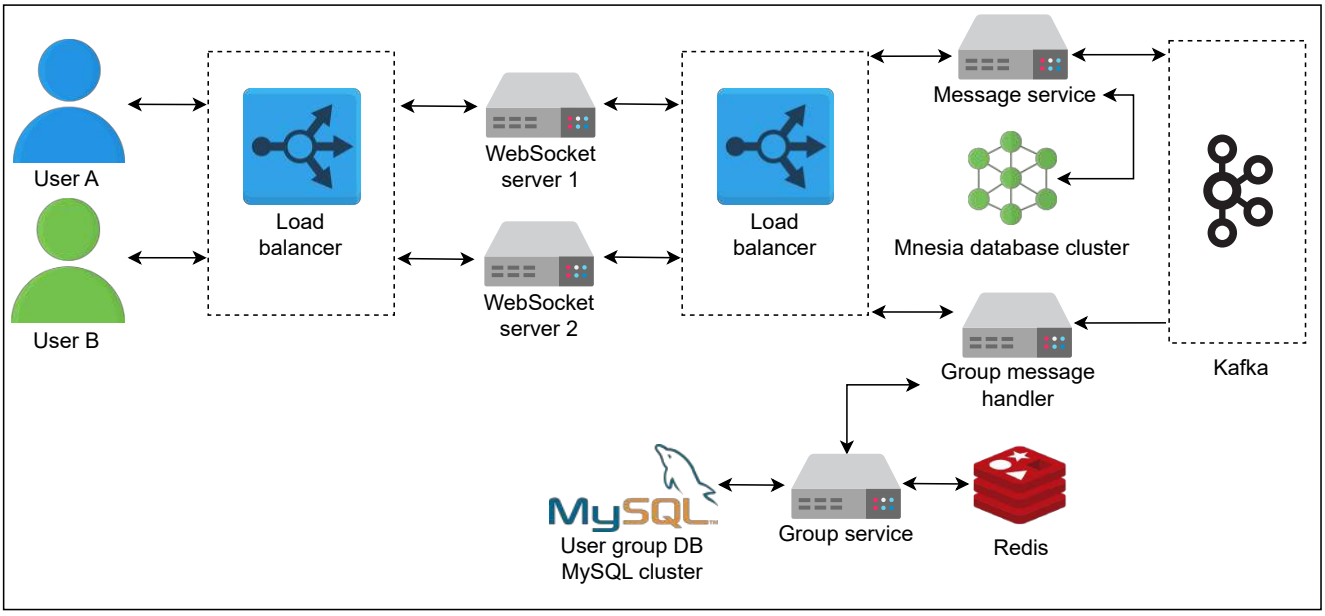
WebSocket servers don't keep track of groups because they only track activ users. However, some users could be online and others could be offline in a

group. For group messages, the following three main components are responsible for delivering messages to each user in a group:

- Group message handler
- Group message service
- Kafka

Let's assume that user A wants to send a message to a group with some unique ID—for example, `Group/A`. The following steps explain the flow of a message sent to a group:

1. Since user A is connected to a WebSocket server, it sends a message to the message service intended for `Group/A`.
2. The message service sends the message to Kafka with other specific information about the group. The message is saved there for further processing. In Kafka terminology, a group can be a topic, and the senders and receivers can be producers and consumers, respectively.
3. Now, here comes the responsibility of the group service. The **group service** keeps all information about users in each group in the system. It has all the information about each group, including user IDs, group ID, status, group icon, number of users, and so on. This service resides on top of the MySQL database cluster, with multiple secondary replicas distributed geographically. A Redis cache server also exists to cache data from the MySQL servers. Both geographically distributed replicas and Redis cache aid in reducing latency.
4. The group message handler communicates with the group service to retrieve data of `Group/A` users.
5. In the last step, the group message handler follows the same process as a WebSocket server and delivers the message to each user.

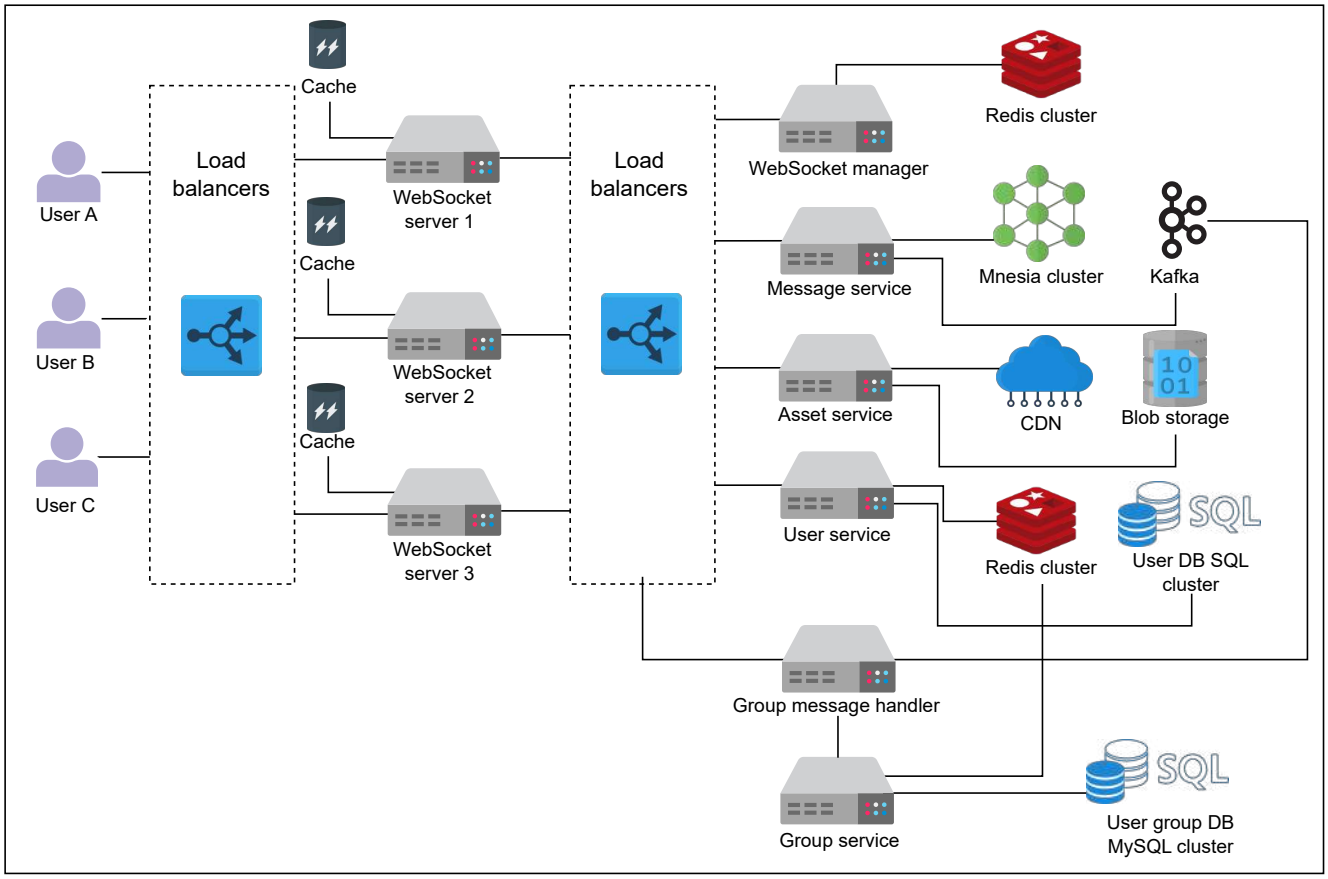


Components responsible for sending group messages

Optional: How encryption and decryption work in WhatsApp

Put everything together

We discussed the features of our WhatsApp system design. It includes user connection with a server, sending messages and media files, group messages, and end-to-end encryption, individually. The final design of our WhatsApp messenger is as follows:


Cache
Load balancers
User A
User B
User C
WebSocket server 1
Cache
WebSocket server 2
Cache
WebSocket server 3
Load balancers
WebSocket manager
Redis cluster
Message service
Mnesia cluster
Kafka
Asset service
CDN
Blob storage
User service
Redis cluster
User DB SQL cluster
Group message handler
Group service
User group DB MySQL cluster

Requirements of the Typeahead Suggestion System's Design

Learn about the requirements and resource estimations for the design of the typeahead suggestion system.

We'll cover the following

Requirements

In this lesson, we look into the requirements and estimated resources that are necessary for the design of the typeahead suggestion system. Our proposed design should meet the following requirements.

Functional requirements

The system should suggest top N (let's say top ten) frequent and relevant terms to the user based on the text a user types in the search box.

Non-functional requirements

- **Low latency:** The system should show all the suggested queries in real time after a user types. The latency shouldn't exceed 200 ms. A study suggests that the average time between two keystrokes is 160 milliseconds. So, our time-budget of suggestions should be greater than 160 ms to give a real-time response. This is because if a user is typing fast, they already know what to search and might not need suggestions. At the same time, our system response should be greater than 160 ms. However, it should not be too high because in that case, a suggestion might be stale and less useful.
- **Fault tolerance:** The system should be reliable enough to provide suggestions despite the failure of one or more of its components.
- **Scalability:** The system should support the ever-increasing number of users over time.

Resource estimation

As was stated earlier, the typeahead feature is used to enhance the user experience while typing a query. We need to design a system that works on a scale that's similar to Google Search. Google receives more than 3.5 billion searches every day. Designing such an enormous system is a challenging task that requires different resources. Let's estimate the storage and bandwidth requirements for the proposed system.

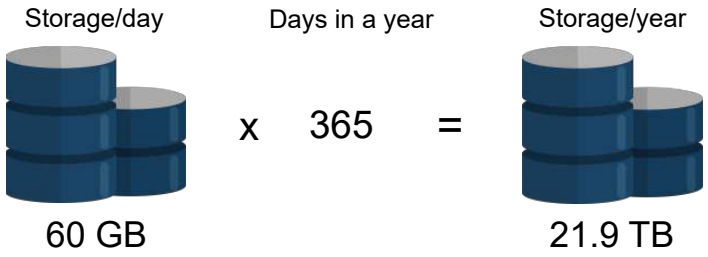
Storage estimation

Assuming that out of the 3.5 billion queries per day, two billion queries are unique and need to be stored. Let's also assume that each query consists of 15 characters on average, and each character takes 2 Bytes of storage. According to this formulation, we would require the following:

$2\,billion \times 15 \times 2 = 60GB$ to store all the queries made in a day.

Storage required per year:

$60GB/day \times 365 = 21.9TB/year$



Storage requirements for the typeahead suggestion system

Bandwidth estimation

3.5 billion queries will reach our system every day. Assume that each query a user types is 15 characters long on average.

Keeping this in mind, the total number of reading requests of characters per day would be as follows:

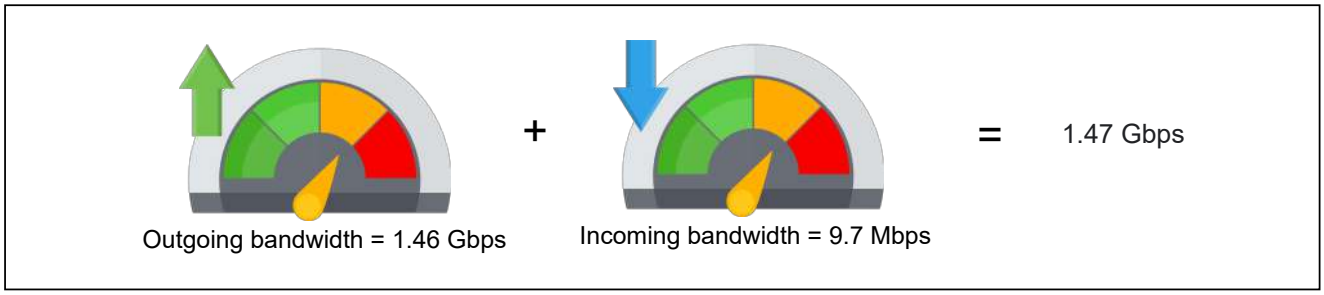
$15 \times 3.5\ billion = 52.5\ billion$ characters per day.

Total read requests per second: $52.5B/86400 \approx 0.607M$ characters/sec. 86,400 is the number of seconds per day.

Since each character takes 2 Bytes, the bandwidth our system would need is as follows:

$0.607M \times 2 \times 8 = 9.7Mb/sec$

$9.7Mb/sec$ is the incoming bandwidth requirement for queries that have a maximum length of 15 characters. Our system would suggest the top ten queries that are roughly of the same length as the query length after each character a user types. Therefore, the outgoing bandwidth requirement would become the following: $15 \times 10 \times 9.7Mb/sec = 1.46Gb/sec$.



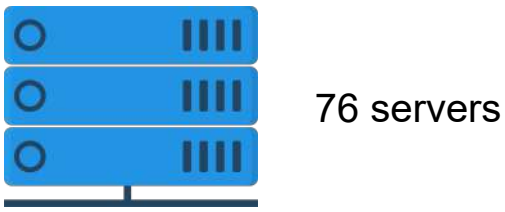
The total bandwidth required by the typeahead suggestion system

Number of servers estimation

Our system will receive 607,000 requests per second concurrently. Therefore, we need to have many servers installed to avoid burdening a single server. Let's assume that a single server can handle 8,000 queries per second. So, we require around 76 servers to handle 607,000 queries.

$607K/8000 \approx 76$ servers

Here, we only refer to the number of application servers. For simplicity, we ignored the number of cache and database servers. The actual number is higher than 76 because we would also need redundant servers to achieve availability.



The number of servers required for the typeahead suggestion system

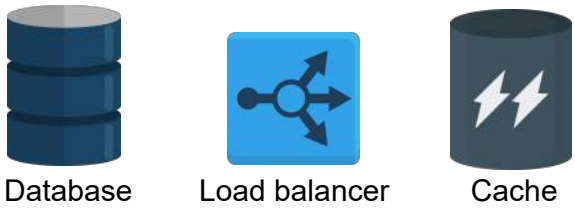
In the table below, adjust the values to see how the resource estimations change.

Resource Estimation for the Typeahead Suggestion System

Total Queries per Day	3.5	billion
Unique Queries per Day	2	billion
Minimum Characters in a Query	15	Characters
Server's QPS	8000	Queries per second
Storage	f 60	GB/day
Incoming Bandwidth	f 9.7	Mb/sec
Outgoing Bandwidth	f 1.455	Gb/sec
Number of Servers	f 76	Severs

Building blocks we will use

The design of the typeahead suggestion system consists of the following building blocks that have been discussed in the initial chapters of the course:



Building blocks required in the design of the typeahead suggestion system

- **Databases** are required to keep the data related to the queries' prefix
- **Load balancers** are required to disseminate incoming queries among a number of active servers.
- **Caches** are used to keep the top N suggestions for fast retrieval.

In the next lesson, we'll focus on the high-level design and APIs of the typeahead suggestion system.

Back

System Design: The Typeahead Suggestion ...

Next

High-level Design of the Typeahead Suggest...

Completed

High-level Design of the Typeahead Suggestion System

Get an overview of the high-level design of the typeahead suggestion system.

We'll cover the following

High-level design

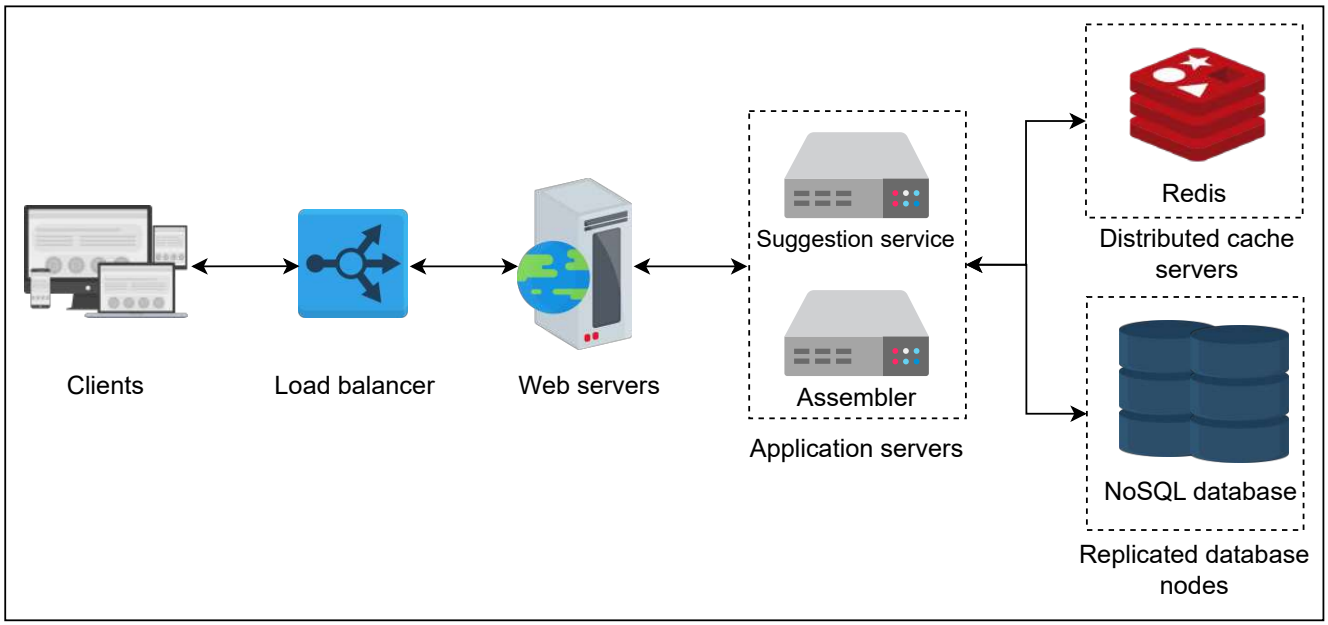
According to our requirements, the system shouldn't just suggest queries in real time with minimum latency but should also store the new search queries in the database. This way, the user gets suggestions based on popular and recent searches.

Our proposed system should do the following:

- Provide suggestions based on the search history of the user.
- Store all the new and trending queries in the database to include them in the list of suggestions.

When a user starts typing a query, every typed character hits one of the application servers. Let's assume that we have a **suggestions service** that obtains the top ten suggestions from the cache, Redis, and returns them as response to the client. In addition to this, suppose we have another service

known as an **assembler**. An assembler collects the user searches, applies some analytics to rank the searches, and stores them in a NoSQL database that's distributed across several nodes.



The high-level design of the typeahead suggestion system

Furthermore, we also need load balancers to distribute the incoming

web servers encapsulate the internal system architecture and provide other services, such as authentication, monitoring, request shaping, management, and more.

API design

Since the system provides suggestions to the user and adds trending queries to the databases, we need the following APIs.

Get suggestion

```
getSuggestions(prefix)
```

This API call retrieves suggestions from the servers. This call is made via the suggestion service and returns the response to the client.

The following table explains the parameter that's passed to the API call:

Parameter	Description
`prefix`	This refers to whatever the user has typed in the search bar.

Add trending queries to the database

```
addToDatabase(query)
```

This API call adds a trending `query` to the database via an assembler if the query has already been searched and has crossed a certain threshold.

Parameter	Description
`query`	This represents a frequently searched query that crosses the predefined limit.

Point to Ponder

Question

Instead of updating the whole page, we just need to update the suggested query in the search box in real time. What technique can

we use for this purpose?

Show Answer

In the next lesson, we'll learn about an efficient data structure that's used to store the suggestions or prefixes.

Back

Requirements of the Typeahead Suggestion ...

Next

Data Structure for Storing Prefixes

Completed

Detailed Design of the Typeahead Suggestion System

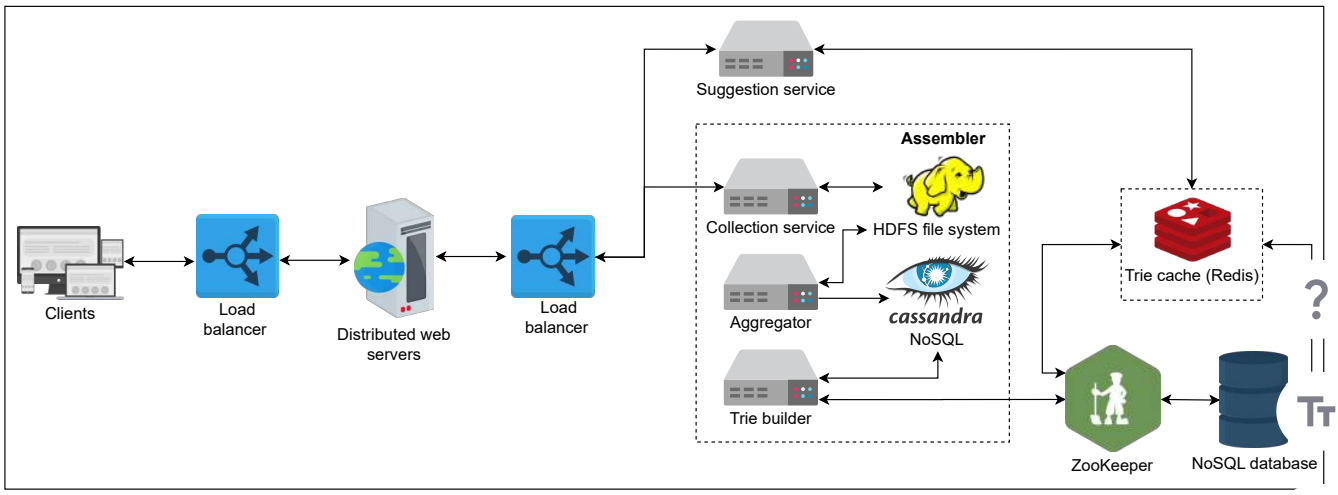
Learn about the detailed design of the typeahead suggestion system.

We'll cover the following

Detailed design

Let’s go over the flow and interaction of the components shown in the illustration below. Our design is divided into two main parts:

- A suggestion service
- An assembler



The detailed design of the typeahead suggestion system

Suggestion service

At the same time that a user types a query in the search box, the `getSuggestions(prefix)` API calls hit the suggestions services. The top ten popular queries are returned from the distributed cache, Redis.

Assembler

In the previous lesson, we discussed how tries are built, partitioned, and stored in the database. However, the creation and updation of a trie shouldn't come in the critical path of a user's query. We shouldn't update the trie in real time for the following reasons:

- There could be millions of users entering queries every second. During such phases with large amounts of incoming traffic, updating the trie in real time on every query can slow down our **suggestion service**.
- We have to provide top suggestions that might not frequently change after the creation or updation of the trie. So, it's less important to update the trie frequently.

In light of the reasons given above, we have a separate service called an assembler that's responsible for creating and updating tries after a certain configurable amount of time. The assembler consists of the following different services:

- **Collection service:** Whenever a user types, this service collects the log that consists of phrases, time, and other metadata and dumps it in a database that's processed later. Since the size of this data is huge, the **Hadoop Distributed File System (HDFS)** is considered a suitable storage system for storing this raw data.

  An example of the raw data from the collection service is shown in the following table. We record the time so that the system knows when to update the frequency of a certain phrase.

Raw Data Collected by the Collection Service

Phrases	Date and Time (DD-MM-YYYY HH:MM:SS)
UNIVERSITY	23-03-2022 10:20:11
UNIQUE	23-03-2022 10:21:10
UNIQUE	23-03-2022 10:22:24
UNIVERSITY	23-03-2022 10:25:09

- **Aggregator:** The raw data collected by the **collection service** is usually not in a consolidated shape. We need to consolidate the raw data to process it further and to create or update the tries. An aggregator retrieves the data from the HDFS and distributes it to different workers. Generally, the MapReducer is responsible for aggregating the frequency of the prefixes over a given interval of time, and the frequency is updated periodically in the associated Cassandra database. **Cassandra** is suitable for this purpose because it can store large amounts of data in a tabular format.

  The following table shows the processed and consolidated data within a particular period. This table is updated regularly by the aggregator and is stored in a hash table in a database like Cassandra. For simplicity, we assume that our data is case insensitive.

Useful Information Extracted from the Raw Data

Phrases	Frequency	Time Interv

UNIVERSAL	1400	1st 15 minut
UNIVERSITY	1340	1st 15 minut
UNIQUE	1200	1st 15 minut

- **Trie builder:** This service is responsible for creating or updating tries. It stores these new and updated tries on their respective shards in the trie database via ZooKeeper. Tries are stored in persistent storage in a file so that we can rebuild our trie easily if necessary. NoSQL document databases such as MongoDB are suitable for storing these tries. This storage of a trie is needed when a machine restarts.

  The trie is updated from the aggregated data in the Cassandra database. The existing snapshot of a trie is updated with all the new terms and their corresponding frequencies. Otherwise, a new trie is created using the data in the Cassandra database.

  Once a trie is created or updated, the system makes it available for the suggestion service.

Point to Ponder

Question

Should we collect data and build a trie per user, or should it be shared among all users?

Show Answer

Back

Data Structure for Storing Prefixes

Next

Evaluation of the Typeahead Suggestion Sys...

Completed

Requirements of Google Docs' Design

Learn about the requirements for designing a collaborative editing service.

We'll cover the following

Requirements

Let's look at the functional and non-functional requirements for designing a collaborative editing service.

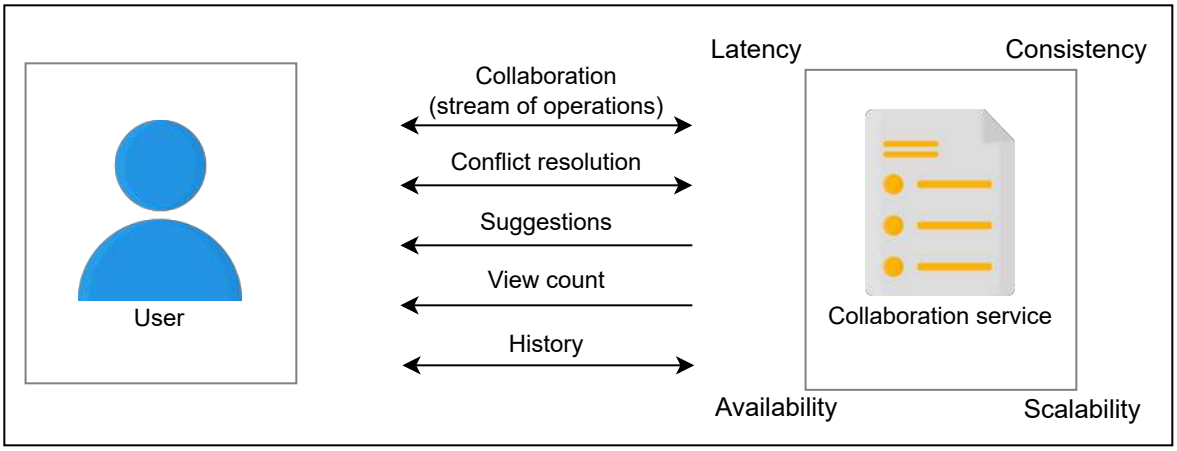
Functional requirements

The activities a user will be able to perform using our collaborative document editing service are listed below:

- **Document collaboration**: Multiple users should be able to edit a document simultaneously. Also, a large number of users should be able to view a document.

- **Conflict resolution**: The system should push the edits done by one user to all the other collaborators. The system should also resolve conflicts between users if they're editing the same portion of the document.
- **Suggestions**: The user should get suggestions about completing frequently used words, phrases, and keywords in a document, as well as suggestions about fixing grammatical mistakes.
- **View count**: Editors of the document should be able to see the view count of the document.
- **History**: The user should be able to see the history of collaboration on the document.

A real-world document editor also has to have functions like document creation, deletion, and managing user access. We focus on the core functionalities listed above, but we also discuss the possibility of other functionalities in the lessons ahead.



Functional and non-functional requirements of collaborative editing service

Non-functional Requirements

- **Latency**: Different users can be connected to collaborate on the same document. Maintaining low latency is challenging for users connected from different regions.
- **Consistency**: The system should be able to resolve conflicts between users editing the document concurrently, thereby enabling a consistent view of the document. At the same time, users in different regions

should see the updated state of the document. Maintaining consistency

robustness against failures.

- **Scalability**: A large number of users should be able to use the service at the same time. They can either view the same document or create new documents.

Resource estimation

Let's make some resource estimations based on the following assumptions:

- We assume that there are 80 million daily active users (DAU).
- The maximum number of users able to edit a document concurrently is 20.
- The size of a textual document is 100 KB.
- Thirty percent of all the documents contain images, whereas only 2% of documents contain videos.
- The collective storage required by images in a document is 800 KB, whereas each video is 3 MB.
- A user creates one document in one day.

Based on these assumptions, we'll make the following estimations.

Storage estimation

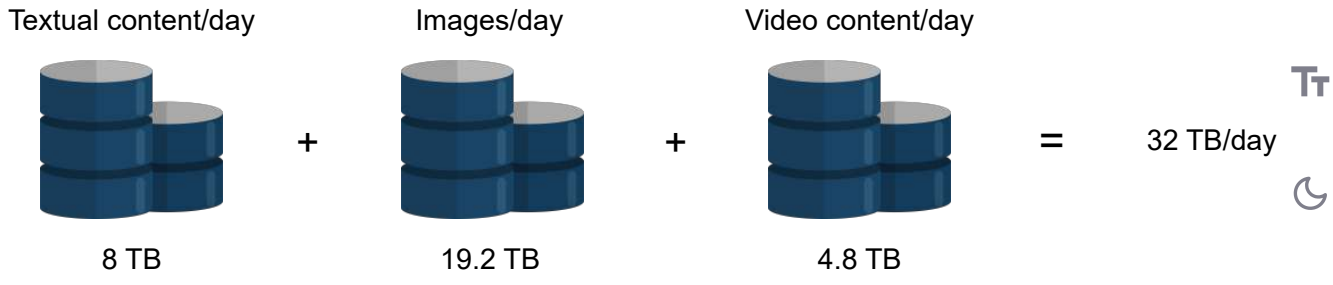
Considering that each user is able to create one document a day, there are a total of 80 million documents created each day. Below, we estimate the storage required for one day:

> **Note:** We can adjust the values in the table below to see how the storage requirement estimations change.

Estimation for Storage Requirements

Number of documents created by each user	1	per day
Number of active users	80	Million
Number of documents in a day	f 80	Million
Storage required for textual content per day	f 8	TB
Storage required for images per day	f 19.2	TB
Storage required for video content per day	f 4.8	TB
Total storage required per day	f 32	TB

See Detailed Calculations



Storage required by online collaborative document editing service per day

Total storage required for one day is as follows: $8 + 19.2 + 4.8 = 32\ TBs$ per day

> **Note:** Although our functional requirements state that we should keep a history of documents, we didn't include storage requirements for historical data for the sake of brevity.

Bandwidth estimation

Incoming traffic: Assuming that 32 TB of data are uploaded per day to the network of a collaborative editing service, the network requirement for incoming traffic will be the following:

$\frac{32\ TB}{86400} \times 8 = 3Gbps$ approximately

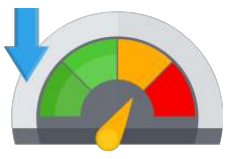
Outgoing traffic: To estimate outgoing traffic bandwidth, we'll assume the number of documents viewed by a user each day. Let's consider that a typical user views five documents per day. Then, the following calculations apply:

> **Note:** We can adjust the values in the table below to see how the calculations change.

Number of documents viewed by users	5	per day
Number of active user	80	Million

s		
Number of documents viewed in a day	f 400	Million
Number of documents viewed in a second	f 4630	per second
Bandwidth required for textual content per second	f 3.704	Gigabits per second(Gbps)
Bandwidth for image-based content per second	f 8.89	Gbps
Bandwidth for video content per second	f 2.22	Gbps
Total outgoing bandwidth required	f 14.81	Gbps

See Detailed Calculations



Incoming traffic bandwidth = 3 Gbps

+

Outgoing traffic bandwidth = 14.81 Gbps

≈ 18 Gbps

Summarizing the bandwidth requirement

> **Note:** The total bandwidth required is equal to the sum of incoming and outgoing traffic. $= 3 + 14.7 \approx 18 Gbps$ approximately.

Number of servers estimation

Let's assume that one user is able to generate 100 requests per day. Keeping in mind the number of daily active users, the number of requests per second (RPS) will be the following:

$$100 \times 80M = \frac{8000M}{86400} = 92.6 \; thousands/sec.$$

Number of RPS

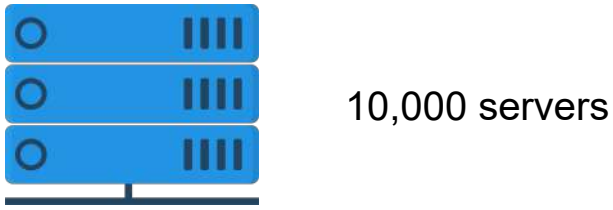
Requests by a user	100	per day
Number of DAU	80	Million
RPS	f 92.6	Thousands per second

We discussed in the Back of the Envelope lesson that RPS isn't sufficient information to calculate the number of servers. We'll use the following approximation for server count.

To estimate the number of servers required to fulfill the requests of 80 million users, we simply divide the number of users by the number of requests a server can handle. In the "Back of the Envelope" lesson, we discussed that our reference server can handle 8,000 requests per second. So, we see the following:

$$\frac{Number\ of\ daily\ active\ users}{Queries\ handled\ per\ second} = \frac{80M}{8000} = 10,000$$

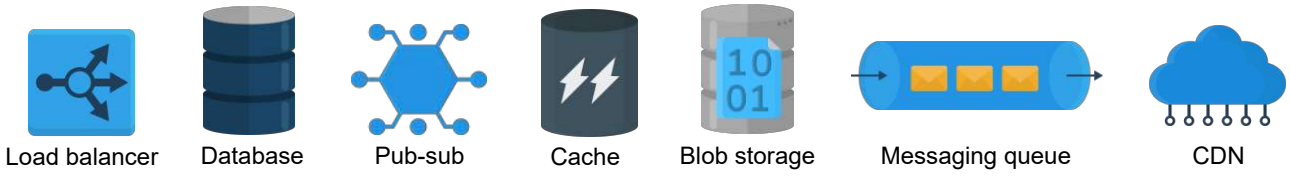
Informally, the equation above assumes that one server can handle 8,000 users.



Number of servers required

Building blocks we will use

We'll use the following building blocks in designing the collaborative document editing service.



Building blocks required to be integrated in the design

- **Load balancers** will be the first point of contact for users.
- **Databases** will be needed to store several things including textual content, history of documents, user data, etc. For this purpose, we may need different types of databases.
- **Pub-sub** systems can complete tasks that can't be performed right awa We'll complete a number of tasks asynchronously in our design. Therefore, we'll use a pub-sub system.
- **Caching** will help us improve the performance of our design.
- **Blob storage** will store large files, such as images and videos.

- **A Queueing system** will queue editing operations requested by different users. Because many editing requests can't be performed simultaneously, we have to temporarily put them in a queue.
- **A CDN** can store frequently accessed media in a document. We can also put read-only documents that are frequently requested in a CDN.

Back

System Design: Google Docs

Next

Design of Google Docs

Completed

Design of Google Docs

Let's understand how we can design the collaborative document editing service using various components.

We'll cover the following

Design

We'll complete our design in two steps. In the first step, we'll explain different components and building blocks and the reason for their choice in our design. The second step will describe how we fulfill various functional requirements by depicting a workflow.

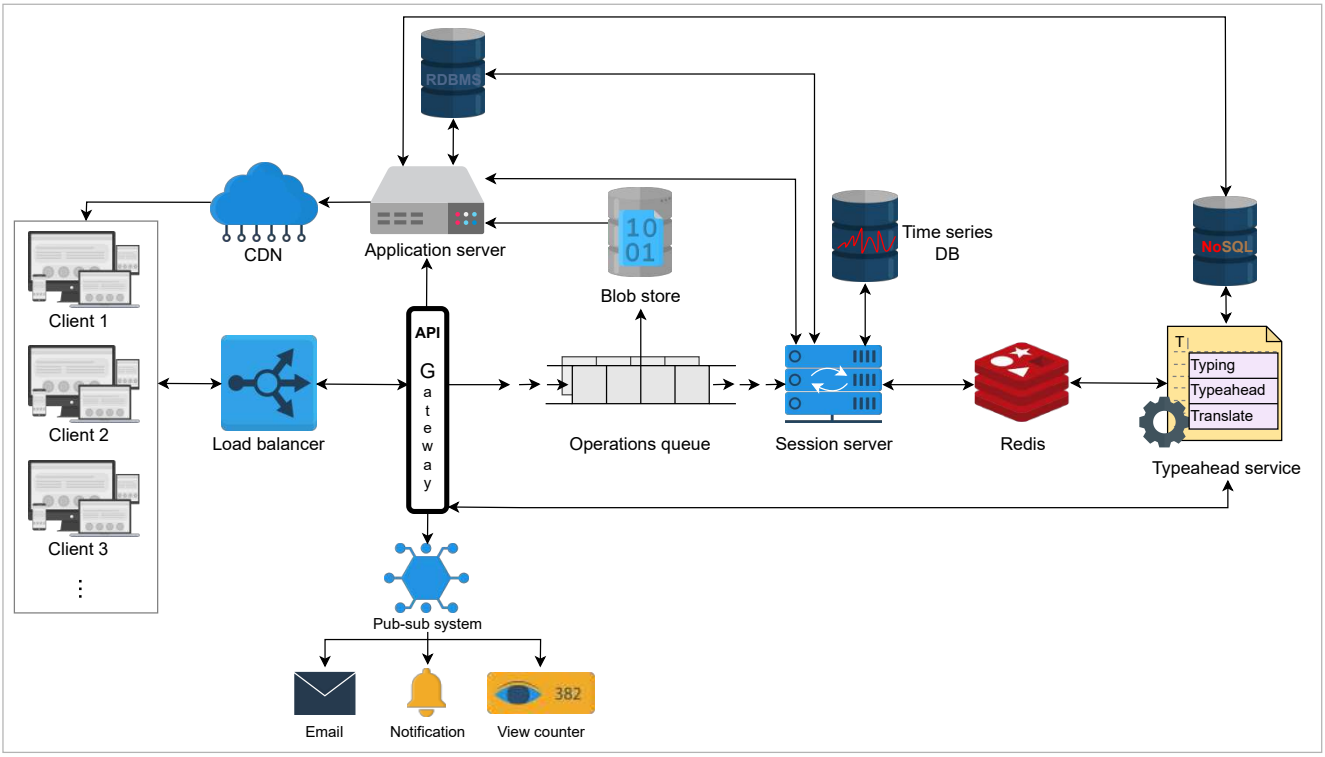
Components

We've utilized the following set of components to complete our design:

- **API gateway**: Different client requests will get intercepted through the API gateway. Depending on the request, it's possible to forward a single request to multiple components, reject a request, or reply instantly using an already cached response, all through the API gateway. Edit requests, comments on a document, notifications, authentication, and data storing requests will all go through the API gateway.

- **Application servers**: The application servers will run business logic and tasks that generally require computational power. For instance, some documents may be converted from one file type to another (for example, from a PDF to a Word document) or support features like import and export. It's also central to the attribute collection for the recommendation engine.
- **Data stores**: Various data stores will be used to fulfill our requirements. We'll employ a relational database for saving users' information and document-related information for imposing privilege restrictions. We can use NoSQL for storing user comments for quicker access. To save the edit history of documents, we can use a time series database. We'll use blob storage to store videos and images within a document. Finally, we can use distributed cache like Redis and a CDN to provide end users good performance. We use Redis specifically to store different data and frequently accessed documents. The CDN stores frequently accessed documents and heavy objects, like images and videos.
- **Processing queue**: Since document editing requires frequently sending small-sized data (usually characters) to the server, it's a good idea to queue this data for periodic batch processing. We'll add characters, images, videos, and comments to the processing queue. Using an HTTP call for sending every minor character is inefficient. Therefore, we'll use WebSockets to reduce overhead and observe live changes to the document by different users.
- **Other components**: Other components include the session servers that maintain the user's session information. We'll manage document access privileges through the session servers. Essentially, there will also be configuration, monitoring, pub-sub, and logging services that will handle tasks like monitoring and electing leaders in case of server failures, queueing tasks like user notifications, and logging debugging information.

The illustration below provides a depiction of how different components and building blocks coordinate to provide the service.



A detailed design of the collaborative document editing service

Quiz

Question 1

Why should we use WebSockets instead of HTTP methods? Why are WebSockets optimal for this kind of communication?

Show Answer

1 of 3

Workflow

In the following steps, we'll explain how different requests will get entertained after they reach the API gateway:

- **Collaborative editing and conflict resolution**: Each request gets forwarded to the operations queue. This is where conflicts get resolved between different collaborators of the same document. If there are no conflicts, the data is batched and stored in the time series database via session servers. Data like videos and images get compressed for storage optimization, while characters are processed right away.
- **History**: It's possible to recover different versions of the document with the help of a time series database. Different versions can be compared using DIFF operations that compare the versions and identify the differences to recover older versions of the same document.
- **Asynchronous operations**: Notifications, emails, view counts, and comments are asynchronous operations that can be queued through a pub-sub component like Kafka. The API gateway generates these requests and forwards them to the pub-sub module. Users sharing documents can generate notifications through this process.
- **Suggestions**: Suggestions are in the form of the typeahead service that offers autocomplete suggestions for typically used words and phrases. The typeahead service can also extract attributes and keywords from within the document and provide suggestions to the user. Since the number of words can be high, we'll use a NoSQL database for this purpose. In addition, most frequently used words and phrases will be stored in a caching system like Redis.
- **Import and export documents**: The application servers perform a number of important tasks, including importing and exporting documents. Application servers also convert documents from one format to another. For example, a `.doc` or `.docx` document can be

converted in to `.pdf` or vice versa. Application servers are also responsible for feature extraction for the typeahead service.

> **Note:** Our use of WebSockets speeds up the overall performance and enables us to facilitate chatting between users who are collaborating on the same document. If we combine WebSockets with a Redis-like cache, it's possible to develop an effective chatting feature.

Quiz

Question 1

We're implementing view counters through an asynchronous method, which means that the number of views of a document may be outdated. Could we use sharded counters or Redis counters to get effective results?

Show Answer

1 of 2

Back

Requirements of Google Docs' Design

Next

Concurrency in Collaborative Edi

Comple

Concurrency in Collaborative Editing

Let's explore different methods of conflict resolution in concurrent collaboration on a document.

We'll cover the following

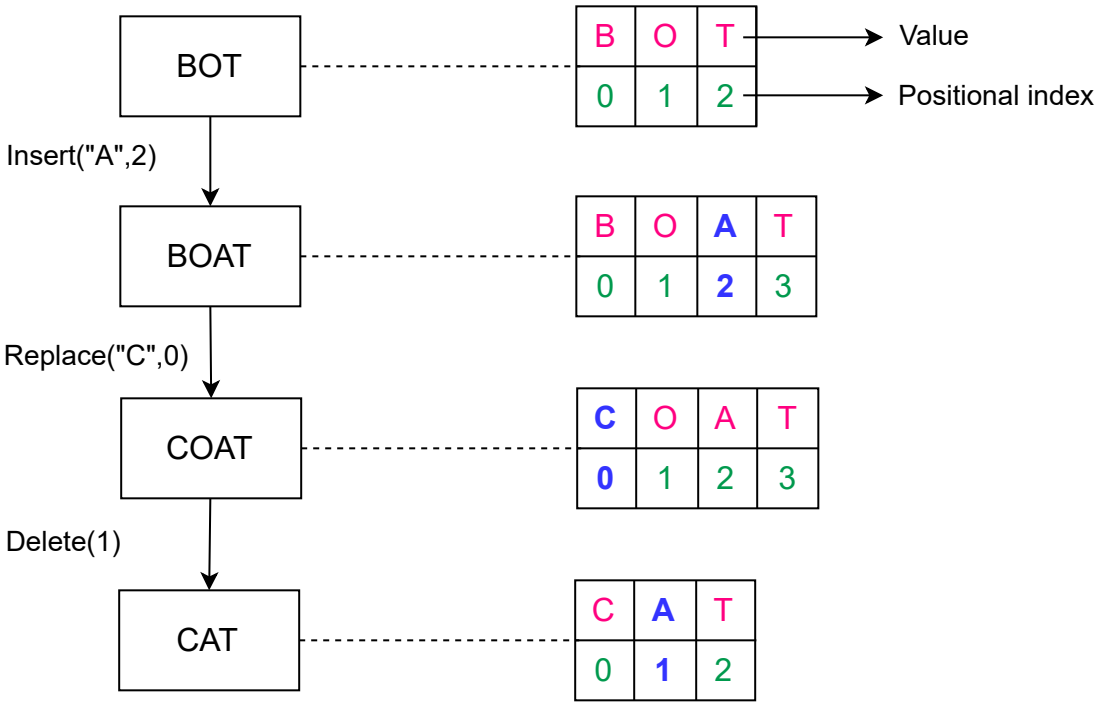
Introduction

We've discussed the design for a collaborative document editing service, but we haven't addressed how we'll deal with concurrent changes in the document by different users. However, before discussing concurrency issues, we need to understand the collaborative text editor.

What is a document editor?

A **document** is a composition of characters in a specific order. Each character has a value and a positional index. The character can be a letter, a number, an enter (↵), or a space. An index represents the character's position within the ordered list of characters.

The job of the text or document editor is to perform operations like `insert()`, `delete()`, `edit()`, and more on the characters within the document. A depiction of a document and how the editor will perform these operations is below.



How a document editor performs different operations

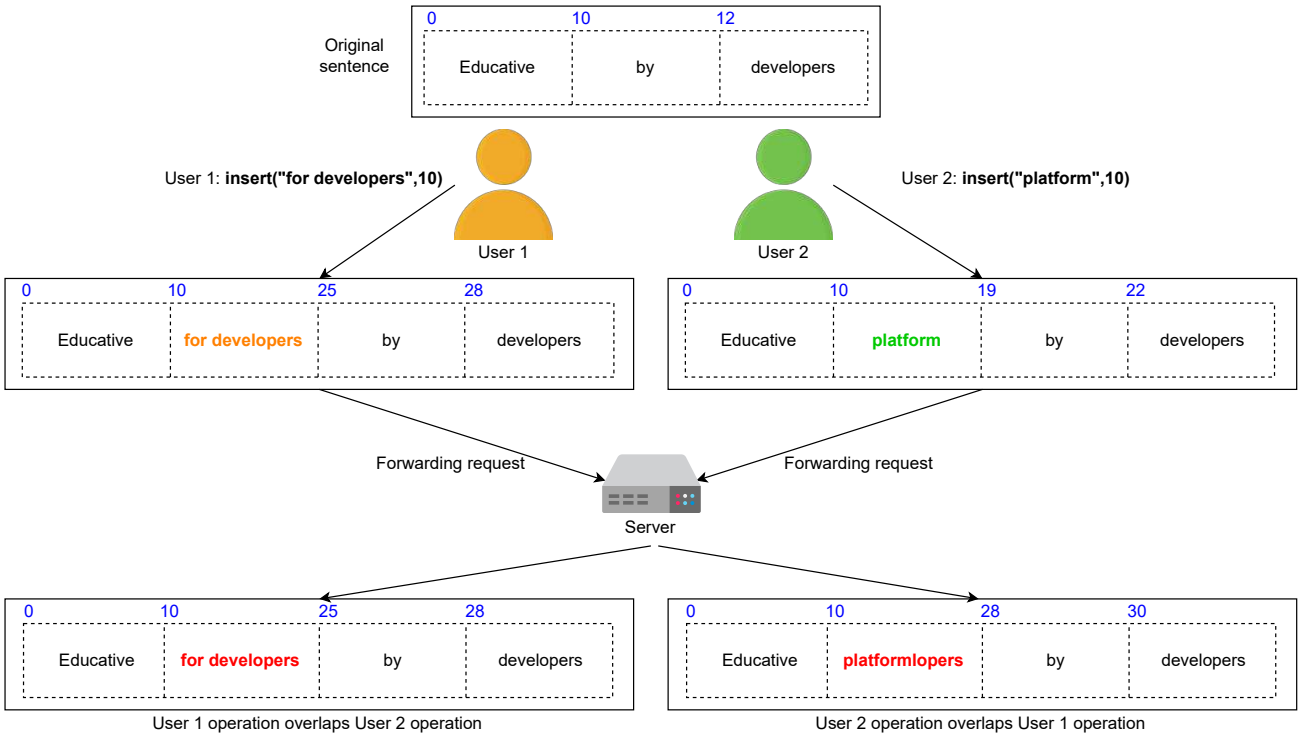
Concurrency issues

Collaboration on the same document by different users can lead to concurrency issues. Conflicts may arise whenever multiple users edit the same portion of a document. Since users have a local copy of the document, the final status of the document may be different at the server from what the users see at their end. After the server pushes the updated version, users discover the unexpected outcome.

Example 1

Let's consider a scenario where two users want to add some characters at the same positional index. Below, we've depicted how two users modifying

the same sentence can lead to conflicting results:



How two users modifying the same text can lead to concurrency issues

As shown above, two users are trying to modify the same sentence, “Educative by developers.” Both users are performing `insert()` at index `10`. The two possibilities are as follows:

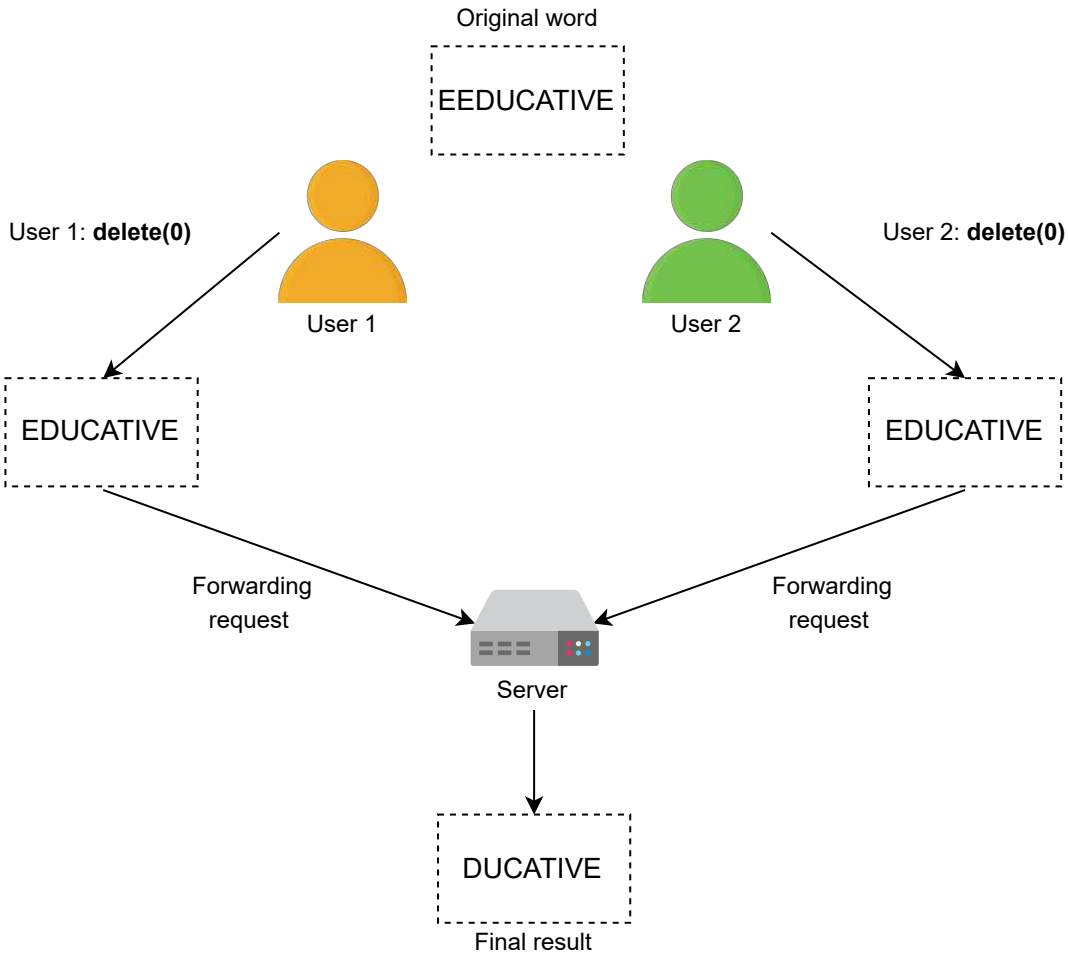
- The phrase “for developers” overwrites “platform.” This leads to an outcome of “for developers.”
- The phrase “platform” overlaps “for developers.” This leads to an outcome of “platformlopers.”

This example shows that operations applied in a different order don’t hold the commutative property.

Example 2

Let’s look at another simple example where two users are trying to delete the same character from a word. Let’s use the word “EEDUCATIVE.” Because

the word has an extra “E,” both the users would want to remove the extra character. Below, we see how an unexpected result can occur:



How deleting the same character can lead to unexpected changes

This second example shows that different users applying the same operation won’t be idempotent. So, conflict resolution is necessary where multiple collaborators are editing the same portion of the document at the same time.

From the examples above, we understand that a solution to concurrency issues in collaborative editing should respect two rules:

- **Commutativity**: The order of applied operations shouldn’t affect the end result.

- **Idempotency**: Similar operations that have been repeated should apply only once.

Below, we identify two well-known techniques for conflict resolution.

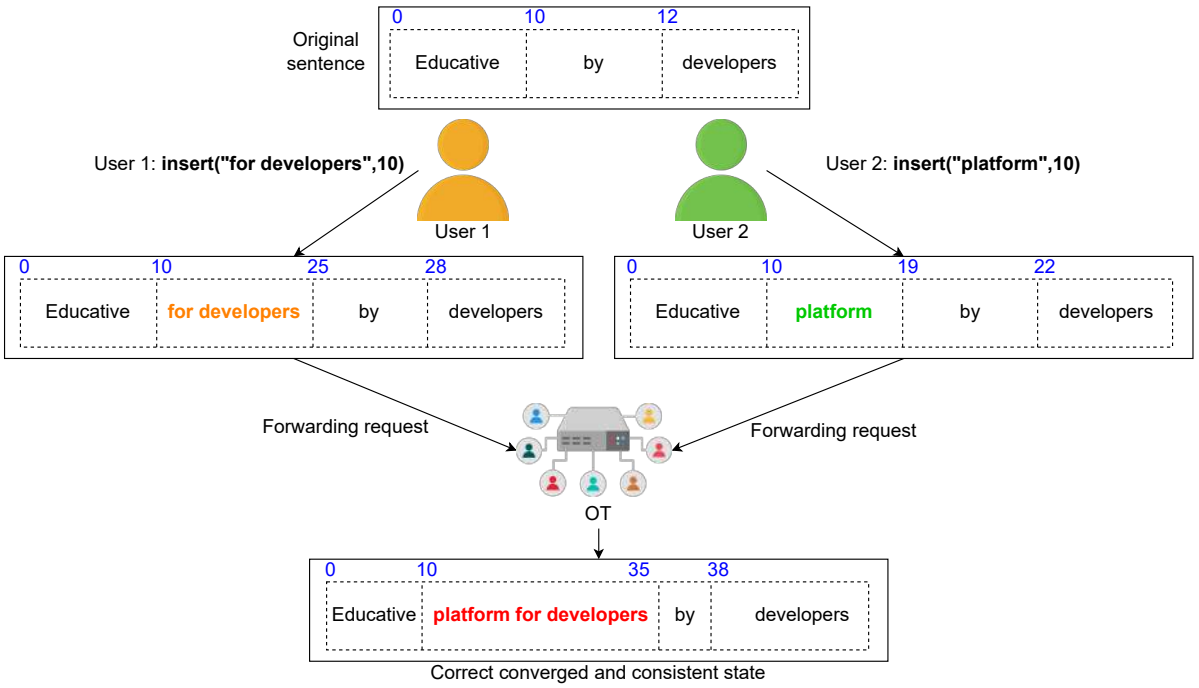
Techniques for conflict resolution

Let's discuss two leading technologies that are used for conflict resolution in collaborative editing.

Operational transformation

Operational transformation (OT) is a technique that's widely used for conflict resolution in collaborative editing. OT emerged in 1989 and has improved throughout the years. It's a lock-free and non-blocking approach for conflict resolution. If operations between collaborators conflict, OT resolves conflicts and pushes the correct converged state to end users. As a result, OT provides consistency for users.

OT performs operations using the positional index method to resolve conflicts, such as the ones we discussed above. OT resolves the problems above by holding commutativity and idempotency.



Ensuring commutativity

1 of 2

Collaborative editors based on OT are consistent if they have the following two properties:

- **Causality preservation**: If operation `a` happened before operation `b`, then operation `a` is executed before operation `b`.
- **Convergence**: All document replicas at different clients will eventually be identical.

The properties above are a part of the **CC consistency model**, which is a model for consistency maintenance in collaborative editing.

Consistency Models in OT

OT has two disadvantages:

- Each operation to characters may require changes to the positional index. This means that operations are **order dependent**↴ on each other.
- It's challenging to develop and implement.

Operational transformation is a set of complex algorithms, and its correct implementation has proved challenging for real-world applications. For example, the Google Wave team took two years to implement an OT algorithm.

Conflict-free Replicated Data Type (CRDT)

The **Conflict-free Replicated Data Type (CRDT)** was developed in an effort to improve OT. A CRDT has a complex data structure but a simplified algorithm.

A CRDT satisfies both commutativity and idempotency by assigning two key properties to each character:

- It assigns a globally unique identity to each character.
- It globally orders each character.

Each operation now has an updated data structure:

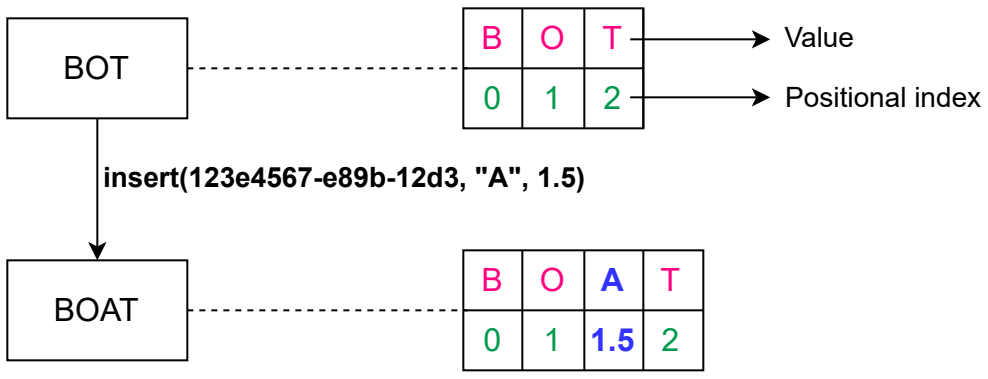
Data	Explanation	Example
SiteiD	Unique Site identifier in the form of a UUID	123e4567-e89b-12d3
SiteCounter	The sequence of operations from a site	87
Value	Value of character	"A"
PositionalIndex	Unique position of each character	4.5

Simplified data structure of a CRDT

The `SiteID` uniquely identifies a user's site requesting an operation with a `Value` and a `PositionalIndex`. The value of `PositionalIndex` can be in fractions for two main reasons.

- The `PositionalIndex` of other characters won't change because of certain operations.
- The order dependency of operations between different users will be avoided.

The example below depicts that a user from site ID `123e4567-e89b-12d3` is inserting a character with a value of `A` at a `PositionalIndex` of `1.5`. Although a new character is added, the positional indexes of existing characters are preserved using fractional indices. Therefore, the order dependency between operations is avoided. As shown below, an `insert()` between `O` and `T` didn't affect the position of `T`.



Preventing order dependency between operations

CRDTs ensure strong consistency between users. Even if some users are offline, the local replicas at end users will converge when they come back online.

Although well-known online editing platforms like Google Docs, Etherpad, and Firepad use OT, CRDTs have made concurrency and consistency in collaborative document editing easy. In fact, with CRDTs, it's possible to implement a serverless peer-to-peer collaborative document editing service.

> **Note:** OT and CRDTs are good solutions for conflict resolution in collaborative editing, but our use of WebSockets makes it possible to highlight a collaborator's cursor. Other users will anticipate the position of a collaborator's next operation and naturally avoid conflict.

Quiz

Facebook, WhatsApp, Instagram, Oculus Outage

Learn the causes of a major Facebook outage and how to avoid them.

We'll cover the following

affecting its other affiliates, including Messenger, WhatsApp, Mapillary, Instagram, and Oculus. Popular media reported the impact of this failure prominently. The *New York Times* reported the following headline: “Gone in Minutes, Out for Hours: Outage Shakes Facebook.”

According to one estimate, this outage cost Facebook about $100 million in revenue losses and many billions due to the declining stock of the company.

Let’s look at the sequence of events that caused this global problem.

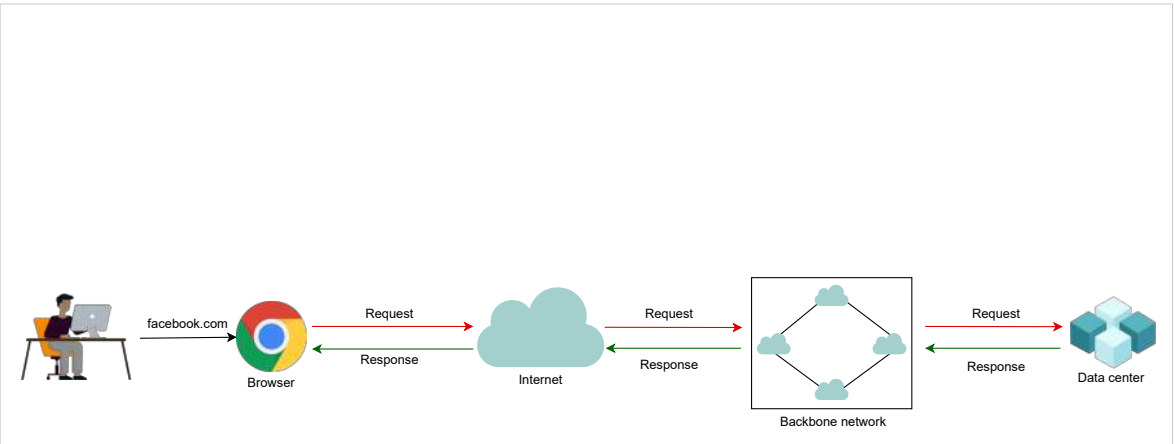
The sequence of events

The following sequence of events led to the outage of Facebook and its accompanied services:

- A routine maintenance system was needed to find out the spare capacity on Facebook’s backbone network.

- Due to a configuration error, the maintenance system disconnected all the data centers from each other on the backbone network. Earlier, an automated configuration review tool was used to look for any issues in the configuration, but tools like these aren't perfect. In this specific case, the review tool missed the problems present in a configuration.
- The authoritative domain name systems (DNSs) of Facebook had a health check rule that if it couldn't reach Facebook's internal data centers, it would stop replying to client DNS queries by withdrawing the routes.
- When the networks routes where Facebook's authoritative DNS was hosted were withdrawn, all cached mapping of human-readable names to IPs soon timed out at all public DNS resolvers. When a client resolves www.facebook.com, the DNS resolver first goes to one of the root DNS servers, which provides a list of authoritative DNS servers for `.com`. The resolver connects to one of them, and then they provide IPs for the authoritative DNS servers for Facebook. However, after route withdrawal, it was impossible to reach them.
- Then, no one was able to reach Facebook and its subsidiaries.

The slides below depict the events in pictorial form.



Facebook's high-level architecture

Analysis

Some of the key takeaways from the series of events shown above are the following:

- **From common activity to catastrophe**: The withdrawal or addition of the network routes is a relatively common activity. The confluence of bugs (first a faulty configuration, and then a bug in an audit tool not able to detect such a problem) triggered a chain of events, which resulted in cascading failures. **A cascading failure** is when one failure can trigger another failure, ultimately bringing the whole system down.
- **Reasons for slow restoration**: It might seem odd that it took six hours to restore the service. Wasn't it easy to reannounce the withdrawn routes? At the scale that Facebook is operating on, rarely is anything done manually, and there are automated systems to perform changes. The internal tools probably relied on the DNS infrastructure, and when all data centers are offline from the backbone, it would have been virtually impossible to use those tools. Manual intervention would have been necessary. Manually bootstrapping a system of this scale isn't easy. The usual physical and digital security mechanisms that were in place made it a slow process to intervene manually.
- **Low probability events can occur**: In retrospect, it might seem odd that authoritative DNS systems disconnect themselves if internal data centers aren't accessible. This is another example where a very rare event, such as none of the data centers being accessible, happened, triggering another event.
- **Pitfalls of automation**: Facebook has been an early advocate of automating network configuration changes, effectively saying that software can do a better job of running the network than humans, who

are more prone to errors. However, software can have bugs, such as this one.

Lessons learned

- **Ready operations team**: There can be a hidden single point of failure in complex systems. The best defense against such faults is to have the operations team ready for such an occurrence through regular training. Thinking clearly under high-stress situations becomes necessary to deal with such events.
- **Simple system design**: As systems get bigger, they become more complex, and they have emergent behaviors. To understand the overall behavior of the system, it might not be sufficient to understand only the behavior of its components. Cascading failures can arise. This is one reason to keep the system design as simple as possible for the current needs and evolve the design slowly. Unfortunately, there's no perfect solution to deal with this problem. We should accept the possibility of such failures, perform continuous monitoring, have the ability to solve issues when they arise, and learn from the failures to improve the system.
- **Contingency plan**: Some third-party services rely on Facebook for single sign-on. When the outage occurred, third-party services were up and running, but their clients were unable to use them because Facebook's login facility was also unavailable. This is another example of assuming that some service will always remain available and of a hidden single point of failure.
- **Hosting DNS to independent third-party providers**: There are a few services that are so robustly designed and perfected over time that the clients start assuming that the service is and will always be 100% available. The DNS is one such service, and it has been very carefully crafted. Designers often assume that it will never fail. Hosting DNS to independent third-party providers might be a way to guard against su

problems. DNS allows multiple authoritative servers, and an organization might have many at different places. Although, we should note that DNS at Facebook's scale isn't simple, is tightly connected to their backbone infrastructure, and changes frequently. Delegating such a piece to an independent third party is expensive, and it might reveal internal service details. So, there can be a trade-off between business and robustness needs.

- **Trade-offs**: There can be some surprising trade-offs. An example here is the need for data security and the need for rapid manual repair. Because so many physical and digital safeguards were in place, manual intervention was slow. This is a catch-22 situation—lowering security needs can cause immense trouble, and slow repair for such events can also make it hard for the companies. The hope is that the need for such repairs is a very rare event.
- **Surge in load**: The failure of large players disrupts the entire Internet. Third-party public resolvers, such as Google and Cloudflare, saw a surge in the load due to unsuccessful DNS retries.
- **Resuming the service**: Restarting a large service isn't as simple as flipping a switch. When the load suddenly becomes almost zero, turning them up suddenly may lead to a many megawatt uptick in power usage. This might even cause issues for the electric grid. Complex systems usually have a steady state, and if they go out of that steady state, care must be taken to bring them back.

Point to Ponder

Question

What can we do to safeguard against the kinds of faults experienced by Facebook?

Show Answer

Back

Introduction to Distributed System Failures

Next

AWS Kinesis Outage Affecting Many Organi...

Mark as Completed