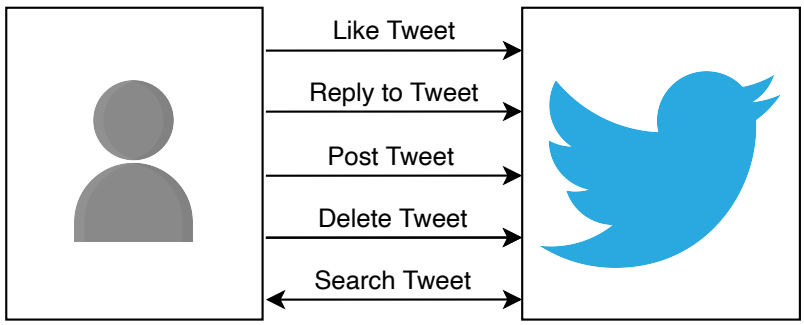
System Design: Twitter

Get an overview of Twitter and a brief description of what we'll learn in this chapter.

We'll cover the following

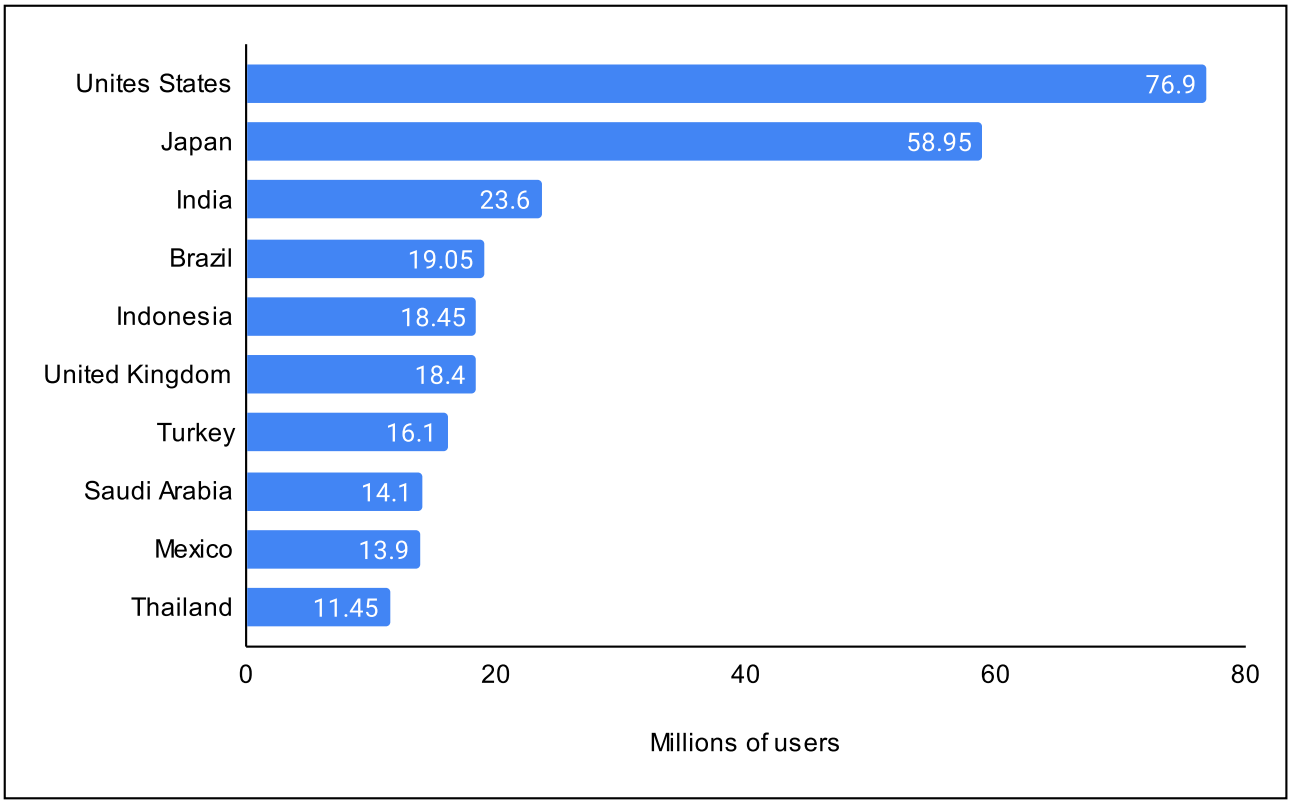
Twitter

Twitter is a free microblogging social network where registered users post messages called "Tweets". Users can also like, reply to, and retweet public tweets. Twitter has about 397 million users as of 2021 that is proliferating with time. One of the main reasons for its popularity is the vast sharing of the breaking news on the platform. Another important reason is that Twitter allows us to engage with and learn from people belonging to different communities and cultures.



A user performs various operation on Twitter

The illustration below shows the country-wise userbase of Twitter as of Januar 2022 (source: Statista), where the US is taking the lead. Such statistics are important when we design our infrastructure because it allows us to allocate more capacity to the users of a specific region, and traffic served from near specific users.



Country wise userbase on Twitter

How will we design Twitter?

We'll divide Twitter's design into four sections:

1. **Requirements**: This lesson describes the functional and non-functional requirements of Twitter. We'll also estimate multiple aspects of Twitter, such as storage, bandwidth, and computational resources.
2. **Design**: We'll discuss the high-level design of Twitter in this lesson. We also briefly explain the API design and identify the significant components of the Twitter architecture. Moreover, we will discuss how to manage the Top-k problem, such as Tweets liked or viewed by millions of users on Twitter.
3. **Client-side load balancers**: This lesson discusses how Twitter performs load balancing for its microservices system to manage billions of requests between various services' instances. Furthermore, we also see why Twitter uses a customized load-balancing technique instead of other commonly used approaches.
4. **Quiz**: Finally, we'll reinforce major concepts of Twitter design with a quiz.

Let's begin with defining Twitter's requirements.

Requirements of Twitter's Design

Understand the requirements and estimation for Twitter's design.

We'll cover the following

Requirements

Let's understand the functional and non-functional requirements below:

Functional requirements

The following are the functional requirements of Twitter:

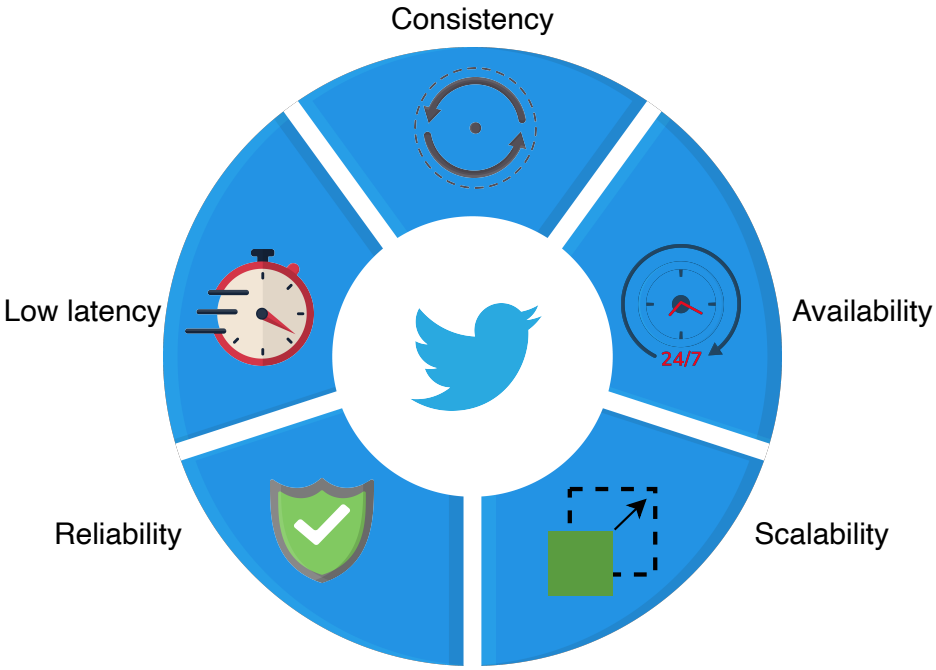
- **Post Tweets**: Registered users can post one or more Tweets on Twitter.
- **Delete Tweets**: Registered users can delete one or more of their Tweets on Twitter.
- **Like or dislike Tweets**: Registered users can like and dislike public and their own Tweets on Twitter.
- **Reply to Tweets**: Registered users can reply to the public Tweets on Twitter.
- **Search Tweets**: Registered users can search Tweets by typing keywords, hashtags, or usernames in the search bar on Twitter.
- **View user or home timeline**: Registered users can view the user's timeline, which contains their own Tweets. They can also view the home's timeline, which contains followers' Tweets on Twitter.
- **Follow or unfollow the account**: Registered users can follow or unfollow

other users on Twitter.

- **Retweet a Tweet**: Registered users can Retweet public Tweets of other users on Twitter.

Non-functional requirements

- **Availability**: Many people and organizations use Twitter to communicate time-sensitive information (service outage messages). Therefore, our service must be highly available and have a good uptime percentage.
- **Latency**: The latency to show the most recent top Tweets in the home timeline could be a little high. However, near real-time services, such as Tweet distribution to followers, must have low latency.
- **Scalability**: The workload on Twitter is read-heavy, where we have many readers and relatively few writers, which eventually requires the scalability of computational resources. Some estimates suggest a 1:1000 write-to-read ratio for Twitter. Although the Tweet is limited to 280 characters, and the video clip is limited to 140s by default, we need high storage capacity to store and deliver Tweets posted by public figures to their millions of followers.
- **Reliability**: All Tweets remain on Twitter. This means that Twitter never deletes its content. So there should be a promising strategy to prevent the loss or damage of the uploaded content.
- **Consistency**: There's a possibility that a user on the east coast of the US does not get an immediate status (like, reply, and so on.) update on the Tweet, which is liked or Retweeted by a user on the west coast of the US. However, the user on the west coast of the US needs an immediate status update on his like or reply. An effective technique is needed to offer rapid feedback to the user (who liked someone's post), then to other specified users in the same region, and finally to all worldwide users linked to the Tweet.

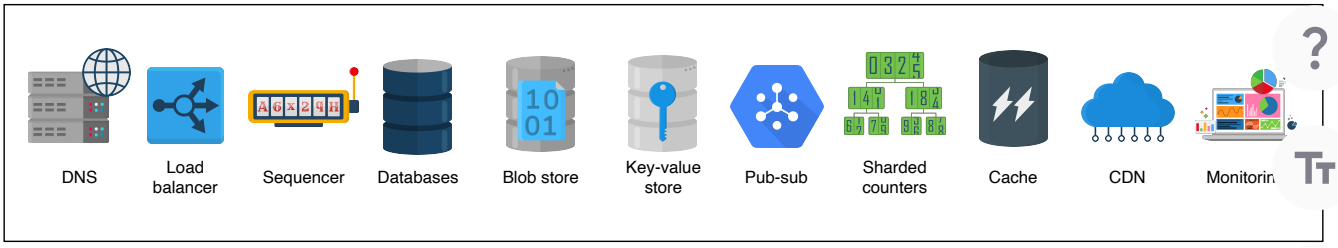


Non-functional requirements of Twitter

Furthermore, we must identify which essential resources must be estimated in the Twitter design. We used Twitter as an example in our Back-of-the-Envelope chapter, so we won't repeat that exercise here.

Building blocks we will use

Twitter's design utilizes the following building blocks that we discussed in the initial chapters.



Building blocks used in Twitter design

- **DNS** is the service that maps human-friendly Twitter domain names to machine-readable IP addresses.
- **Load balancers** distribute the read/write requests among the respective services.
- **Sequencers** generate the unique IDs for the Tweets.

- **Databases** store the metadata of Tweets and users.
- **Blob stores** store the images and video clips attached with the Tweets.
- **Key-value stores** are used for multiple purposes such as indexing, identifying the specified counter to update its value, identifying the Tweets of a particular user and many more.
- **Pub-sub** is used for real-time processing such as elimination of redundant data, organizing data, and much more.
- **Sharded counters** help to handle the count of multiple features such as viewing, liking, Retweeting, and so on,. of the accounts with millions of followers.
- A **cache** is used to store the most requested and recent data in RAM to give users a quick response.
- **CDN** helps end users to access the data with low latency.
- **Monitoring** analyses all outgoing and incoming traffic, identifies the redundancy in the storage system, figures out the failed node, and so on.

In the next lesson, we'll discuss the design of the Twitter system.

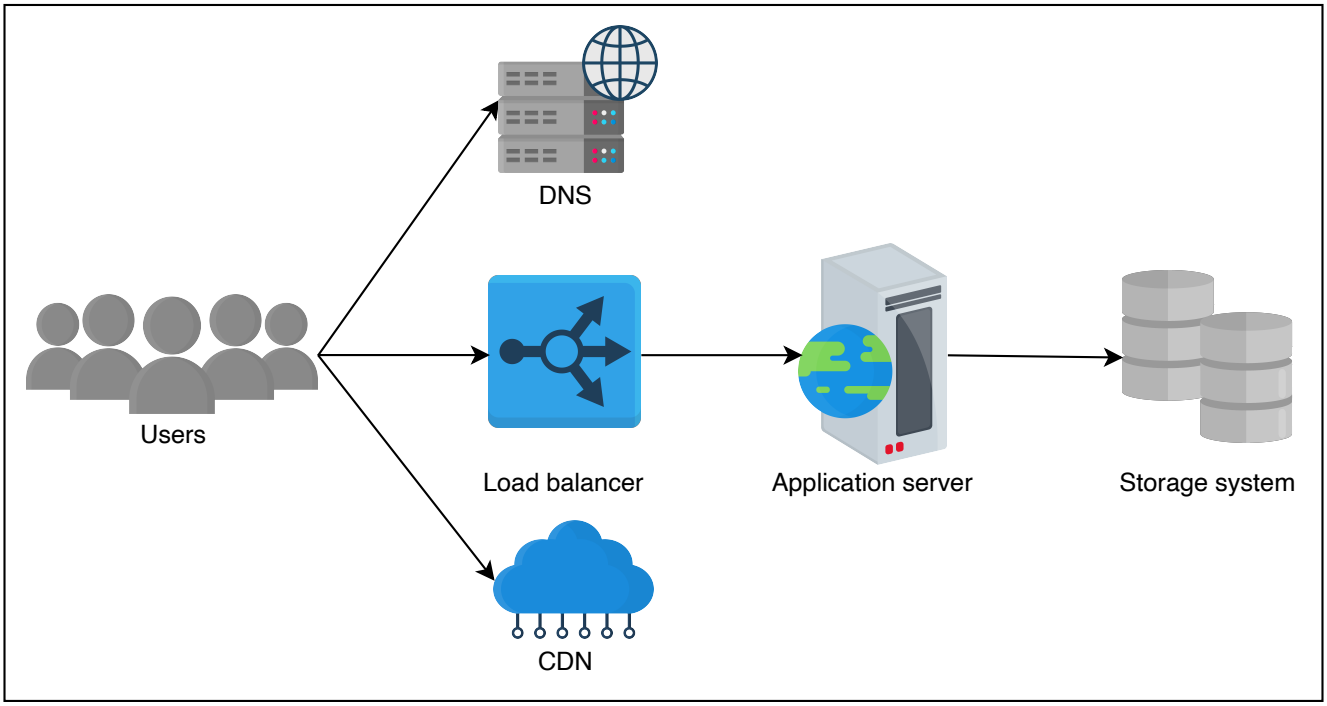
High-level Design of Twitter

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.



Twitter components

- **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 parameters 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, med
ia_field, post_time, tweet_location, list_of_used_hashtags, list_of_ta
gged_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
`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 size of a video.
`media_field`	It specifies the type of media (image, video, GIF, and so on) deliv

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?

Hide Answer

The text limit is 280 characters in a Tweet. So, users can use the hashtag(s) such that text length (including plain text, any links, and hashtags) does not exceed the limit of 280 characters.

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, ex
pansions, sort_order, next_token, user_location)
```

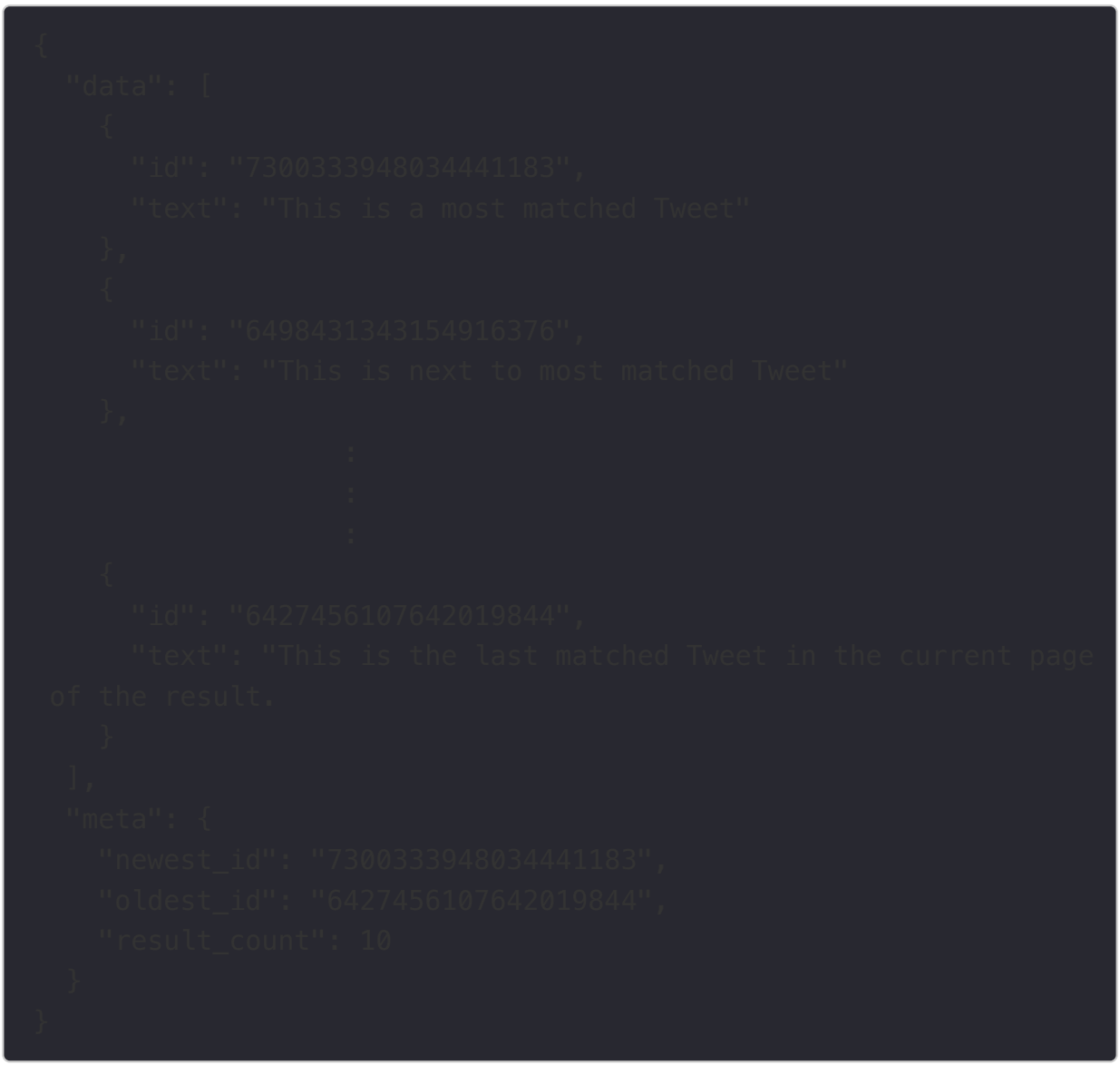
Some new parameters introduced in this case are:

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, is 10.
`exclude`	It specifies what to exclude from the returned Tweets, that is, repli maximum limit on returned Tweets is 3200, but when we exclude limit is reduced to 800 Tweets.
`media_field`	It specifies the media (image, video, GIF) delivered in each returne
`expansions`	It enables us to request additional data objects in the returned Tw mentioned user, referenced Tweet, attached media, attached place
`sort_order`	It specifies the order in which Tweets are returned. By default, it w Tweets first.
`next_token`	It is used to get the next page of results. For instance, if `max_result` the result set contains 200 Tweets, then the value of `next_token` is response to request the next page containing the following 100 Tw 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.

Hide Hint



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_location)
```

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?

Hide Answer

The decision to send specified promoted ads is made on the `user_location` parameter. For example, the user belongs to the `NewYork` city, so most probably it gets promoted ads associated or originated in that region.

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 Tv
`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, the 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.

Detailed Design of Twitter

Take a deep dive into the detailed design of Twitter.

We'll cover the following

Storage system

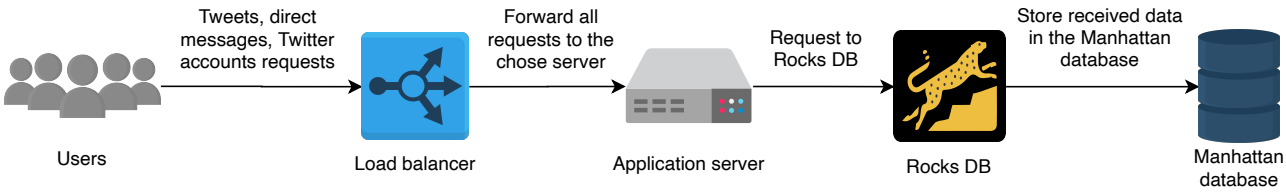
Storage is one of the core components in every real-time system. Although we have a detailed chapter on storage systems, here, we'll focus on the storage system used by Twitter specifically. Twitter uses various storage models for different services to take full advantage of each model. We'll discuss 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.

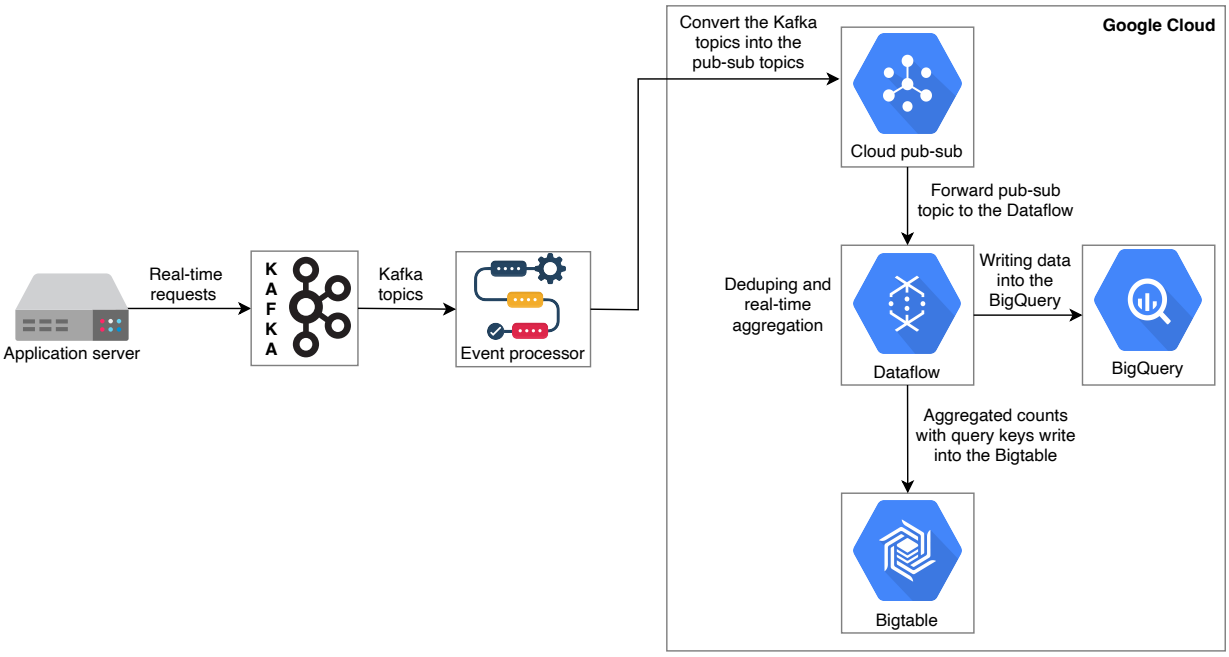


Twitter's data move to the Manhattan database

- **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.

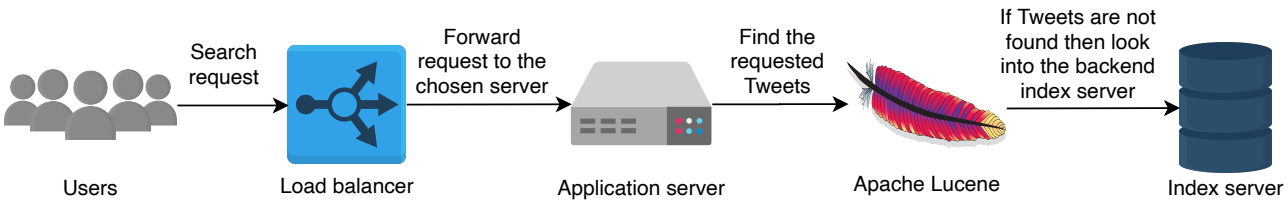
- **Kafka and Cloud dataflow**: Twitter evaluates around 400 billion real-time events and generates petabytes of data every day. For this, it processes events using Kafka on-premise and uses Google Dataflow jobs to handle deduping and real-time aggregation on Google Cloud. After aggregation, the results 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. See the Pub-sub chapter for a deep dive into this.



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) in RAM for low latency and quick updates. The full index is a hundred times larger than the real-time index. However, Twitter performs batch processing for the full indexes. See the Distributed Search chapter to deep dive into how indexing works.



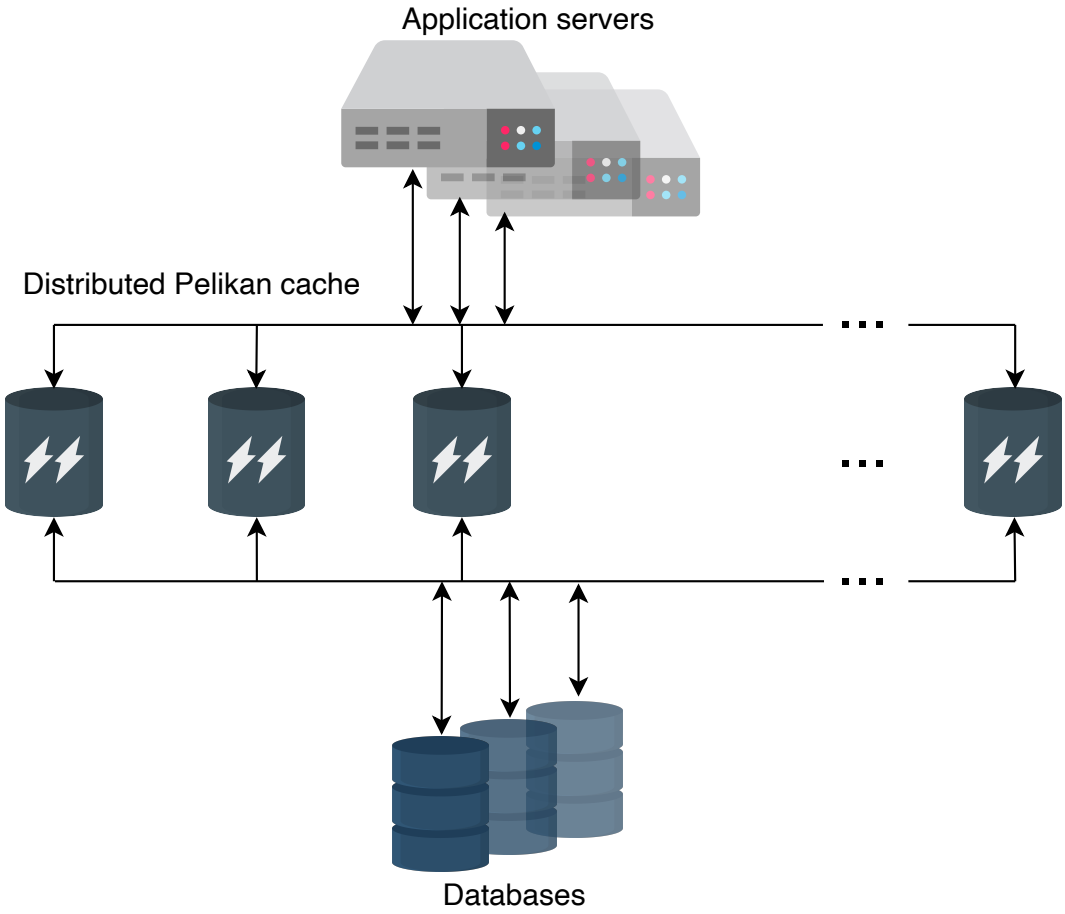
Processing the search request

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

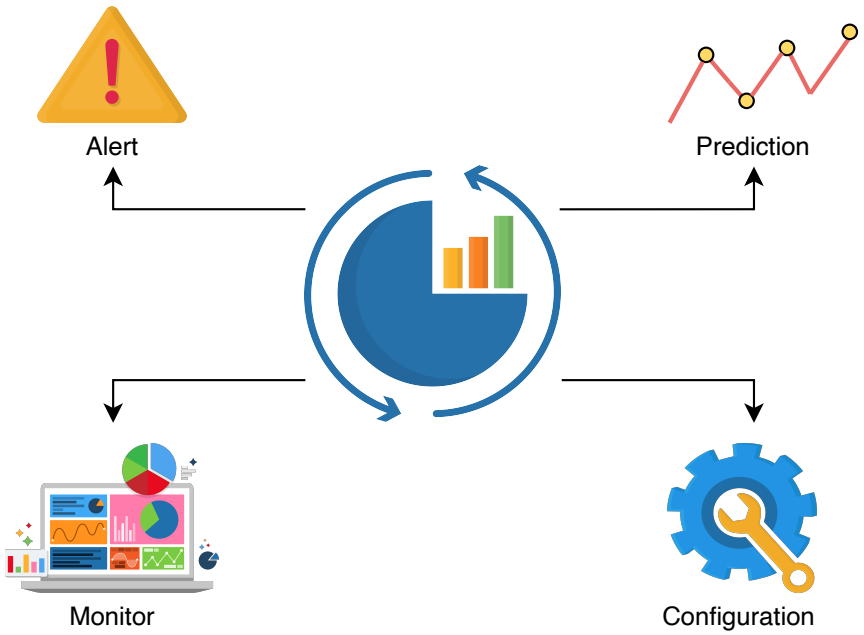
> **Note:** Pelikan also introduced another back-end server named **Segcache**, which is extremely scalable and memory-efficient for small objects. Typically, the median size of the small object is between 200 to 300 bytes in a large-scale application's cache. Most solutions (Memcache and Redis) have 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

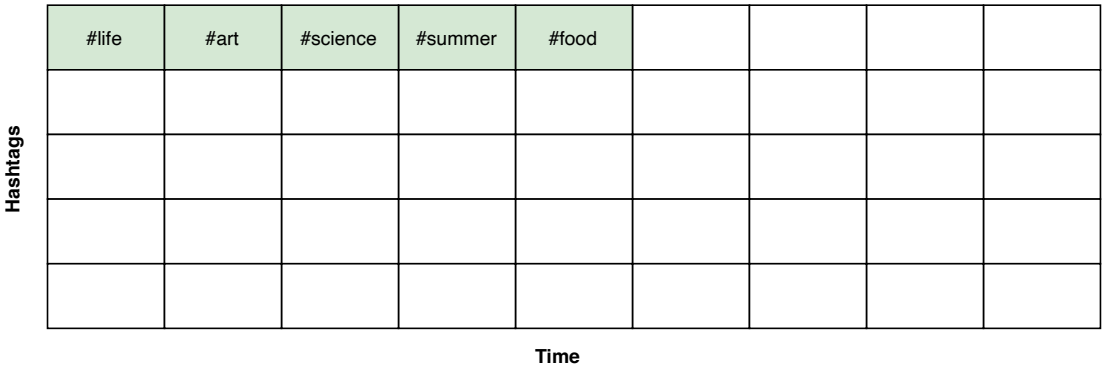
clusters' topology information, metadata, and so on. Twitter also uses it for the leader election on the various systems.

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.

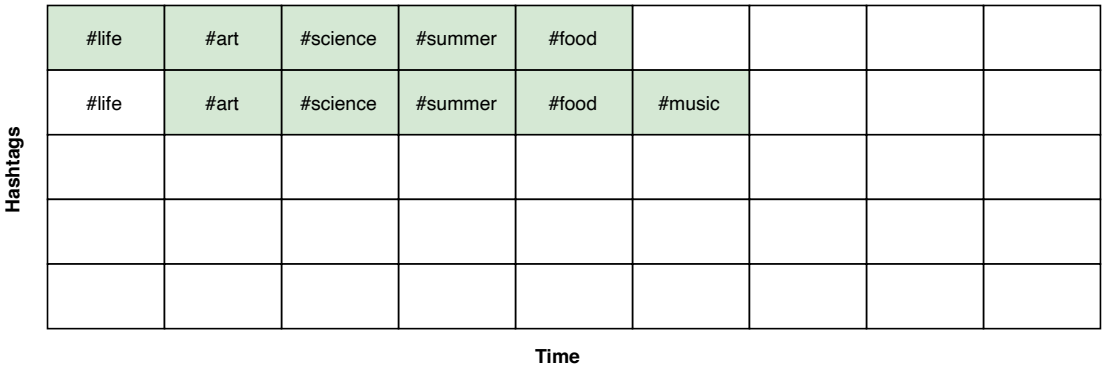
Moreover, a single counter for each specific operation on the particular Tweet is not enough. It's challenging to handle millions of increments or decrements requests against a particular Tweet in a single counter. Therefore, we need multiple distributed counters to manage burst write requests (increments 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.



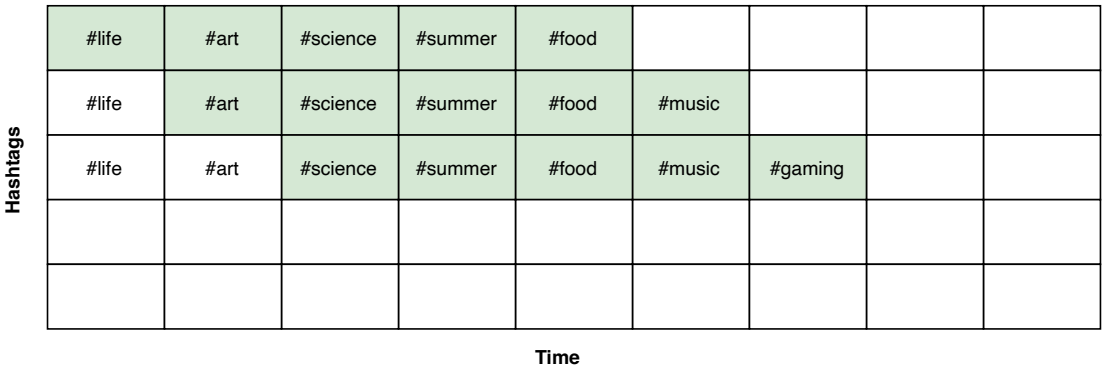
Top-k trends in a specified time frame

1 of 5

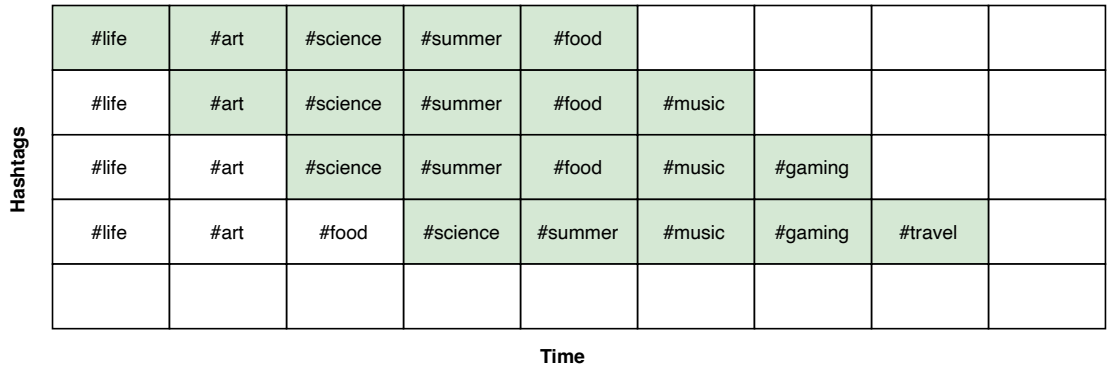


Dropped #life hashtag and showing next top-k trends in a specified time frame

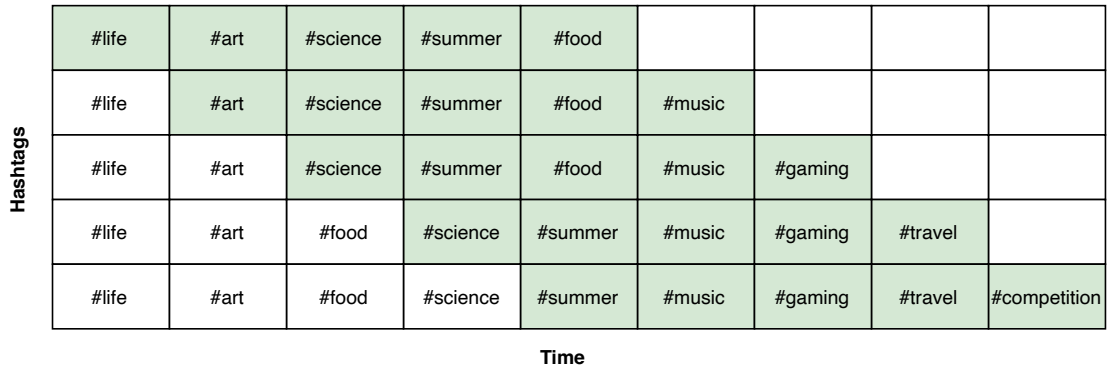
2 of 5



Dropped #life and #art hashtags and showing next top-k trends in a specified time frame



Dropped #life, #art, and #food hashtags and showing next top-k trends in a specified time frame



Dropped #life, #art, #food, and #science hashtags and showing next top-k trends in a specified time frame

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 increase 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:

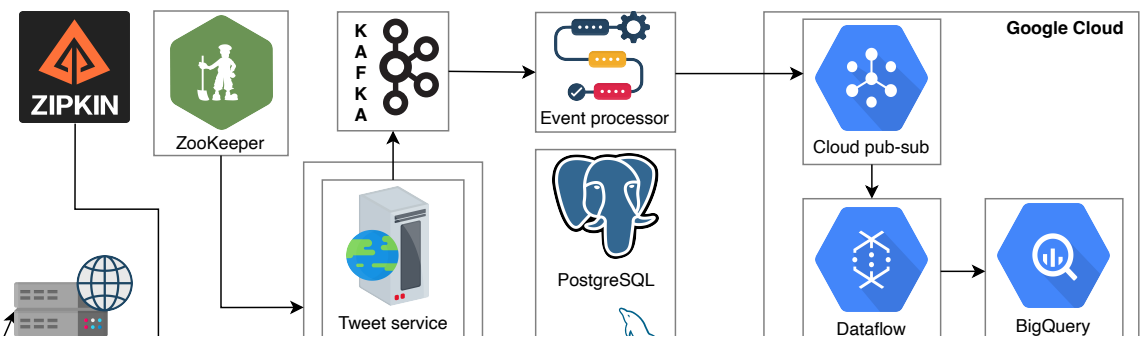
- First, end users get the address of the nearest load balancer from the local DNS.
- Load balancer routes end users' requests to the appropriate servers according to the requested services. Here, we'll discuss the Tweet, timeline, and search services.
    - **Tweet service**: When end users perform 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

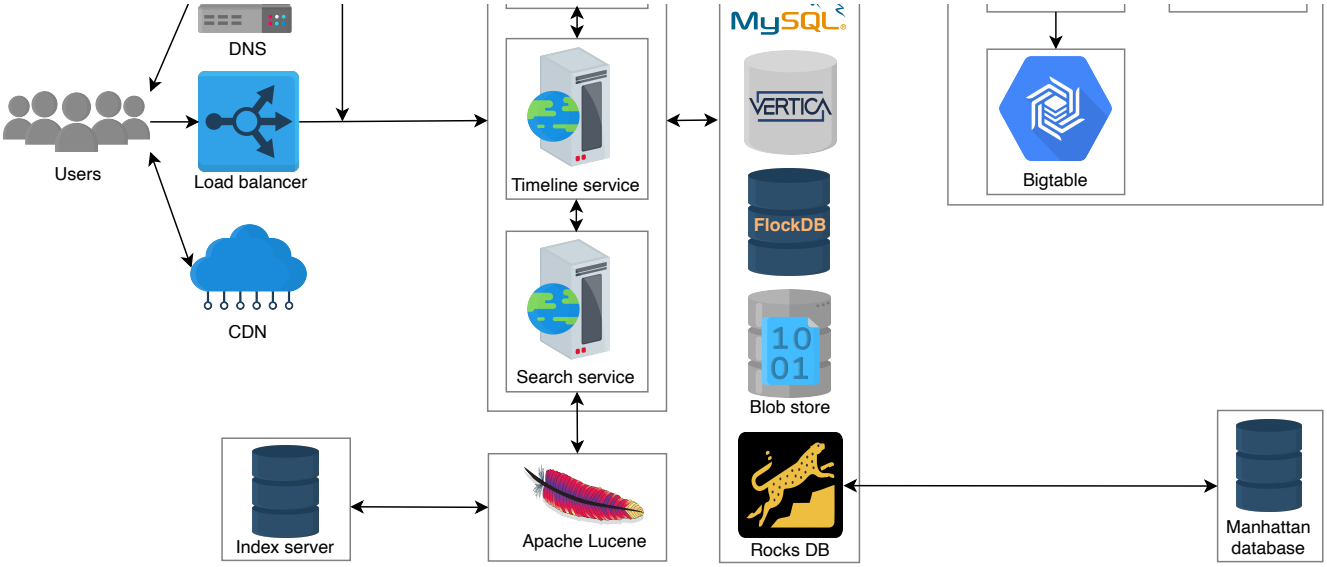
other metrics from the real-time streams and client logs, is achieved in the Apache Kafka.

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.





Twitter design overview

Back

Mark As Completed

Next

High-level Design of Twitter

Client-side Load Balancer for Twitter

Client-side Load Balancer for Twitter

Let's understand how Twitter performs client-side load balancing.

We'll cover the following

Introduction

In the previous lesson, we conceived the design of Twitter using a dedicated load balancer. Although this method works, and we’ve employed it in other designs, it may not be the optimal choice for Twitter. This is because Twitter offers a variety of services on a large scale, using numerous instances and dedicated load-balancers are not a suitable choice for such systems. To understand the concept better, let’s understand the history of Twitter’s design.

Twitter’s design history

The initial design of Twitter included a monolithic (Ruby on Rails) application with a MySQL database. As Twitter scaled, the number of services increased and the MySQL database was sharded. A monolithic application design like this is a disaster because of the following reasons:

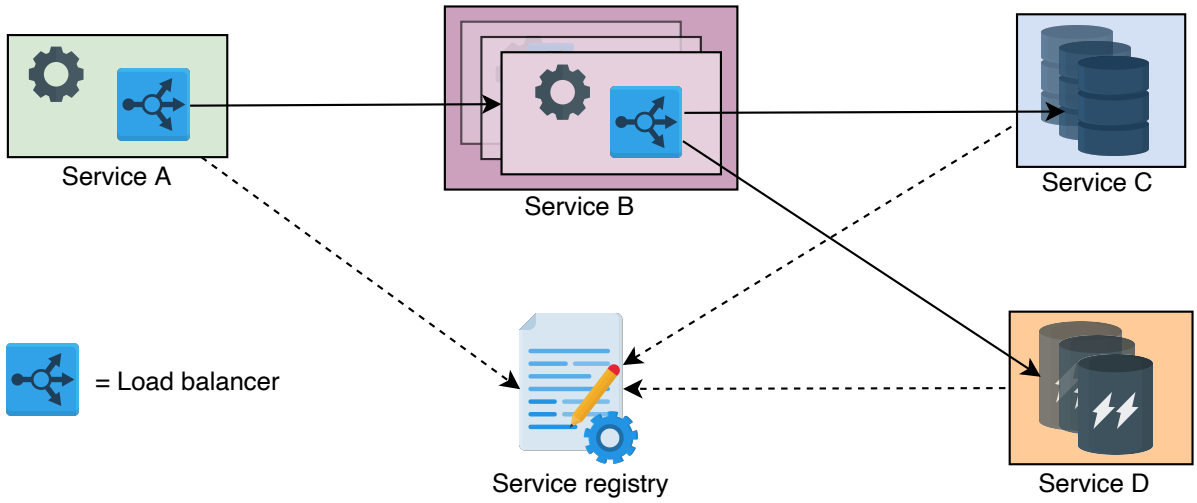
- A large number of developers work on the same codebase, which makes it difficult to update individual services.
- One service’s upgrade process may lead to the breaking of another.

- Hardware costs grow because a single machine performs numerous services.
- Recovery from failures is both time-consuming and complex.

With the way Twitter has evolved, the only way out was many microservices where each service can be served through hundreds or thousands of instances.

Client-side load balancing

In a client-side load balancing technique, there's no dedicated intermediate load-balancing infrastructure between any two services with a large number of instances. A node requesting a set of nodes for a service has a built-in load balancer. We refer to the requesting node or service as a client. The client can use a variety of techniques to select a suitable instance to request the service. The illustration below depicts the concept of client-side load balancing. Every new arriving service or instance will register itself with a service registry so that other services are aware of its existence



How client-side load balancing works

In the diagram above, Service A has to choose a relatively less-burdened instance of Service B. Therefore, it will make use of the load balancer component inside it to choose the most suitable instance of Service B. Using the same client-side load balancing method, Service B will talk to other services. It's clear that Service A is the client when it is calling Service B, whereas Service B is the client when it is talking to Service C and D. As a result, there will be no central entity doing load balancing. Instead, every node will do load balancing of its own.

Advantages: Using client-side load-balancing has the following benefits.

- Less hardware infrastructure/layers are required to do load balancing.
- Network latency will be reduced because of no intermediate hop.
- Client-side load balancers eliminate bandwidth bottlenecks. On the other hand, in a dedicated load balancing layer, all requests go through a single machine that can choke if traffic multiplies.
- Fewer points of failure in the overall system.
- There is no end users queue waiting for the resource (server) for the particular services because many load balancers are routing the traffic. Eventually, it increases the quality of experience (QoE).

Many real-world applications like Twitter, Yelp, Netflix, and others use client-side load balancing. We'll discuss the client-side load balancing techniques used by Twitter in the next section.

Client-side load balancing in Twitter

Twitter uses a client-side load balancer referred to as **deterministic aperture** that is part of a larger RPC framework called Finagle. **Finagle** is a protocol-agnostic, open-source, and asynchronous RPC library.

> **Note:** Consider the following question. What does the client-side load balancer of Twitter balance on? That is, what parameters are used to fairly balance the load across different servers?

Twitter primarily uses two distributions to measure the effectiveness of a load balancer:

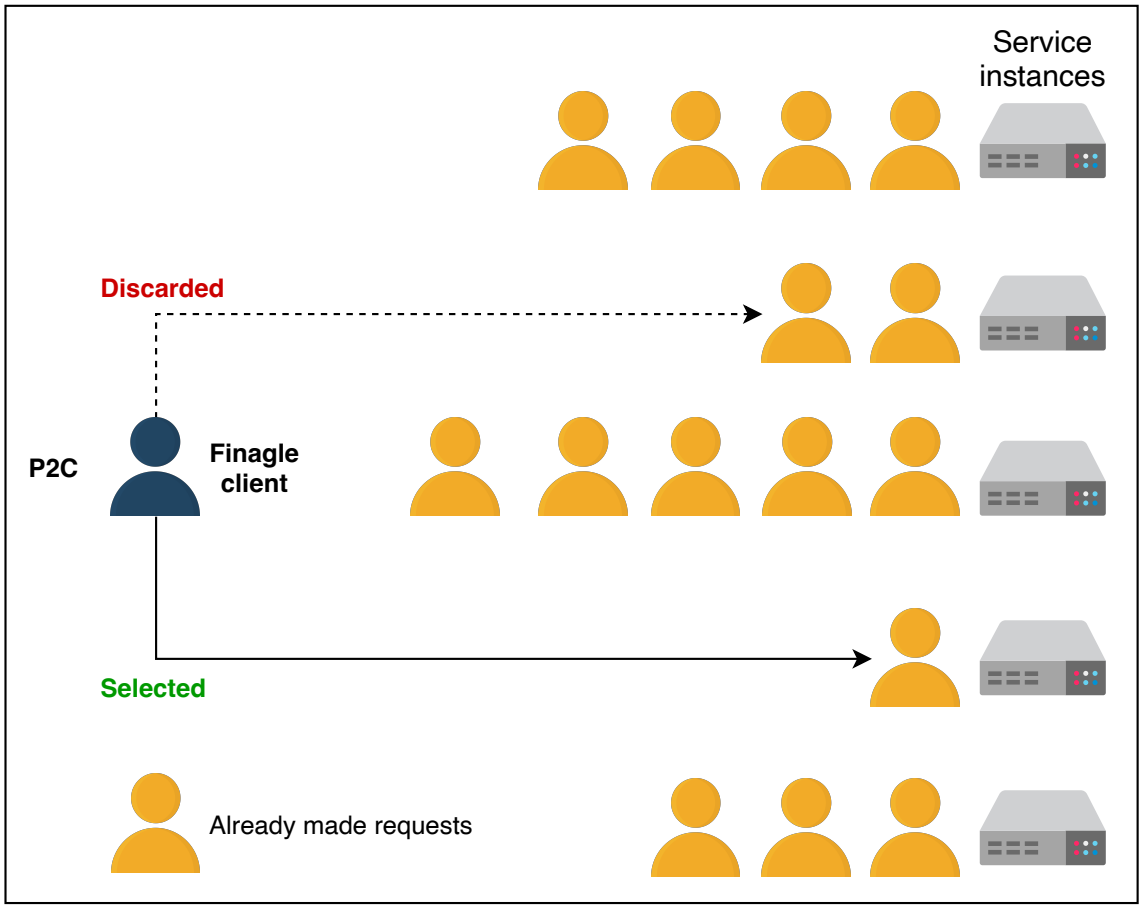
1. The distribution of requests (OSI layer 7)
2. The distribution of sessions (OSI layer 5)

Although requests are the key metric, sessions are an important attribute for achieving a fair distribution. Therefore, we'll develop techniques for fair distribution of requests as well as sessions. Let's start with the simple technique

of Power of Two Random Choices (P2C) for request distribution first.

Request distribution using P2C

The **P2C technique** for request distribution gives uniform request distribution as long as sessions are uniformly distributed. In P2C, the system randomly picks two unique instances (servers) against each request, and selects the one with the least amount of load. Let's look at the illustration below to understand P2C.



Power of two random choices

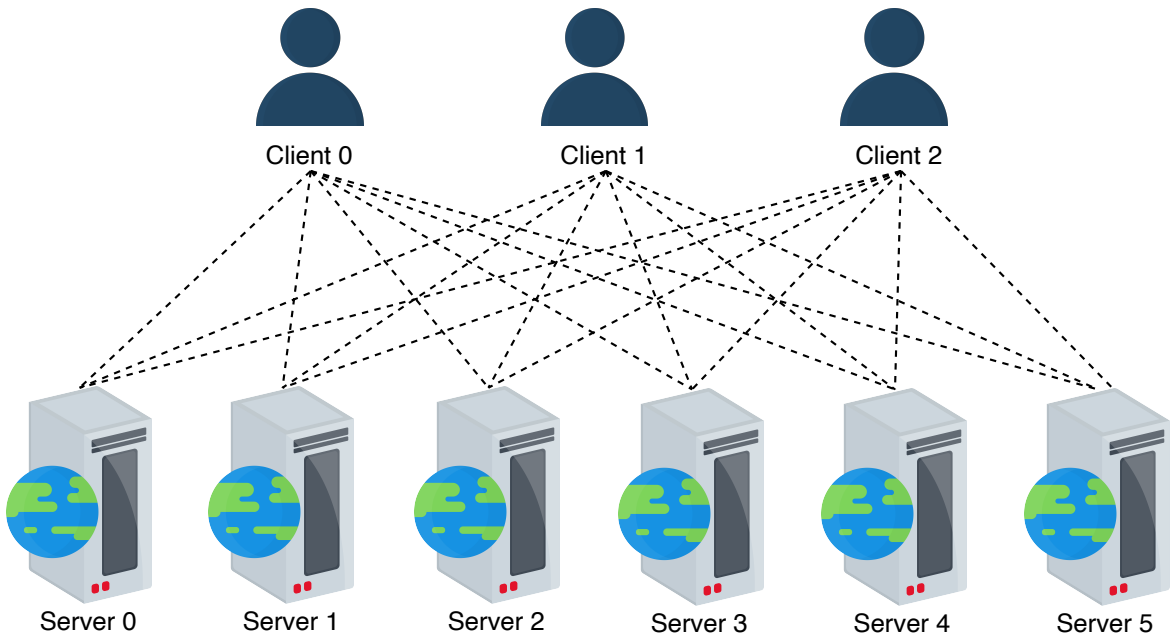
P2C is based on the simple idea that comparison between two randomly selected nodes provides load distribution that is exponentially better than random selection

Now that we've established how to distribute requests fairly, let's explore different techniques of session distribution.

Session distribution

Solution 1: We'll start our discussion with the **Mesh topology**. Using this

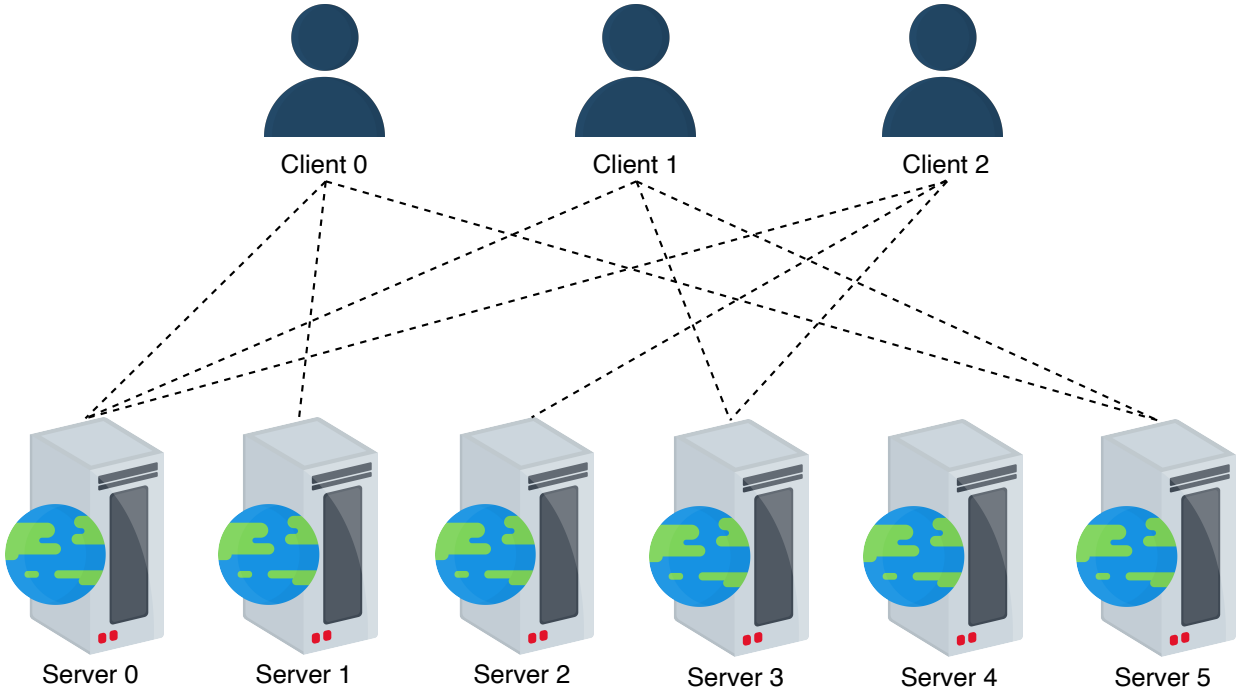
approach, each client (load balancer) starts a session with all of the given instances of a service. The illustration below represents the concept of mesh topology.



Mesh topology

Obviously, this method is fair because the sessions are evenly distributed. However, the even distribution of sessions comes at a very high cost, especially when scaled to thousands of servers. Apart from that, some unhealthy or stale sessions may lead to failures.

Solution 2: To solve the problem of scalability with solution 1, Twitter devised another solution called **random aperture**. This technique randomly chooses a subset of servers for session establishment.



Random aperture

Of course, random selection will reduce the number of established sessions as we can see in the diagram above. However, the question is, how many servers will be chosen randomly? This is not an easy question to answer and the answer varies, depending on the request concurrency of a client.

To answer the question above, Twitter installed a **feedback controller** inside the random aperture that decides the subset size based on the client's load. As a result, we're able to reduce the session load and ensure scalability. However, this solution isn't fair. We can see the imbalance in the illustration above such that **Server 4** has no session whereas **Server 0** has three sessions.

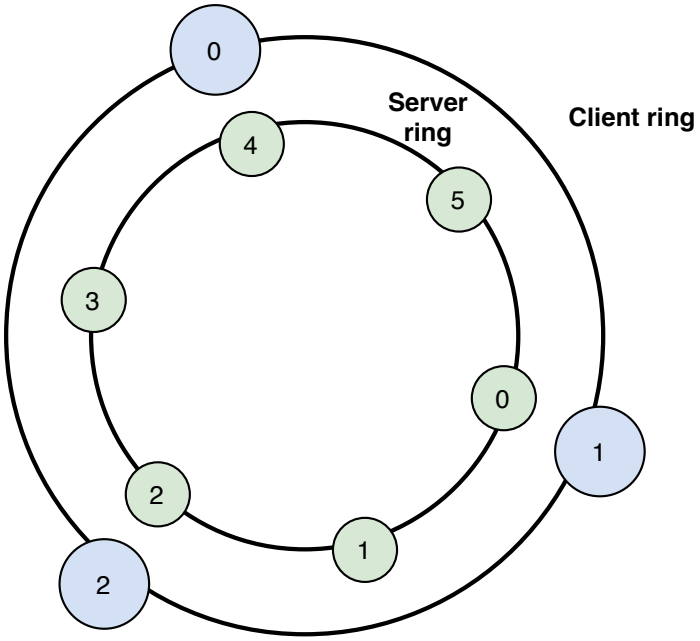
This imbalance creates problems such as idle servers, high load on a few servers, and so on. Therefore, we need a solution that is both fair and scalable.

Solution 3: We learned from solution 2 that we can scale by subsetting the number of session establishments. However, we failed to do the session distribution fairly. To resolve the problem, Twitter came up with a **deterministic aperture** solution. The key benefits of this approach are:

- Little coordination is required between clients and servers to converge on a solution.

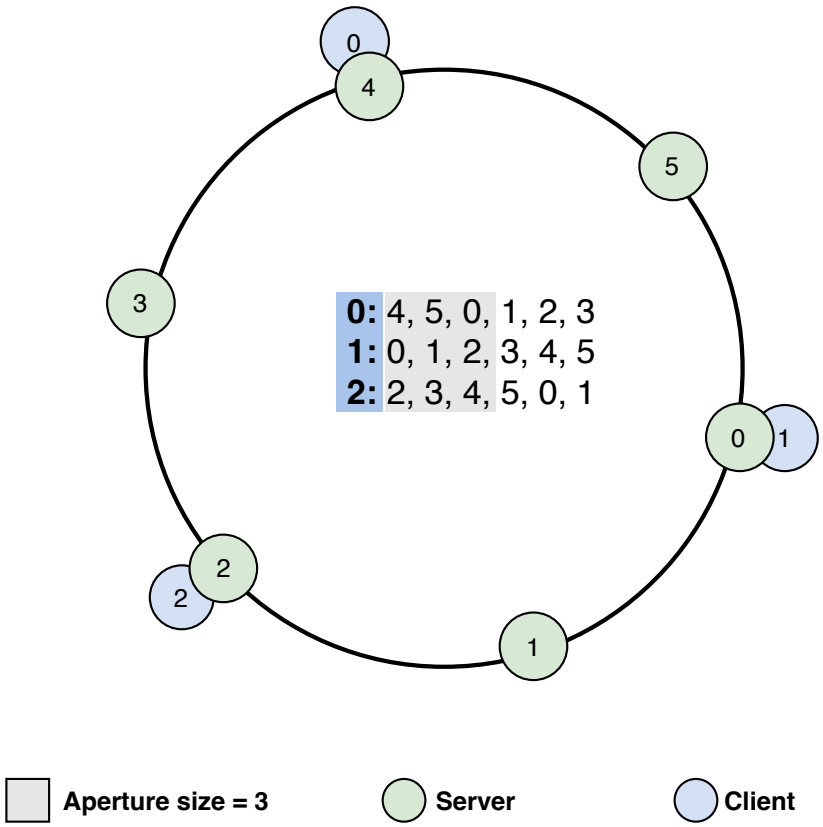
- Minimal disruption occurs if there are any changes in the number of clients or server instances for whatever reason.

For this solution, we represent clients and servers in a ring with equally distanced intervals, where each object on the ring represents the node or server labeled with a number. The illustration below shows what Twitter refers to as the discrete ring coordinates (that is, a specific point on the ring).



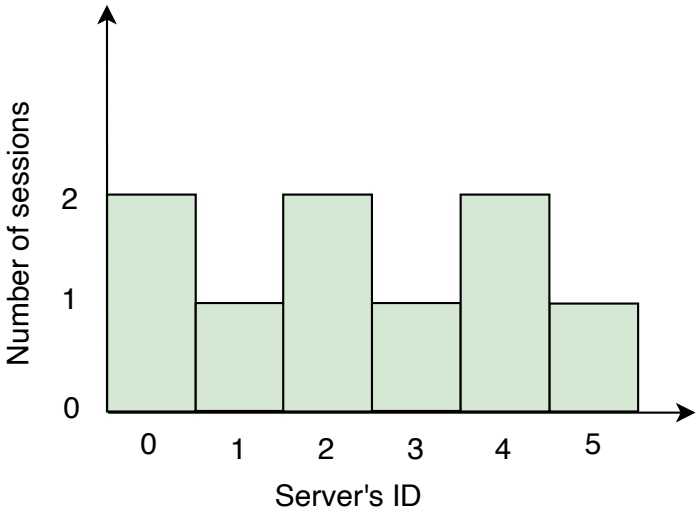
Ring coordinates

Now, we compose client and server rings to derive a relationship between them. We've also defined an aperture size of 3. Each client will create a session with three servers in the ring. Clients select a server by circulating clockwise in the ring. This approach is similar to consistent hashing, except that here we guarantee equal distribution of servers across the ring. Let's see the illustration below where clients have sessions with servers chosen from the given list of servers, respectively.



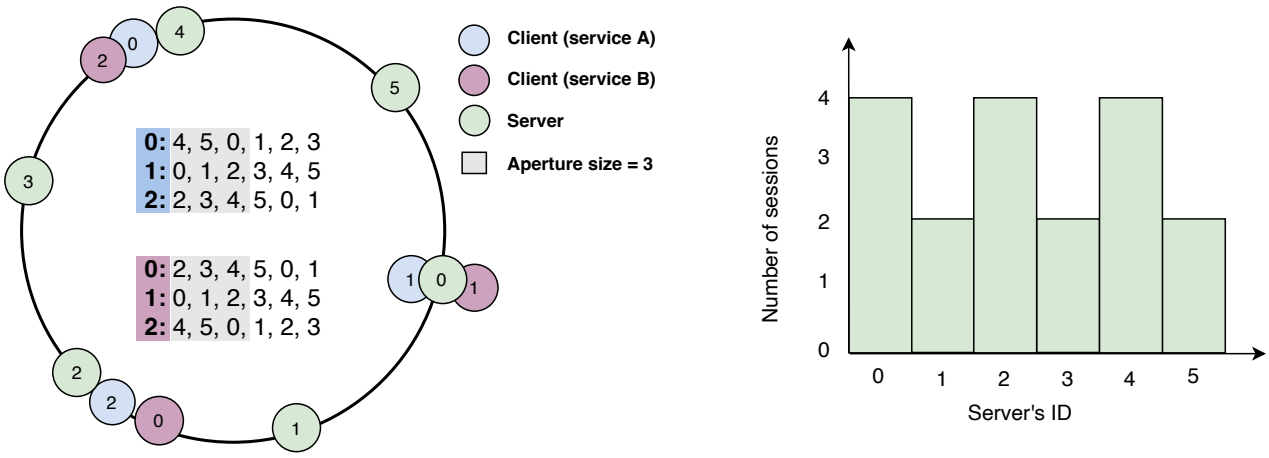
Merge client and server ring coordinates

In the diagram above, client 0 establishes sessions with servers 4, 5, and 0. Client 1 establishes sessions with servers 0, 1, and 2, whereas Client 2 establishes sessions with servers 2, 3, and 4. When the aperture rotates or moves in the given subset of arrays of the same size, each client's service has new servers available to create sessions. The number of sessions established per server is shown in the figure below.



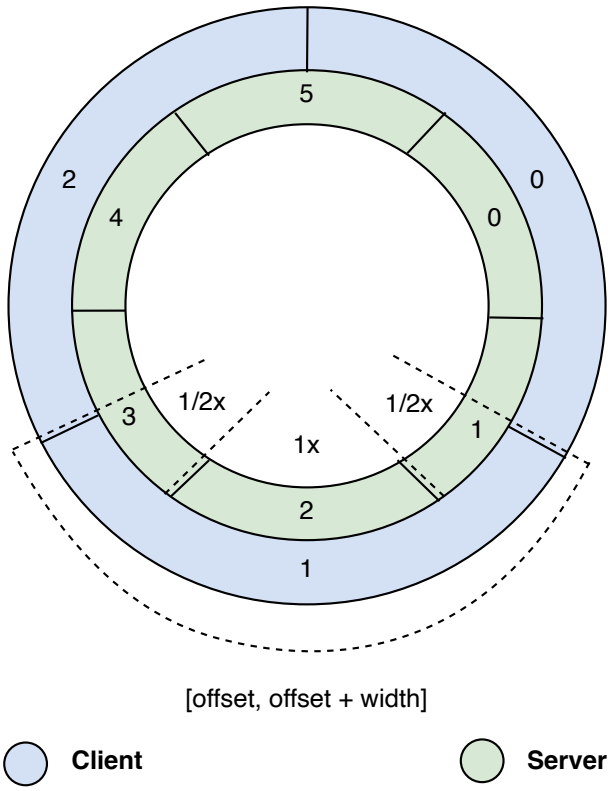
Number of sessions established per server

We can see that there's no idle server or server with a high load in the histogram graph above. Thus, we achieved a fair distribution for the clients belonging to the same service. It's clear that we've half resolved the problem by improving the session distribution among servers as compared to the random aperture (solution 02). However, the problem of unfair distribution can escalate as the number of clients grows. In that case, some servers will get crowded when more than one client (service) talks to the back-end servers. Let's see the illustration below to understand the problem.



Two different clients creating session with backend servers

The illustration above depicts that the problem compounded as we increased the number of clients (services). The solution is continuous ring coordinates where we derive relationships between clients and servers using overlapping ring slices, rather than specific points on the rings. The important thing here is that the overlapping of rings can be partial since servers on the ring may not have enough instances to divide among the clients equally. Let's look at the illustration below.



Overlapping among clients' services and servers

The illustration above shows that clients 0 and 1 (symmetrically) share the same server (server 1). It means that there is fractional overlapping concerning the ring coordinates. However, this overlapping can lead to asymmetric sharing as well. Symmetric or asymmetric, the load balancing can be done through P2C. For instance, in the diagram above, servers 1 and 3 will receive half the load that server 2 gets. This means that a server can receive more or less load, depending on its overlapped slice size with a client.

Now, we know how to map clients to the subset of the servers. We'll use P2C with continuous ring coordinates. The client will pick points (coordinations) in its range instead of picking the instance of the server. The selection process of

clients to create sessions with backend servers are as follows:

1. Pick two coordinates (offset, offset + width) within its range and map these coordinates to the discrete instances (servers).
2. Select the one instance with the most negligible load from two chosen instances (P2C) by its degree of intersection.

> **Note:** Continuous aperture is scalable and fair at the cost of an additional session over boundary node due to partial overlapping. It is also able to achieve little coordination and minimal disruption.

Let's look at the below table to get an overview of the discussed approaches.

Summary of Solutions for Session Distribution

Approach	Scalability	Fair distribution	
Mesh topology	✗	✓	
Random aperture	✓	✗	
Deterministic aperture (discrete ring)	✓	**weak**	
Deterministic aperture (continuous ring)	✓	✓	

Conclusion

This design problem highlights that tweaking the performance of common building blocks and utilizing the right (rich) combination of them is necessary for real-world services. Our needs from one service to the next will dictate which design point we choose in our solution space. In this chapter, we saw examples

of how Twitter uses a combination of data stores and custom algorithms for load-balancing.

Back

Mark As Completed

Next

Detailed Design of Twitter

Quiz on Twitter's Design

System Design: Newsfeed System

Learn about the basics of designing the newsfeed system.

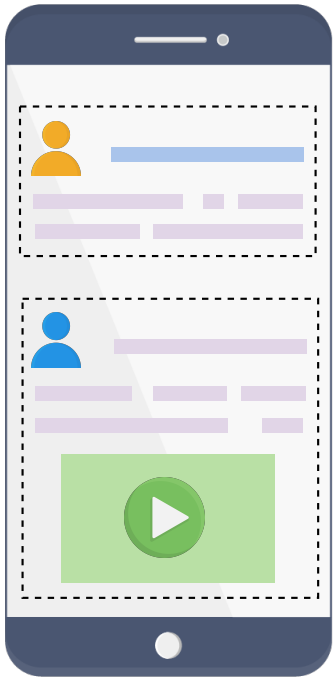
We'll cover the following

What is a newsfeed?

A **newsfeed** of any social media platform (Twitter, Facebook, Instagram) is a list of stories generated by entities that a user follows. It contains text, images, videos, and other activities such as likes, comments, shares, advertisements, and many more. This list is continuously updated and presented to the relevant users on the user's home page. Similarly, a newsfeed system also displays the newsfeed to users from friends, followers, groups, and other pages, including a user's own posts.

A newsfeed is essential for social media platform users because it keeps them informed about the latest industry developments, current affairs, and relevant information. It also provides them additional reasons to return and connect with a platform on a regular basis. Billions of users use such platforms. The challenging task is to provide a personalized newsfeed in real-time while keeping the system scalable and highly available.

This chapter will discuss the high-level and detailed design of a newsfeed system for a social platform like Facebook, Twitter, Instagram, etc.



Newsfeeds on a mobile application

How will we design the newsfeed system?#

We have divided the design of the newsfeed system into the following three lessons:

1. **Requirements:** In this lesson, we'll identify the functional and non-functional requirements. We'll also estimate the resource requirements to provide a personalized newsfeed to billions of users each day.
2. **Design:** We'll discuss the high-level and detailed design of the newsfeed system in this lesson. We'll also describe the API design and database schema for our proposed design. Moreover, this lesson will also help us to rank newsfeed to provide a better experience to users.
3. **Evaluation:** In this lesson, we'll evaluate the design of the newsfeed system based on the non-functional requirements. We'll also take a quiz to assess our understanding of the design of the newsfeed system.

Let's list down the requirements for designing our version of the newsfeed

system.

Back

Mark As Completed

Next

Quiz on Twitter's Design

Requirements of a Newsfeed Syste...

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

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.

1. **Users' metadata storage estimation:** Suppose the storage required for one user's metadata is 50 KB. For 1 billion users, we would need $1B \times 50KB = 50TB$.

   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' met adata (in KBs)	50
Total storage required for all users (i n 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:

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

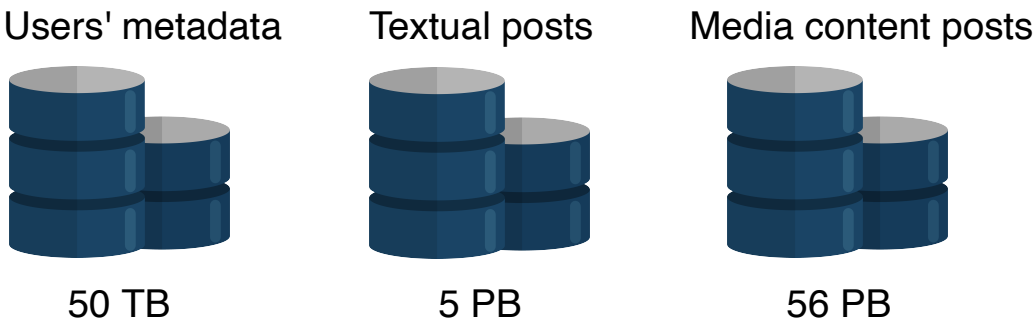
3. **Media content storage estimate:** Along with text, a post can also contain media content. Therefore, we assume that $1/5th$ posts have videos and $4/5th$ include images. The assumed average image size is 200KB and the video size is 2MB.

   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.

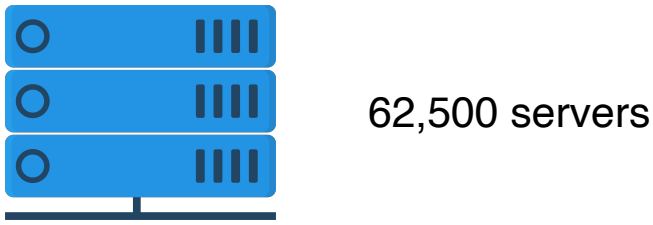
Number of active users (in million)	500
Maximum allowed text storage per post (in KBs)	50
Number of precomputed posts per user (top N)	200
Storage required for textual posts (in PBs)	f 5
Total required media content storage for active users (in PBs)	f 56

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:

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



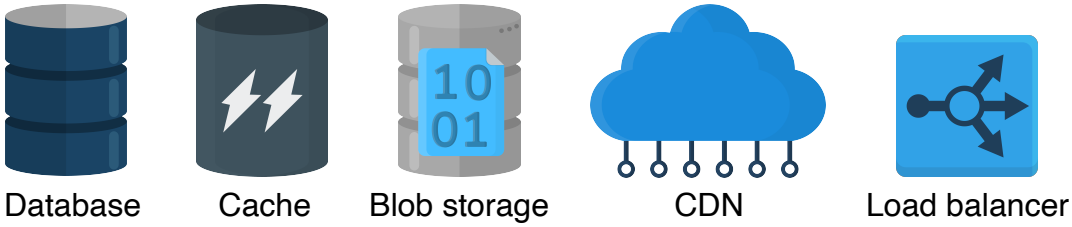
Number of servers required for the newsfeed system

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

☑ Mark As Completed

Next →

System Design: Newsfeed System

Design of a Newsfeed System

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.

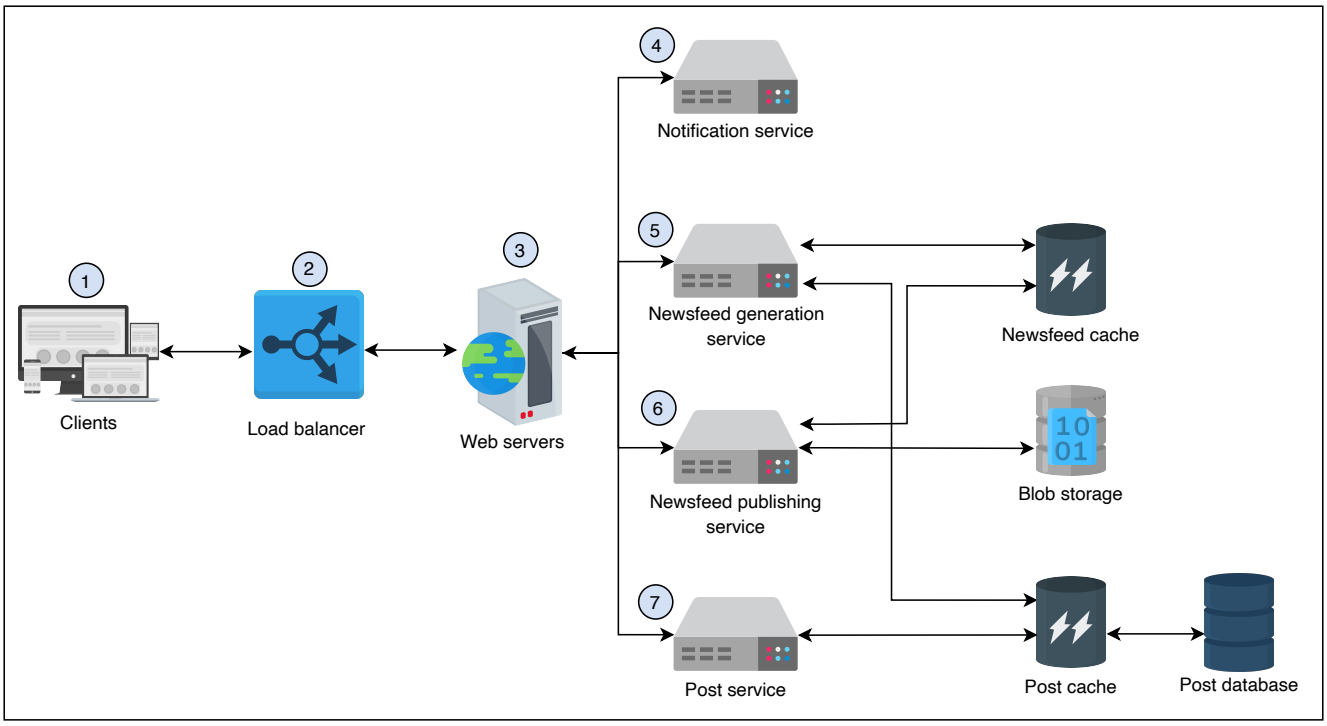
High-level design of a newsfeed system

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 are 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.

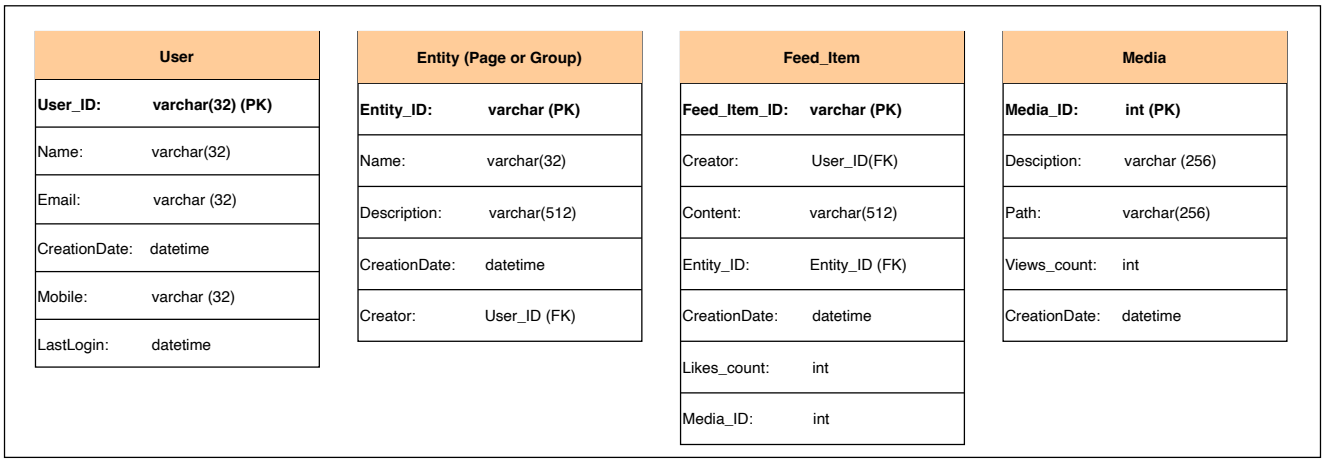
The following parameters are used for this API:

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

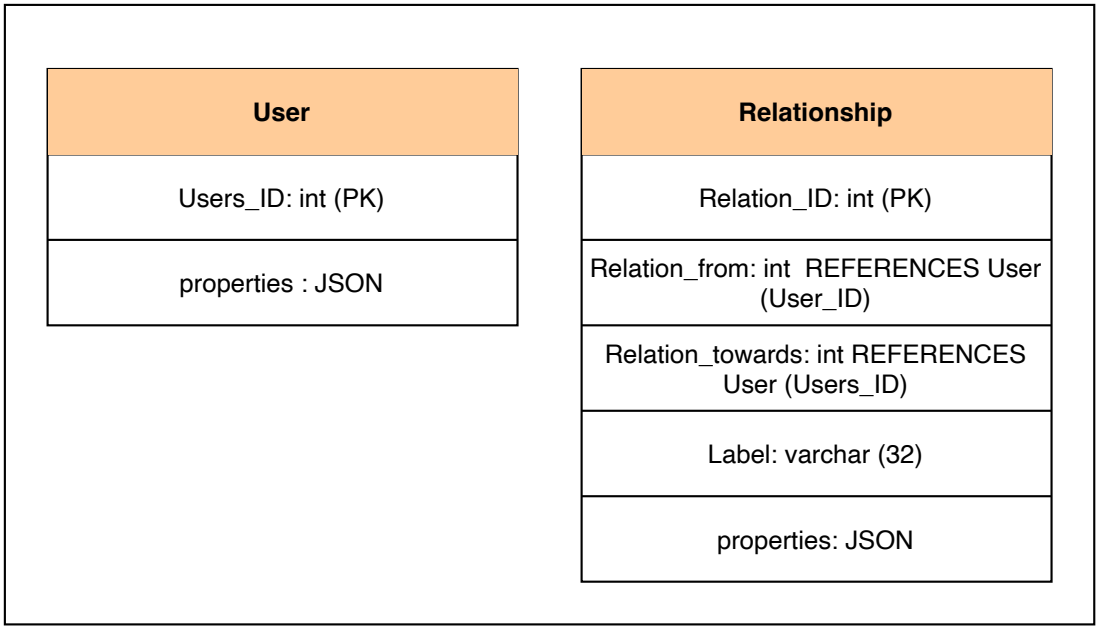
We use a graph database to store relationships between users, friends, and

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

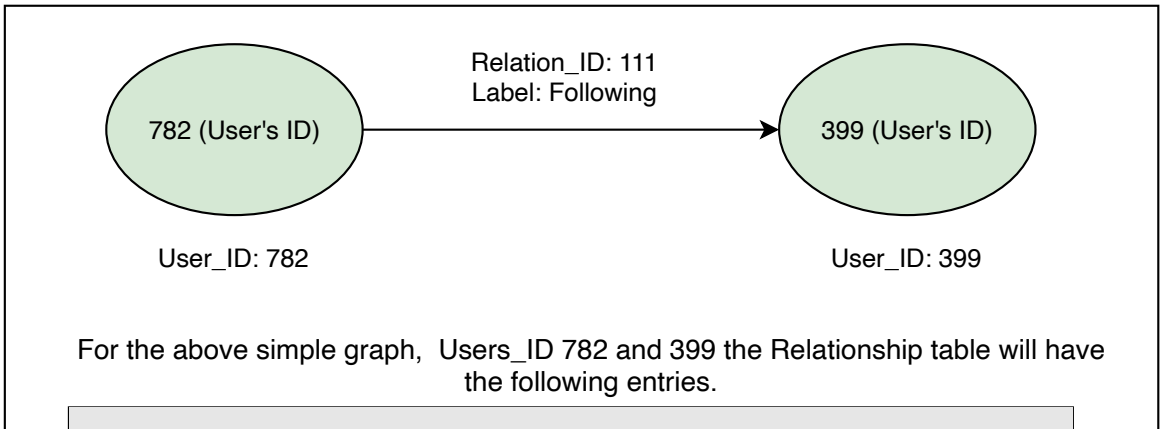
Therefore, we follow a relational schema for the graph store, as shown in the following figure. The schema uses the PostgreSQL JSON data type to store the properties of each vertex (user) or edge (relationship).

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:



Relation_ID	Relation_from	Relation_towards	Label	Properties
111	782	399	Following	Some description as a JSON data type

Relationship

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.

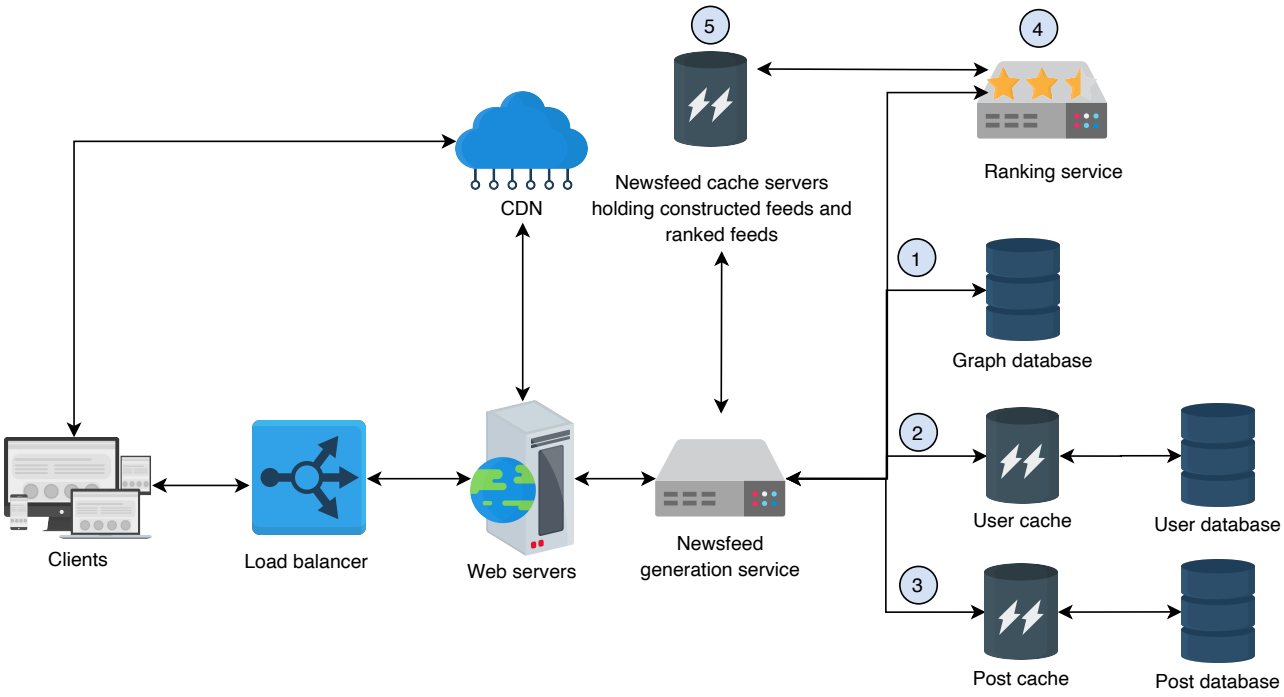
The following steps are performed in sequence to generate a newsfeed for Alice:

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?

Hide Answer

A memory-efficient way would be to store just the mapping between users and their corresponding posts in a table in the cache, that is., `<Post_ID, User_ID>`. During the feed publishing phase, the system will retrieve posts from the post database and generate the newsfeed for a user who follows another user with `User_ID`.

The newsfeed publishing service

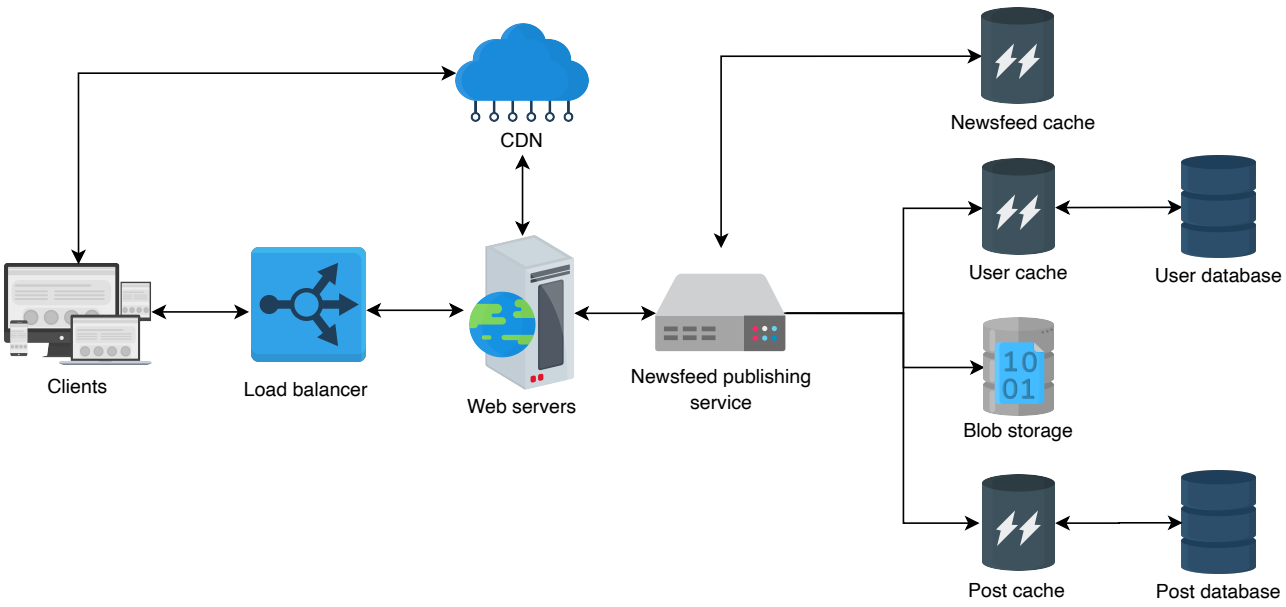
At this stage, the newsfeeds are generated for users from their respective friends, followers, and entities and are stored in the form of `<Post_ID, User_ID>` in the news feed cache.

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?

Hide Answer

When a user, say Bob, creates a new post, it is stored in the post database and cache. In the next step, the **newsfeed generation service** generates a newsfeed for Bob's friends and followers, and the newsfeed cache is updated. The updated newsfeed is delivered to the corresponding users on the next page/screen refresh event.

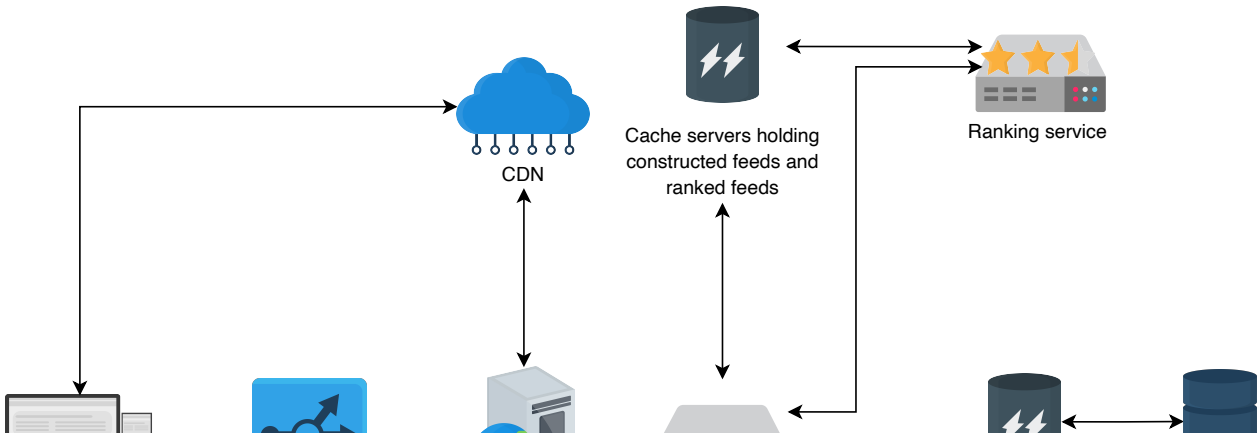
The newsfeed ranking service

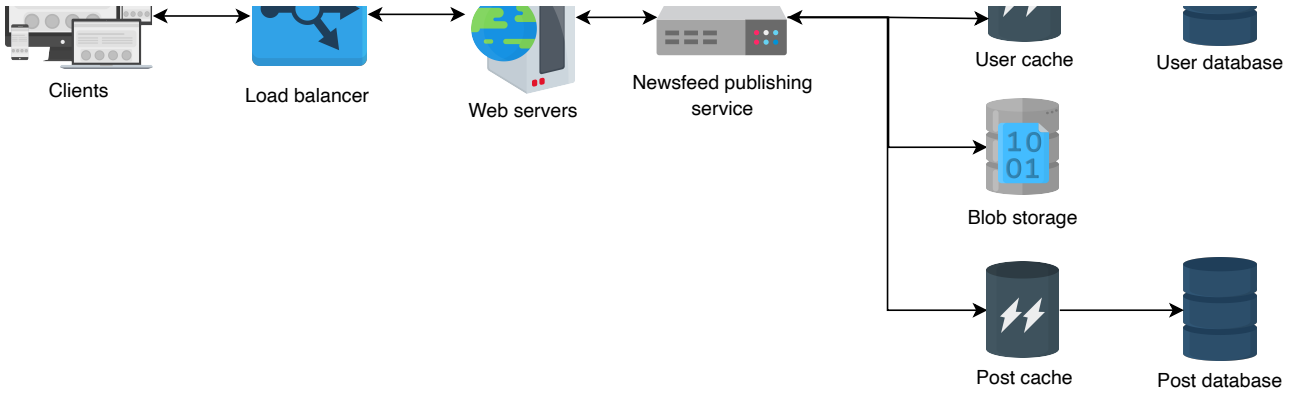
Often we see the relevant and important posts on the top of our newsfeed whenever we log in to our social media accounts. This ranking involves multiple advanced ranking and recommendation algorithms.

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, etc 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:

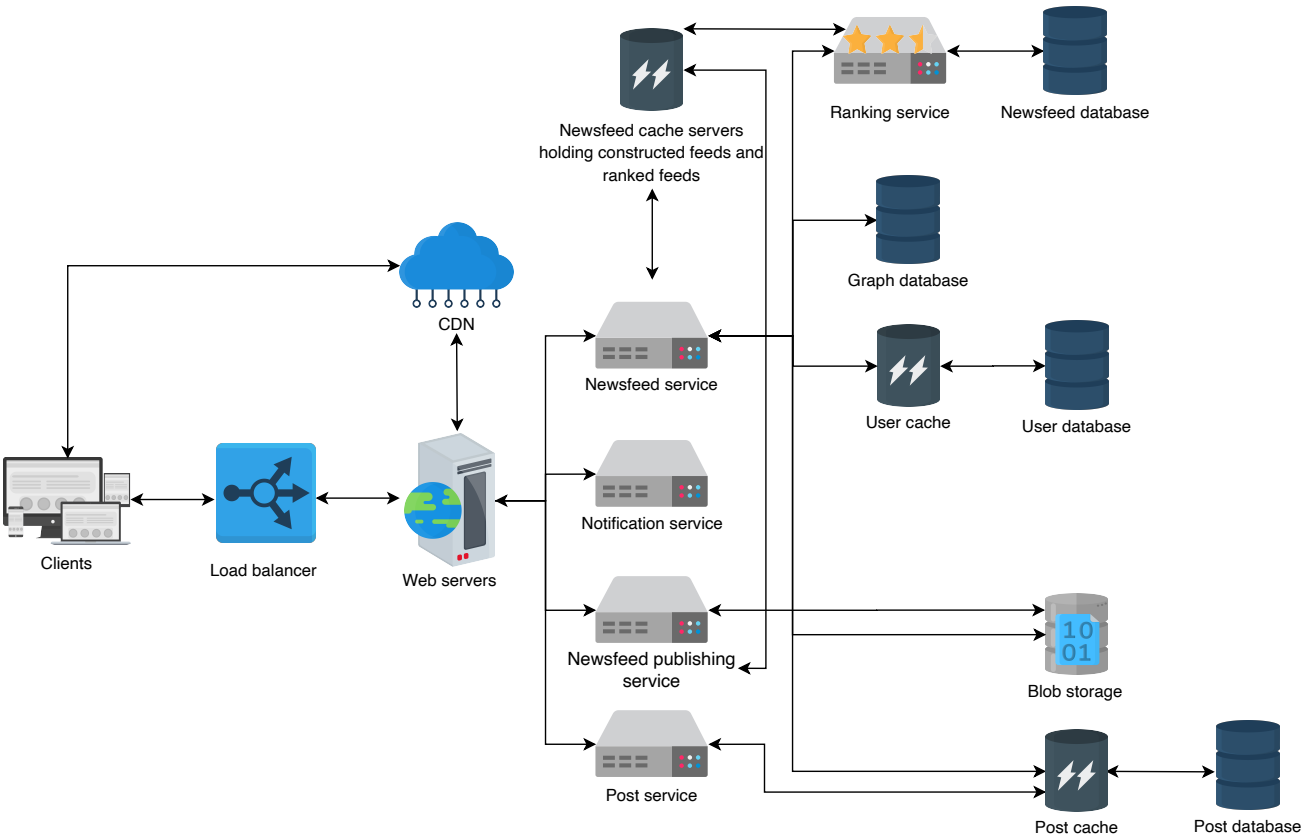
A newsfeed consisting of top 4 posts based on the relevance scores

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 processing 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.

Back

Mark As Completed

Next

Requirements of a Newsfeed Syste...

Evaluation of a Newsfeed System's ...

Evaluation of a Newsfeed System's Design

Evaluate the newsfeed design with respect to its non-functional requirements.

We'll cover the following

Fulfill requirements

Our non-functional requirements for the proposed newsfeed system design are scalability, fault tolerance, availability, and low latency. Let's discuss how the proposed system fulfills these requirements:

1. **Scalability:** The proposed system is scalable to handle an ever-increasing number of users. The required resources, including load balancers, web servers, and other relevant servers, are added/removed on demand.
2. **Fault tolerance:** The replication of data consisting of users' metadata, posts, and newsfeed makes the system fault-tolerant. Moreover, the redundant resources are always there to handle the failure of a server or its component.
3. **Availability:** The system is highly available by providing redundant servers and replicating data on them. When a user gets disconnected due to some fault in the server, the session is re-created via a load balancer with a different server. Moreover, the data (users metadata, posts, and newsfeeds) is stored on different and redundant database clusters, which provides high availability and durability.
4. **Low latency:** We can minimize the system's latency at various levels by:
    - Geographically distributed servers and the cache associated with them. This way, we bring the service close to users.

- Using CDNs for frequently accessed newsfeeds and media content.

Quiz on the newsfeed system's design

Test your understanding of the concepts related to the design of the newsfeed system with a quiz.

1 Which component is responsible for storing relationships between users, their friends, and followers?

Reset Quiz

Question 1 of 4
0 attempted

Submit Answer

Summary

In this chapter, we learned to design a newsfeed system at scale. Our design ranked enormous user data to show carefully curated content to the user for better user experience and engagement. Our newsfeed design is general enough that it can be used in many places such as Twitter feeds, Facebook posts, YouTube and Instagram recommendations, News applications, and so on.