Grokking Modern System Design Interview for Engineers & Managers

0% completed

Key-value Store

Content Delivery Network (CDN)

Sequencer

Distributed Monitoring

Monitor Server-side Errors

Monitor Client-side Errors

Distributed Cache

Distributed Messaging Queue

Pub-sub

Rate Limiter

Blob Store

Distributed Search

Distributed Logging

Distributed Task Scheduler

Sharded Counters

Concluding the Building Blocks Discussion

Design YouTube

Design Quora

Design Google Maps

Design a Proximity Service / Yelp

Design Uber

Design Twitter

System Design: Twitter

Requirements of Twitter's Design

High-level Design of Twitter

Detailed Design of Twitter

Client-side Load Balancer for Twitter

Quiz on Twitter's Design

Design Newsfeed System

Design Instagram

Design a URL Shortening Service / TinyURL

Design a Web Crawler

Design WhatsApp

Design Typeahead Suggestion

Design a Collaborative Document Editing Service / Google Docs

Spectacular Failures

Concluding Remarks

Course Certificate

Mark Course as 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. 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
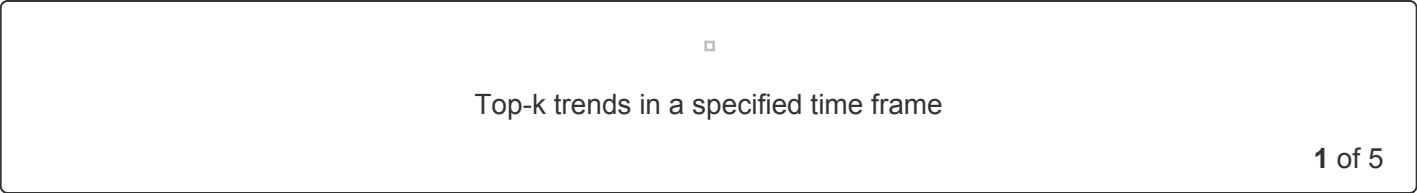
Timeline: Twitter shows two types of timelines: home and user timelines. Here, we'll discuss the home timeline that displays a stream of Tweets posted by the followed accounts. The decision to show Top-k Tweets in the timeline includes followed accounts Tweets and Tweets that are liked or Retweeted by 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 <u>deduping</u> 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

High-level Design of Twitter

Next

Client-side Load Balancer for Twitter

Mark as Completed

Report an Issue