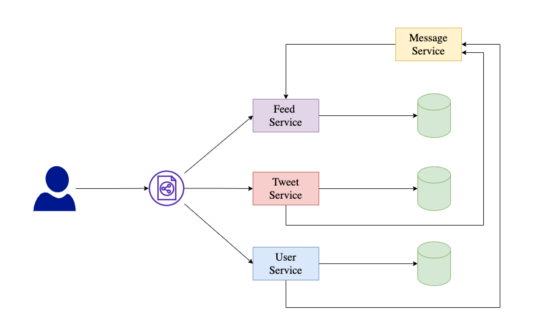
**Essential Conditions**

* User is able to tweet
* Push alerts and tweets can be sent by a service.
* The user can view both the home timeline and the user timeline.
* Keyword searches are possible for users.
* The service should be very accessible.

**High-Level Design**

To handle client requests, the system needs a large number of application servers with load balancers in front of them to distribute traffic. On the backend, it's necessary to have database servers that can store data efficiently. In the event of a service similar to Twitter, photos and videos must also be stored in the object file storage.

The following describes the overall high-level architecture of our platform:

The platform offers a variety of services, each of which serves a specific purpose. To cover all of the use cases in scope, each of these services provides a set of APIs that the client app or other services may use. Furthermore, each of these services has its specialized data store for storing its data. When a service receives a request, such as a new tweet or a request to follow a user, it validates the request and stores the data in its data store. The service also sends a message to the messaging service in the form of an event, which other services can consume and use to change their state.

**Capacity Estimation**

* Twitter roughly has 100 million active users and handles 500 million tweets per day on average.
* Twitter is more read-heavy than write-heavy, we should optimize for fast reads of tweets.
* It has 250 billion read requests per month and roughly 10 billion searches per month.

**Let's assume:**

Size per tweet = 10 KB Size of tweet per month = 10 KB \* 500 million \* 30 = 0.15 PB(storage)

Read requests per second = 100k (250B\*400/1B)

Tweets per second = 6,000 (15B\*400/1B)

**Data model and Storage**

For the data storage, both SQL and NoSQL alternatives could be taken into consideration. We will choose a NoSQL solution for the data storage because of the scalability concerns with SQL solutions and the large number of users to support. For additional things like tweets, comments, and likes, we can use a big column data store like Cassandra.

To store entities like the user and their followers, we might use a graph-based data storage solution like NeoJ or Cassandra.

**Detailed Components Design**

The following are the main system elements of our platform:

**User Service**: Offers an API for controlling users and interactions, such when one user follows or unfollows another. It is possible to establish a distinct User Follower Service in place of including the follow/unfollow use case as part of the User Service.

**Tweet Service** : In order to create and save tweets in the data store, the Tweet Service offers an API. Additionally, it controls a user's capacity to like and comment on tweets. The Tweet Service can optionally be further divided into the Tweet Comments and Tweet Like Services.

**Feed Service**: The feed service determines which tweets will show up on the user's timeline. Additionally, this service features an API that provides tweets so that the timeline can be displayed.

The pagination feature of this API also enables users to access tweets in chunks as they peruse their timeline.

**Timeline generation**: As an easy illustration, the timeline should display all of the most recent posts from followers. Users with a big number of followers will experience very delayed timeline creation because the algorithm will need to search through, combine, and rank a lot of tweets. As a result, the system should regenerate the timeline rather than producing it when the user loads the page.

Users' timelines must be continuously created and stored on specialised servers. Whenever users load the app, the system could offer the pre-generated timeline from the cache. Using this method, users' timelines are regularly compiled and delivered to them whenever they need it rather than being created on demand.

**Timeline Updates**: If the system treats all users the same, a user's timeline generation interval will be considerable, and he will experience a significant delay in receiving new postings. Prioritizing users with new updates is one approach to address this. New tweets are added to the message queue, then picked up by timeline generator services, which re-generates the timeline for all followers.

**For publishing new posts to users, we can use these approaches:**

Pull model: Customers can manually request data whenever they need it or on a regular basis. This method has the drawback that new data is not shown until the client submits a pull request. Furthermore, because no new data has been contributed, the majority of pull requests will return an empty answer, wasting resources.

Push model: When a person tweets, the system might immediately notify all of their followers. When a user has millions of followers, the server must simultaneously send updates to a big number of people, which could be a disadvantage of this strategy.

Hybrid: Pull and push concepts can be combined to form a hybrid system. Only those with a few hundred (or thousands) of followers get access to the system's information. We might give the updates to admirers of celebs.

**Optimization**

We'll examine a number of optimization techniques for scaling our architecture for performance, scalability, redundancy, and other non-functional requirements in the following section.

**Data Sharding**

It is hard to preserve all of the data from a datastore on a single computer due to the large number of users and tweets; therefore, it is best to begin specifying a partitioning plan right once. The partition approach helps divide data for one service into multiple divisions. A distinct cluster server receives a replication of every partition. Data sharing is a technique for disseminating information based on predetermined standards. By requiring a data node to seek for a document within a smaller group of nodes, data sharding increases a data node's speed. Let's examine some of the various ways we may divide our data.

Sharding based on

UserlD: We link each user to a server based on UserlD, which keeps all of the user's tweets, likes, and followers, among other things. When users (such as celebrities) are popular, this strategy fails, and we end up with more data and access on a subset of servers than on others.

Sharding based on TweetID: We map each tweet to a server that holds the tweet information using TweetlD. We must query all servers to find tweets, and each server will return a collection of tweets. This technique overcomes the problem of hot users, but it increases latency because all servers must be queried.

**Data Caching**

Caching is a critical feature that will aid in the scaling of our system. It improves performance by returning precomputed data from the cache rather than computing data for each request.

The Feed Service is the appropriate component for introducing caching and improving system speed. To save the chronology for each user, we may use any caching solution such as Redis, Memcache, or others. Because the cached data on Twitter is primarily brief tweets, it may be feasible to store the whole user timeline or its majority. Finally, when the user browses the feed and adds additional tweets to the cache, we can preemptively request the next batch of feed data for that user from the store and cache it, decreasing latency and enhancing the user experience.

**Load Balancing**

To uniformly distribute the load (requests) from various geographic regions across several data centres, we may employ DNS load balancing. A load balancer will be placed in front of each service to distribute incoming requests to different service nodes according to capacity projections. It is possible to use load balancing techniques like round-robin, least connection, and others. Because our services are stateless, we can also add or remove nodes from the cluster without losing any state.