Designing Facebook’s Newsfeed

Let's design Facebook's Newsfeed, which would contain posts, photos, videos, and status updates from all the people and pages a user follows.

Similar Services: Twitter Newsfeed, Instagram Newsfeed, Quora Newsfeed
Difficulty Level: Hard

We'll cover the following

1. What is Facebook’s newsfeed?#

A Newsfeed is the constantly updating list of stories in the middle of Facebook’s homepage. It includes status updates, photos, videos, links, app activity, and ‘likes’ from people, pages, and groups that a user follows on Facebook. In other words, it is a compilation of a complete scrollable version of your friends’ and your life story from photos, videos, locations, status updates, and other activities.

For any social media site you design - Twitter, Instagram, or Facebook - you will need some newsfeed system to display updates from friends and followers.

2. Requirements and Goals of the System#

Let's design a newsfeed for Facebook with the following requirements:

Functional requirements:

1. Newsfeed will be generated based on the posts from the people, pages, and groups that a user follows.
2. A user may have many friends and follow a large number of pages/groups.
3. Feeds may contain images, videos, or just text.
4. Our service should support appending new posts as they arrive to the newsfeed for all active users.

Non-functional requirements:

1. Our system should be able to generate any user's newsfeed in real-time - maximum latency seen by the end user would be 2s.
2. A post shouldn't take more than 5s to make it to a user's feed assuming a new newsfeed request comes in.

3. Capacity Estimation and Constraints#

Let's assume on average a user has 300 friends and follows 200 pages.

Traffic estimates: Let's assume 300M daily active users with each user fetching their timeline an average of five times a day. This will result in 1.5B newsfeed requests per day or approximately 17,500 requests per second.

Storage estimates: On average, let's assume we need to have around 500 posts in every user's feed that we want to keep in memory for a quick fetch. Let's also assume that on average each post would be 1KB in size. This would mean that we need to store roughly 500KB of data per user. To store all this data for all the active users we would need 150TB of memory. If a server can hold 100GB we would need around 1500 machines to keep the top 500 posts in memory for all active users.

4. System APIs#

> 💡 Once we have finalized the requirements, it's always a good idea to define the system APIs. This should explicitly state what is expected from the system.

We can have SOAP or REST APIs to expose the functionality of our service. The following could be the definition of the API for getting the newsfeed:

```
getUserFeed(api_dev_key, user_id, since_id, count, max_id, exclude_replies)
```

Parameters:
api_dev_key (string): The API developer key of a registered can be used to, among other things, throttle users based on their allocated quota.
user_id (number): The ID of the user for whom the system will generate the newsfeed.
since_id (number): Optional; returns results with an ID higher than (that is, more recent than) the specified ID.
count (number): Optional; specifies the number of feed items to try and retrieve up to a maximum of 200 per distinct request.
max_id (number): Optional; returns results with an ID less than (that is, older than) or equal to the specified ID.
exclude_replies(boolean): Optional; this parameter will prevent replies from appearing in the returned timeline.

Returns: (JSON) Returns a JSON object containing a list of feed items.

5. Database Design#

There are three primary objects: User, Entity (e.g. page, group, etc.), and FeedItem (or Post). Here are some observations about the relationships between these entities:

- A User can follow other entities and can become friends with other users.
- Both users and entities can post FeedItems which can contain text, images, or videos.

- Each FeedItem will have a UserID which will point to the User who created it. For simplicity, let's assume that only users can create feed items, although, on Facebook Pages can post feed item too.
- Each FeedItem can optionally have an EntityID pointing to the page or the group where that post was created.

If we are using a relational database, we would need to model two relations: User-Entity relation and FeedItem-Media relation. Since each user can be friends with many people and follow a lot of entities, we can store this relation in a separate table. The "Type" column in "UserFollow" identifies if the entity being followed is a User or Entity. Similarly, we can have a table for FeedMedia relation.

User	
PK	**UserID: int**
	Name: varchar(20)
	Email: varchar(32)
	DateOfBirth: datetime
	CreationDate: datetime
	LastLogin: datetime

Entity	
PK	**EntityID: int**
	Name: varchar(20)
	Type: tinyint
	Description: varchar(512)
	CreationDate: datetime
	Category: smallint
	Phone: varchar(12)
	Email: varchar(20)

UserFollow	
PK	**UserID: int** **EntityOrFriendID: int**
	Type: tinyint

FeedItem	
PK	**FeedItemID: int**
	UserID: int
	Contents: varchar(256)
	EntityID: int
	LocationLatitude: int
	LocationLongitude: int
	CreationDate: datetime
	NumLikes: int

FeedMedia	
PK	**FeedItemID: int** **MediaID: int**

Media	
PK	**MediaID: int**
	Type: smallint
	Description: varchar(256)
	Path: varchar(256)
	LocationLatitude: int
	LocationLongitude: int
	CreationDate: datetime

6. High Level System Design#

At a high level this problem can be divided into two parts:

Feed generation: Newsfeed is generated from the posts (or feed items) of users and entities (pages and groups) that a user follows. So, whenever our system receives a request to generate the feed for a user (say Jane), we will perform the following steps:

1. Retrieve IDs of all users and entities that Jane follows.
2. Retrieve latest, most popular and relevant posts for those IDs. These are the potential posts that we can show in Jane's newsfeed.
3. Rank these posts based on the relevance to Jane. This represents Jane's current feed.
4. Store this feed in the cache and return top posts (say 20) to be rendered on Jane's feed.
5. On the front-end, when Jane reaches the end of her current feed, she can fetch the next 20 posts from the server and so on.

One thing to notice here is that we generated the feed once and stored it in the cache. What about new incoming posts from people that Jane follows? If Jane is online, we should have a mechanism to rank and add those new posts to her feed. We can periodically (say every five minutes) perform the above steps to rank and add the newer posts to her feed. Jane can then be notified that there are newer items in her feed that she can fetch.

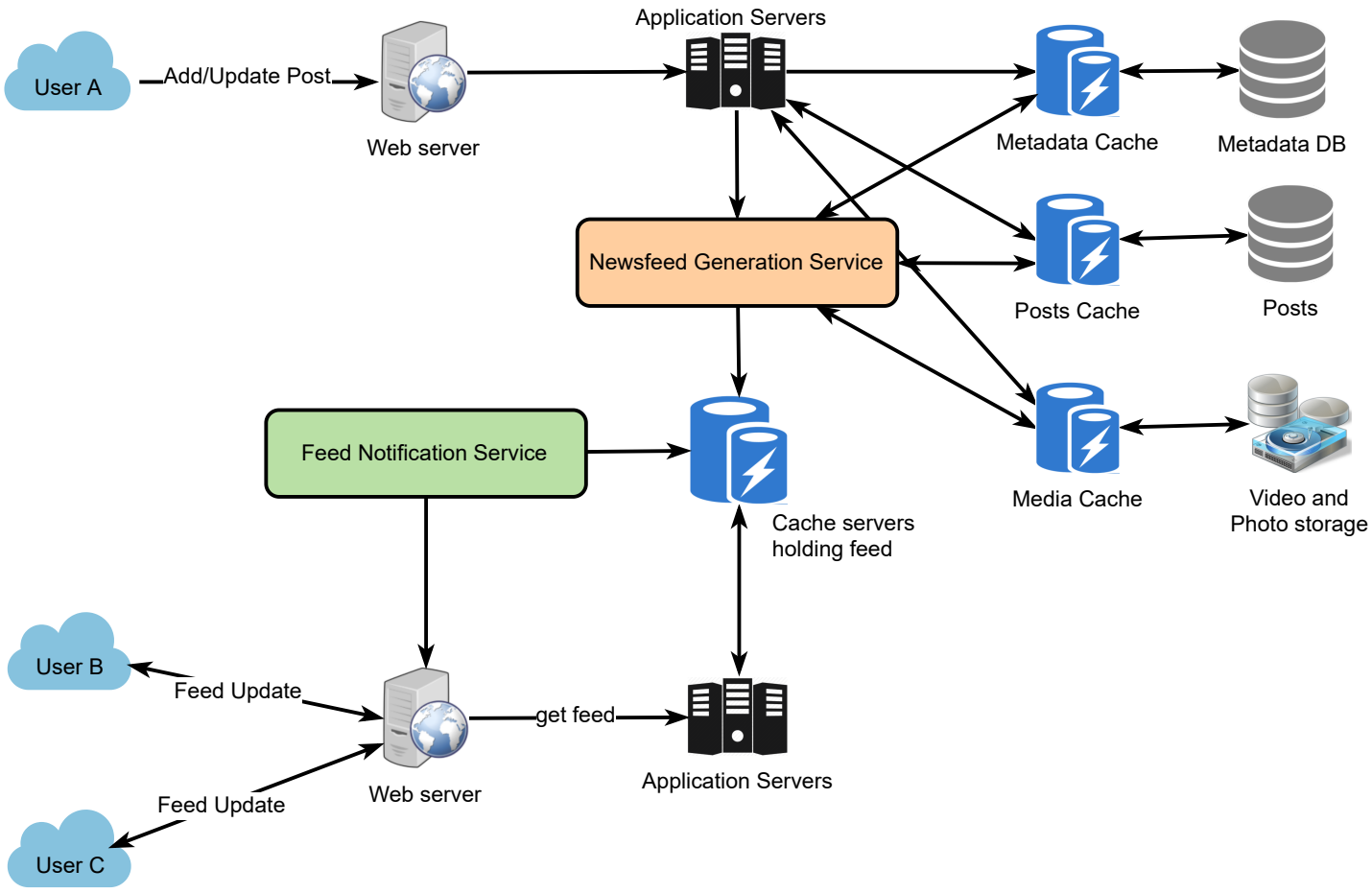
Feed publishing: Whenever Jane loads her newsfeed page, she has to request and pull feed items from the server. When she reaches the end of her current feed, she can pull more data from the server. For newer items either the server can notify Jane and then she can pull, or the server can push, these new posts. We will discuss these options in detail later.

At a high level, we will need following components in our Newsfeed service:

1. **Web servers:** To maintain a connection with the user. This connection will be used to transfer data between the user and the server.
2. **Application server:** To execute the workflows of storing new posts in the database servers. We will also need some application servers to retrieve and to push the newsfeed to the end user.
3. **Metadata database and cache:** To store the metadata about Users, Pages, and Groups.

4. **Posts database and cache:** To store metadata about posts and their contents.
5. **Video and photo storage, and cache:** Blob storage, to store all the media included in the posts.
6. **Newsfeed generation service:** To gather and rank all the relevant posts for a user to generate newsfeed and store in the cache. This service will also receive live updates and will add these newer feed items to any user's timeline.
7. **Feed notification service:** To notify the user that there are newer items available for their newsfeed.

Following is the high-level architecture diagram of our system. User B and C are following User A.



Facebook Newsfeed Architecture

7. Detailed Component Design#

Let's discuss different components of our system in detail.

a. Feed generation

Let's take the simple case of the newsfeed generation service fetching most recent posts from all the users and entities that Jane follows; the query would look like this:

```
(SELECT FeedItemID FROM FeedItem WHERE UserID in (
    SELECT EntityOrFriendID FROM UserFollow WHERE UserID = <current_user_id> a
nd type = 0(user))
)
UNION
(SELECT FeedItemID FROM FeedItem WHERE EntityID in (
    SELECT EntityOrFriendID FROM UserFollow WHERE UserID = <current_user_id> a
nd type = 1(entity))
)
ORDER BY CreationDate DESC
LIMIT 100
```

Here are issues with this design for the feed generation service:

1. Crazy slow for users with a lot of friends/follows as we have to perform sorting/merging/ranking of a huge number of posts.
2. We generate the timeline when a user loads their page. This would be quite slow and have a high latency.
3. For live updates, each status update will result in feed updates for all followers. This could result in high backlogs in our Newsfeed Generation Service.
4. For live updates, the server pushing (or notifying about) newer posts to users could lead to very heavy loads, especially for people or pages that have a lot of followers. To improve the efficiency, we can pre-generate the timeline and store it in a memory.

Offline generation for newsfeed: We can have dedicated servers that are continuously generating users' newsfeed and storing them in memory. So, whenever a user requests for the new posts for their feed, we can simply serve it from the pre-generated, stored location. Using this scheme, user's newsfeed is not compiled on load, but rather on a regular basis and returned to users whenever they request for it.

Whenever these servers need to generate the feed for a user, they will first query to see what was the last time the feed was generated for that user. Then, new feed data would be generated from that time onwards. We can store this data in a hash table where the "key" would be UserID and "value" would be a STRUCT like this:

```
Struct {
    LinkedHashMap<FeedItemID, FeedItem> feedItems;
    DateTime lastGenerated;
}
```

We can store FeedItemIDs in a data structure similar to Linked HashMap (https://docs.oracle.com/javase/7/docs/api/java/util/LinkedHashMap.html) or TreeMap (https://docs.oracle.com/javase/6/docs/api/java/util/TreeMap.html), which can allow us to not only jump to any feed item but also iterate through the map easily. Whenever users want to fetch more feed items, they can send the last FeedItemID they currently see in their newsfeed, we can then jump to that FeedItemID in our hash-map and return next batch/page of feed items from there.

How many feed items should we store in memory for a user's feed? Initially, we can decide to store 500 feed items per user, but this number can be adjusted later based on the usage pattern. For example, if we assume that one page of a user's feed has 20 posts and most of the users never browse more than ten pages of their feed, we can decide to store only 200 posts per user. For any user who wants to see more posts (more than what is stored in memory), we can always query backend servers.

Should we generate (and keep in memory) newsfeeds for all users? There will be a lot of users that don't log-in frequently. Here are a few things we can do to handle this; 1) a more straightforward approach could be, to use an LRU based cache that can remove users from memory that haven't accessed their newsfeed for a long time 2) a smarter solution can figure out the login pattern of users to pre-generate their newsfeed, e.g., at what time of the day a user is active and which days of the week does a user access their newsfeed? etc.

Let's now discuss some solutions to our "live updates" problems in the following section.

b. Feed publishing

The process of pushing a post to all the followers is called fanout. By analogy, the push approach is called fanout-on-write, while the pull approach is called fanout-on-load. Let's discuss different options for publishing feed data to users.

1. **"Pull" model or Fan-out-on-load:** This method involves keeping all the recent feed data in memory so that users can pull it from the server whenever they need it. Clients can pull the feed data on a regular basis or manually whenever they need it. Possible problems with this approach are a) New data might not be shown to the users until they issue a pull request, b) It's hard to find the right pull cadence, as most of the time pull requests will result in an empty response if there is no new data, causing waste of resources.

2. **"Push" model or Fan-out-on-write:** For a push system, once a user has published a post, we can immediately push this post to all the followers. The advantage is that when fetching feed you don't need to go through your friend's list and get feeds for each of them. It significantly reduces read operations. To efficiently handle this, users have to maintain a Long Poll (https://en.wikipedia.org/wiki/Push_technology#Long_polling) request with the server for receiving the updates. A possible problem with this approach is that when a user has millions of followers (a celebrity-user) the server has to push updates to a lot of people.

3. **Hybrid:** An alternate method to handle feed data could be to use a hybrid approach, i.e., to do a combination of fan-out-on-write and fan-out-on-load. Specifically, we can stop pushing posts from users with a high number of followers (a celebrity user) and only push data for those users who have a few hundred (or thousand) followers. For celebrity users, we can let the followers pull the updates. Since the push operation can be extremely costly for users who have a lot of friends or followers, by disabling fanout for them, we can save a huge number of resources. Another alternate approach could be that, once a user publishes a post, we can limit the fanout to only her online friends. Also, to get benefits from both the approaches, a combination of 'push to notify' and 'pull for serving' end-users is a great way to go. Purely a push or pull model is less versatile.

How many feed items can we return to the client in each request? We should have a maximum limit for the number of items a user can fetch in one request (say 20). But, we should let the client specify how many feed items they want with each request as the user may like to fetch a different number of posts depending on the device (mobile vs. desktop).

Should we always notify users if there are new posts available for their newsfeed? It could be useful for users to get notified whenever new data is available. However, on mobile devices, where data usage is relatively expensive, it can consume unnecessary bandwidth. Hence, at least for mobile devices, we can choose not to push data, instead, let users “Pull to Refresh” to get new posts.

8. Feed Ranking#

The most straightforward way to rank posts in a newsfeed is by the creation time of the posts, but today’s ranking algorithms are doing a lot more than that to ensure “important” posts are ranked higher. The high-level idea of ranking is first to select key “signals” that make a post important and then to find out how to combine them to calculate a final ranking score.

More specifically, we can select features that are relevant to the importance of any feed item, e.g., number of likes, comments, shares, time of the update, whether the post has images/videos, etc., and then, a score can be calculated using these features. This is generally enough for a simple ranking system. A better ranking system can significantly improve itself by constantly evaluating if we are making progress in user stickiness, retention, ads revenue, etc.

9. Data Partitioning#

a. Sharding posts and metadata

Since we have a huge number of new posts every day and our read load is extremely high too, we need to distribute our data onto multiple machines such that we can read/write it efficiently. For sharding our databases that are storing posts and their metadata, we can have a similar design as discussed under Designing Twitter (https://www.educative.io/collection/page/5668639101419520/5649050225344512/5741031244955648/).

b. Sharding feed data

For feed data, which is being stored in memory, we can partition it based on UserID. We can try storing all the data of a user on one server. When storing, we can pass the UserID to our hash function that will map the user to a cache server where we will store the user's feed objects. Also, for any given user, since we don't expect to store more than 500 FeedItemIDs, we will not run into a scenario where feed data for a user doesn't fit on a single server. To get the feed of a user, we would always have to query only one server. For future growth and replication, we must use Consistent Hashing (https://www.educative.io/collection/page/5668639101419520/5649050225344512/5709068098338816).

Back

Designing a Web Crawler

Next

Designing Yelp or Nearby Friends

Mark as Completed

Report an Issue