1: CDN (Content Delivery Network)

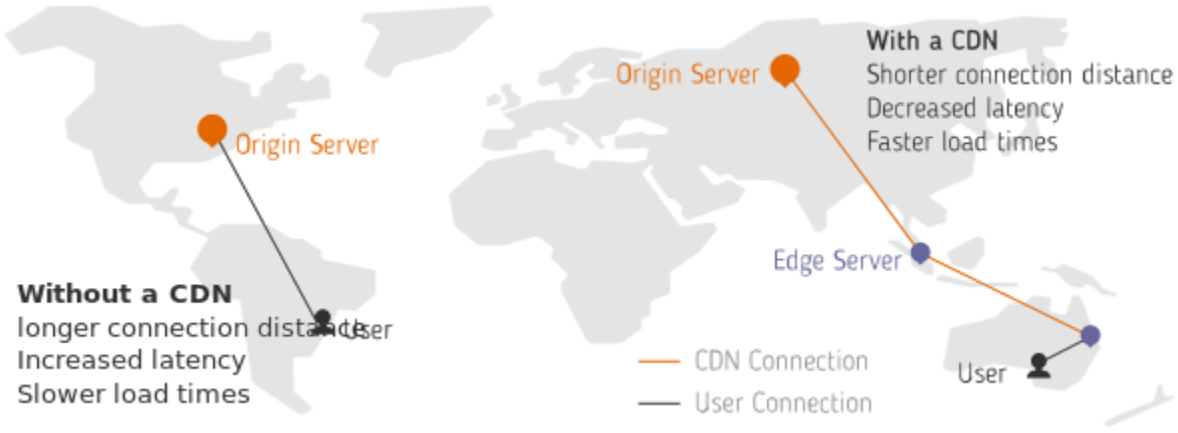
I will discuss fetching images/multimedia from the website (like **del.icio.us**). You have the browser in some region(say India), and you must fetch the files from the servers located in another region(say the US). When you try to access from your browser, a request is made to the load balancer, and then it goes to the application server and requests files from the file storage. You know that transferring files and other data will be fast for the machines in the same region. But it can take time for machines located on different continents.

From the website perspective, users worldwide should have a good experience, and these separate regions act as a hindrance. **So what's the solution?**

The solution for the problem is CDN, Continent Delivery Network. Examples of CDN are companies like

- Akamai
- Cloudflare
- CloudFront by Amazon
- Fastly

These companies' primary job is to have machines worldwide, in every region. They store your data, distribute it to all the regions, and provide different CDN links to access data in a particular region. Suppose you are requesting data from the US region. Obviously, you can receive the HTML part/ code part quickly since it is much smaller than the multimedia images. For multimedia, you will get CDN links to files of your nearest region. Accessing these files from the nearest region happens at a much larger pace. Also, you pay per use for using these CDN services.



One question you might think is **how your machine talks to the nearest region only**(gets its IP, not of some machine located in another region), when CDN has links for all the regions. Well, this happens in two ways:

1. A lot of ISP have CDN integrations. Tight coupling with them helps in giving access to the nearest IP address. For example, Netflix's CDN does that.
2. Anycast (https://www.cloudflare.com/en-gb/learning/cdn/glossary/anycast-network/)

This CDN process to get information from the nearest machine is also a form of caching.

2. Cache eviction

There are various eviction strategies to remove data from the cache to make space for new writes. Some of them are:

- FIFO (First In, First Out)
- LRU (Least Recently Used)
- LIFO (Last In, First Out)
- MRU (Most Recently Used)

The eviction strategy must be chosen based on the data that is more likely to be accessed. The caching strategy should be designed in such a way that you have a lot of cache hits than a cache miss.

3. Caching Metadata - Global Caching

Ranklist Discussion: Let's take an example of the rank list in a contest with immense traffic. During the contest, people might be on the problem list page, reading a problem, or on the rank list page (looking for the ranks). If scores for the participants are frequently updated, computing the rank list becomes an expensive process (sorting and showing the rank list). Whenever a person wants the rank list, it is fetched from DB. This causes a lot of load on the database.

The solution can be computing the rank list periodically and caching it somewhere for a particular period. Copy of static rank list gets generated after a fixed time (say one minute) and cached. It reduces the load on DB significantly.

Storing the rank list in the local server will be less effective since there will be many servers, and every minute cache miss may occur for every server. A much better approach is to store the rank list in the global cache shared by all app servers. Therefore there will be only one cache miss every minute. **Here global caching performs better than local caching**. Redis can be used for the purpose.

Redis: Redis is one of the most popular caching mechanisms used everywhere. It is a single-threaded key-value store. The values which Redis supports are:

- String

- Integer
- List
- Set
- Sorted_set

Redis

Key	Value
"Rahul_score"	500

The main scenarios where global caching is used are:

1. Caching something that is queried often
2. Storing derived information, which might be expensive to compute on DB.

And we can use Redis for either of the cases mentioned above to store the most relevant information. It is used to decrease data latency and increase throughput.

> To get a sense of Redis and have some hands-on you can visit: https://try.redis.io/
> You can also check the following:
> - Set in Radis https://redis.io/docs/data-types/sets/
> - Sorted set in Radis:https://redis.io/docs/data-types/sorted-sets/

Facebooks's newsfeed

How Facebook computes its newsfeed?
Let's do another case study. What if we were supposed to build the system that computes news feed for Facebook. Let's first discuss the basic architecture of Facebook.

Facebook has a lot of users, and each user has a bunch of attributes. Let's first discuss the schema of Facebook if all information could fit on a single machine SQL DB. You can for now assume that we care about the most basic v0 version of Facebook which has no concept of pages/groups/likes/comments, etc.

User

id	name	email	relationship_st	last_active

Users also have friends, and users can make posts on Facebook.

User_friends

user_id	friend_id

Posts

id	text	user_id	timestamp	....

And there are two kinds of pages a user sees on Facebook:

- **Newsfeed**: posts made by friends of the user.
- **Profile page**: it has information about a particular user and his posts.

If all the related information (user info, user_friend info, and posts info) could fit on a single machine, computing the newsfeed and profile page would become easy.

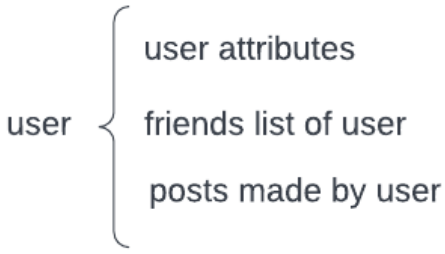
Newsfeed	Profile Page
Posts made by friends of the user. We can use the query: SELECT * FROM User_friends a JOIN Posts b ON a.user_id = <user_id> AND b.user_id = a.friend_id AND b.timespamp < NOW - 30 days LIMIT x OFFSET y	Posts made by the user. We can use the query: SELECT * FROM Posts WHERE user_id = <user_id> LIMIT x OFFSET y

In the above query “LIMIT x OFFSET y” is done to paginate results as there could be a lot of matching entries.

Here, the assumption is made that all the information fits in the single machine, but this is not the case generally. Therefore information needs to be **sharded** between the machines.

So, **what will be the sharding key?**

If we use user_id as sharding key, that means for a given user, all their attributes, their friend list and posts made by them become one entity and would be on one machine.



However, posts made by friends of the user will be on the machine assigned to the friend user_id [Not guaranteed to be on the same machine].

If you come and ask for information to be fetched to show the profile page of user_id X, that is simple. I go to the machine for X and get user_attributed, friend list and posts made by X (paginated).

However, what happens when I ask for the news feed for user X. For the news feed, I need posts made by friends of X. If I go to the machine for X, that is guaranteed to have the list of friends of X, but not guaranteed to have posts made by those friends, as those friends could be assigned to other machines. That could become extremely time consuming process.

How can we optimize newsfeed fetch?

One might think that caching **user → newsfeed** is a good option. But it has the following drawbacks:

1. More Storage required
2. Fan out update: Have to update posts in every friend's list everytime a single post is made (1000+ writes for every single post made assuming 1000+ avg friends).
3. Changing newsfeed algorithms becomes hard

Let's estimate what is the amount of post we generate every day. Posts made by users are far less than the number of active users (80-20-1 rule). Only 1% will do posts, 80% reading, and 20% will interact.

Lets do some math.
FB MAU - 1 Billion
FB DAU - 500 million.
People who would write posts = 1% of 500 million = 5 million.
Assuming each person writes 4 posts on average (overestimating), we have roughly 20 million posts everyday.

A post has some text, some metadata (timestamp, poster_id, etc.) and optionally images/videos. Assuming images/videos go in a different storage, what's the space required to store a single post?
Metadata:

- *Poster_id - 8 bytes*
- *Timestamp - 8 bytes*
- *Location_id - 8 bytes*
- *Image / video path (optional) - 24 bytes (estimated).*

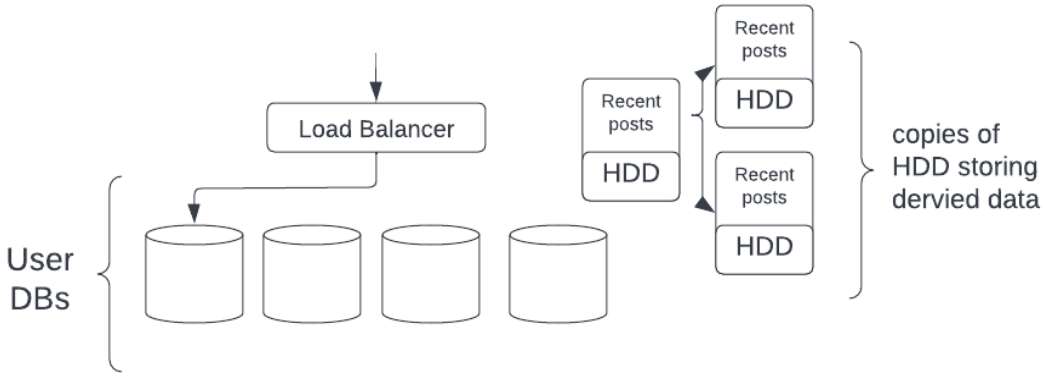
On text, hard to estimate the exact size. Twitter has limit of 140 characters on tweets. Assuming FB posts are slightly longer, lets assume 250 bytes/250 characters on avg. for a post.

So, total size of the post = 250 + 8 + 8 + 8 + 24 = approx 300.

*Total space required for posts generated in a single day = # of posts * size of post*
*= 20 million * 300 bytes = 6 GB approx.*

News feed is supposed to show only recent posts from a friend. You don't expect to see a year old post in your news feed. Let's assume you only need to show posts made in the last 30 days. In that case, you need 6 GB * 30 = 180GB of space to store every post generated in the last 30 days.

Therefore all the recent posts can be stored in a separate database and retrieving becomes easier from the derived data. We can replicate and have multiple copies (of all posts) in a lot of machines to distribute the read traffic on recent posts.



1. Fetch the friend_ids of the user.
2. Select recent posts made by the user's friend: SELECT * FROM all_posts WHERE user_id IN friend_ids LIMIT x OFFSET y

This approach uses much lesser storage and approach than the previous system. Here the cache is stored in a hard disk, not in RAM, but still, this is much faster than getting data from an actual storage system.

We can also delete the older posts from HDD: DELETE * FROM all_posts WHERE timestamp < NOW - 30 days. This will help in better storage management.