System Design - Caching Contd.

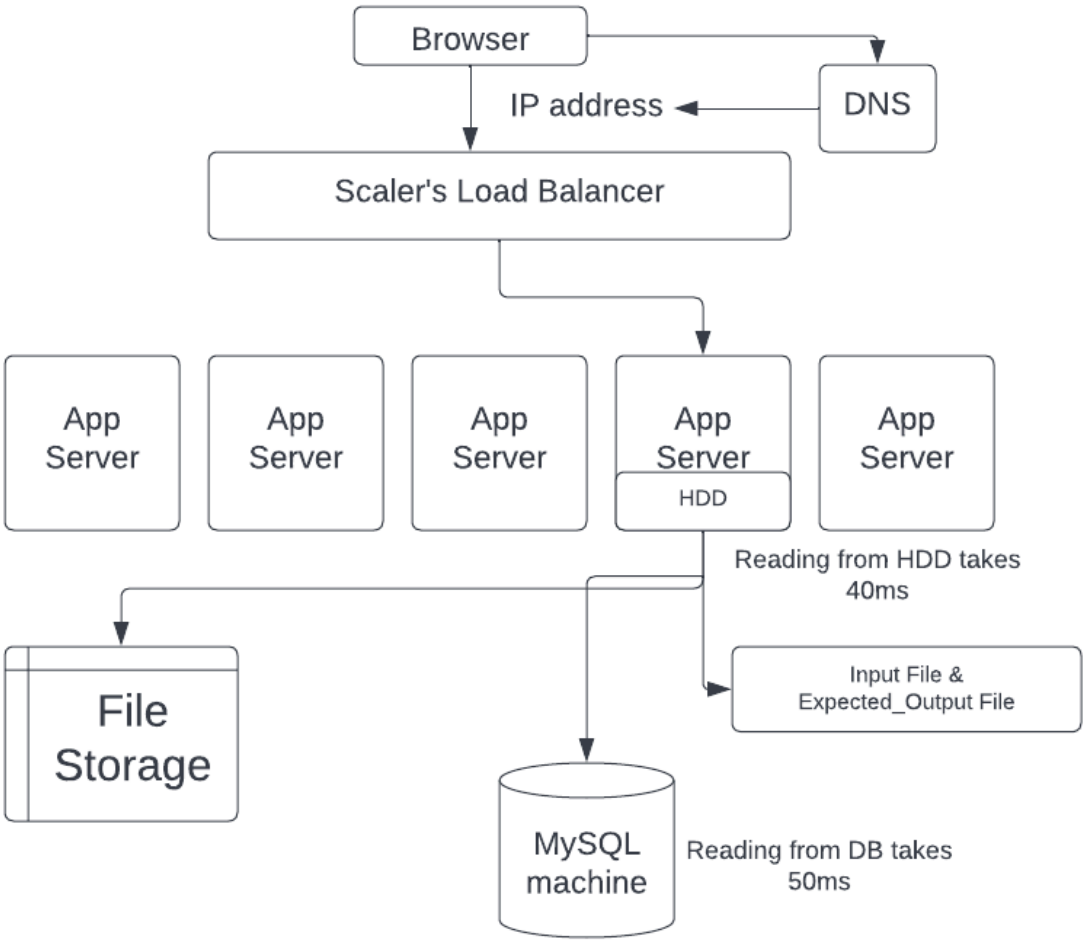
Agenda:
- Local Cache Case Study
- Global Cache → why?
- Global Cache Case Study

In the last class, we discussed how caching could be done at multiple layers: in-browser, using CDN for larger resources, in the application layer, or in the database layer. We initially started with the local caches and ended the class on the case study. The problem statement was:

Consider the case of submitting DSA problems on Scaler; when you submit a problem on Scaler, the browser talks to Scaler's load balancer. And the submission goes to one of the app servers out of many. The app server machine gets the user id, problem id, code, and language. To execute the code, the machine will need the input file and the expected output file for the given problem. The files can be large in size, and it takes time (assumption: around 2 seconds) to fetch files from the **file storage**. This makes code submissions slow.

So, how can you make the process fast?

Assumptions:
1. If the file is present on the machine itself, on the hard disk, then reading the file from the hard disk takes 40ms.
2. Reading from a DB machine (MySQL machine), reading a table or a column (not a file) takes around 50ms.



It can be noted that input files and the expected output file can be changed. The modified changes should be immediately reflected in the code submissions.

Solution

Different approaches to solve the problem can be

TTL

TTL Low: If TTL is very low (say for 1 min), then cache files become invalid on the app server machines after every minute. Hence most of the time, test data won't be available on the machine and is to be fetched from file storage. The number of cache misses will be high for TTL very low.

TTL High: If TTL is very high, then case invalidation happens too late. Say you keep TTL 60 min, and in between the time you change the input & expected files, the changes will not be reflected instantly.

So TTL can be one of the approaches, but it is not a good one. You can choose TTL based on the cache miss rate or cache invalidation rate.

Global Cache

Storing the data in a single machine can also be an option, but there are two problems with this:

1. If storing in memory, the remote machine has limited space and can run out of space quickly because the size of the input-output files is very large.
2. The eviction rate will be very high, and the number of cache misses will be more.

If instead you store it in the hard disk, then there is the issue of transferring huge amount of data on network.

File Metadata

The best approach would be we identify whether the file has changed or not using the metadata for the files.

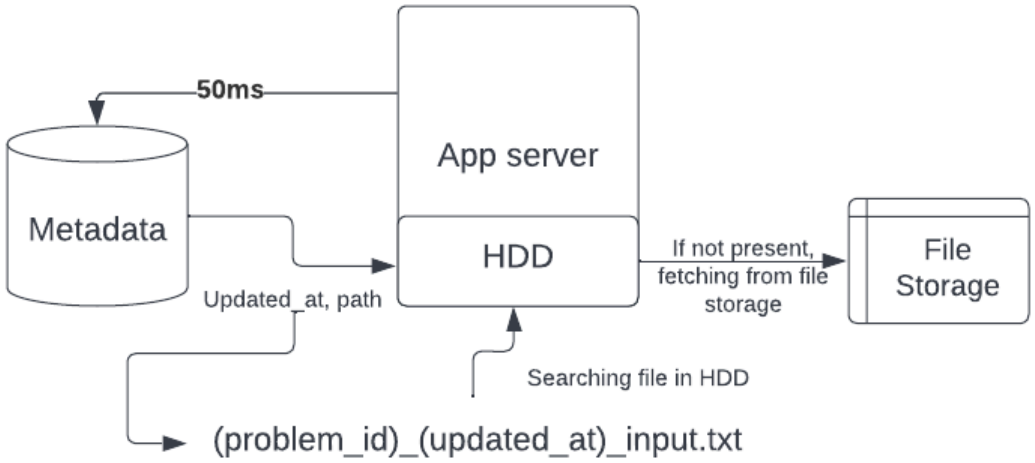
Let's assume in the MySQL database, there exists table problems_test_data. It contains details problem_id, input_filepath, input_file_updated_at, input_file_created_at for input files, and similar details for the output files as well. If a file is updated on the file storage, its metadata will also be updated in the SQL database.

problems_test_data

problem_id	input_filepath	input_file_updated_at	input_file_created_at

Updating meta data of a file in DB when uploading/modifying to file storage.

Now all the files can be cached in the app server with a better approach to constructing file names. The file can be **(problem_id)_(updated_at)_input.txt**



When a submission comes for a problem, we can go to the database (MySQL dB) and get the file path and its last updated time. If a file with problem_id_updated_at_input.txt exists in a machine cache, it is guaranteed that the existing file is the correct one. If the file doesn't exist, then the path can be used to fetch it from the file storage (along with now storing it locally on the machine for the future). Similar things can be done for the output files as well. Here the metadata about the file is used to check whether the file has been changed/updated or not, and this gives us a very clean cache invalidation.

Updating a file,
All cache servers have some files stored if an update is to be done for a file stored in S3. The process looks like this:
For a problem (say for problem_id 200) if an update request comes to modify an input file to a newly uploaded file.

- Upload new input file to file storage (S3). It will give a path (say new_path) for the file stored location.
- Next, MySQL DB has to be updated. The query for it looks like
  UPDATE problem_test data WHERE problem_id = 200 SET inputfile_path = new_path AND inputfile_updated_at = NOW()
- Now, if submission comes and the metadata in DB does not match that of the file existing in the cache, the new file needs to be fetched from the file storage at the location new_path. The returned file will be stored in the HDD of the app server. For the next requests, it will be present on the hard disk already (if not evicted).

It can be noted that every time a submission is made, we have to go to the MySQL DB to fetch all the related information of the problem/user. The information like whether it's already solved, problem score, and user score. It's a better option to fetch the file's metadata simultaneously

while we fetch other details. If solutions pass, the related details have to be updated on DB again.

A separate cache for all machines is better than one single-layer cache.
Here(https://gist.github.com/jboner/2841832) is why.

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

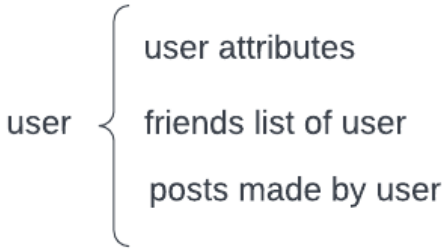
Posts made by friends of the user. We can use the query: SELECT * FROM User_friends a JOIN Posts b ON a.user_id = <user_id> AND b.user_id = a.friend_id AND b.timespamp < NOW - 30 days LIMIT x OFFSET y	Posts made by the user. We can use the query: SELECT * FROM Posts WHERE user_id = <user_id> LIMIT x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

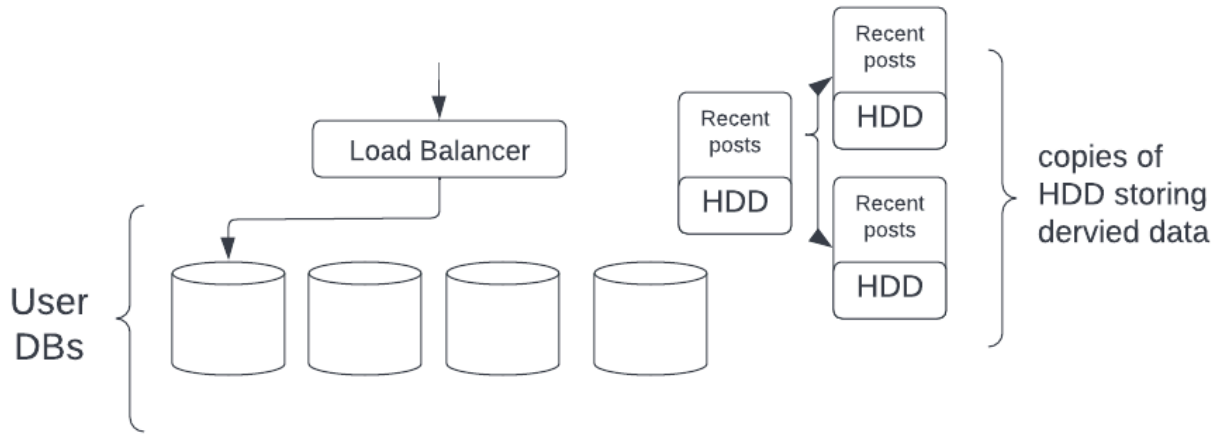
On text, hard to estimate the exact size. Twitter has limit of 140 characters on tweets. Assuming FB posts are slightly longer, lets assume 250 bytes/250 characters on avg. for a post.

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.