ECE 1779 Project 1 Documentation

Group 3

Part 1. Project Abstract

In this project, we implemented a website that allows users to upload photos and retrieve photos with a key. Behind the scenes, a cache for fast data retrieval and a database to store all key-value pairs are built. The cache includes several attributes that can be set by the user, while the database stores these attributes along with additional cache statistics. Figure 1 shows the overview of our system. Detailed workflows will be presented in the part 2.

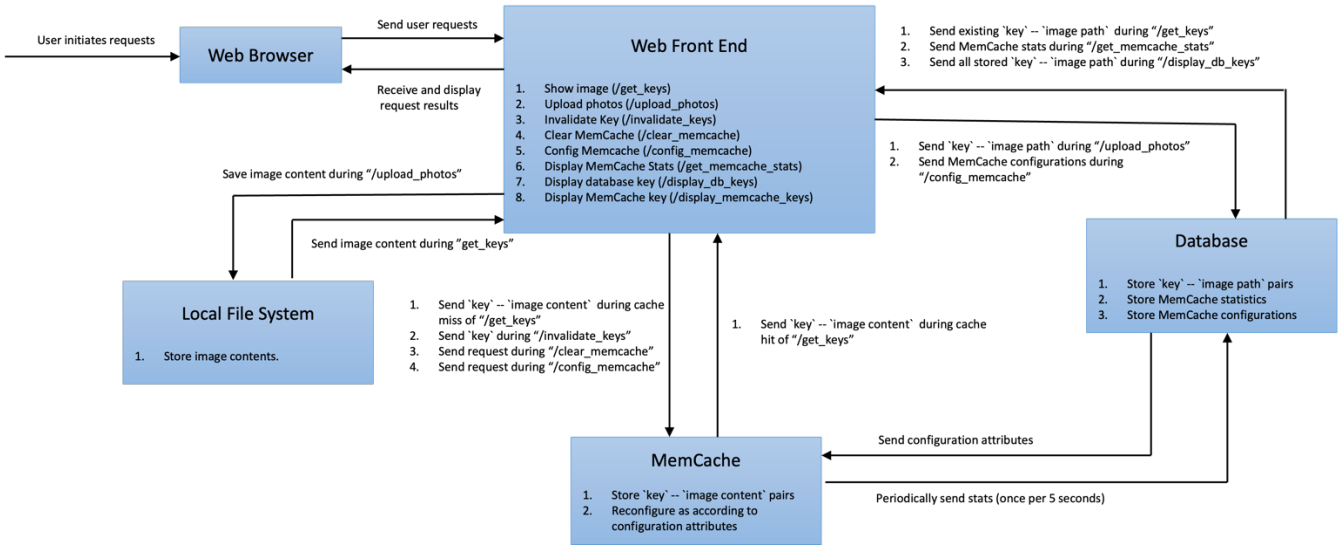


Figure 1. Overall System Architecture

Part 2. Application Components and Functions

1. Web Browser: Initiates user requests and displays received results.

2. Front End: Manages user requests
   a. Upload photos: Allows users to upload a photo along with a key. The key-filename pair is stored in the database while the file itself, i.e., the image content is stored in the local file system. Detailed workflows are shown in figure 2.

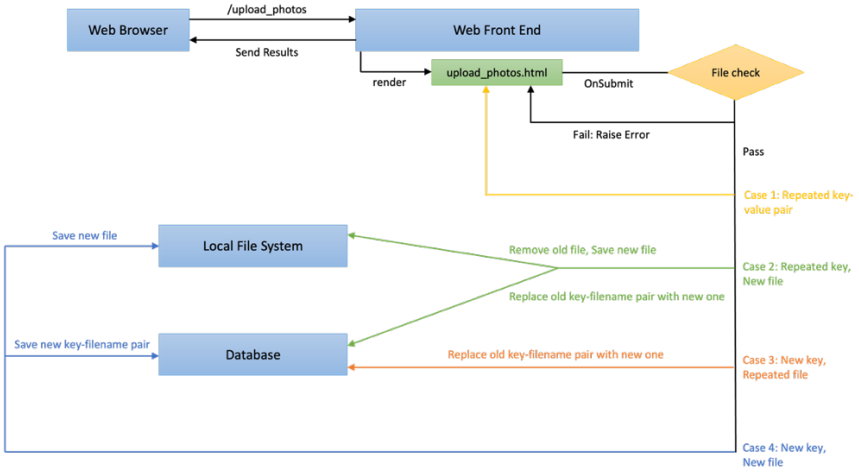


Figure 2. /upload_photo request workflow

   b. Get a photo with key: The full workflow when the front end receives a get photo request is displayed in figure 4. There are three possible cases when a key is an input by the user. Their respective workflows are shown below. Numbers indicate their steps labelled in figure 3.

Cases	Workflow

Key in MemCache	Step 1 → Step 2: Yes → Show image
Key in Database	Step 1 → Step 2: No → Step 3: Yes → Step 4 → Step 5 → Step 6 → Show image
Key not found	Step 1 → Step 2: No → Step 3: No → "Unknown Key!"

Table 1. /get_photos request workflow

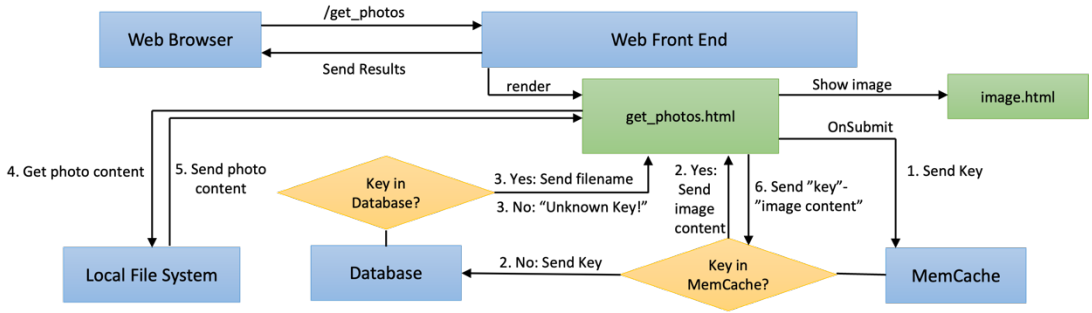


Figure 3. /get_photos request workflow

c. Invalidate key: Front End requests MemCache to drop the key if a current key-image pair exists in MemCache.

d. Clear Memcache: Front End requests MemCache to drop all.

e. Configure Memcache attributes: Two attributes can be set, being the capacity of the cache (in MB) and the replacement policy (Random Replacement (*RR*) / Least Recently Used (*LRU*)). Detailed workflows are shown in figure 4.

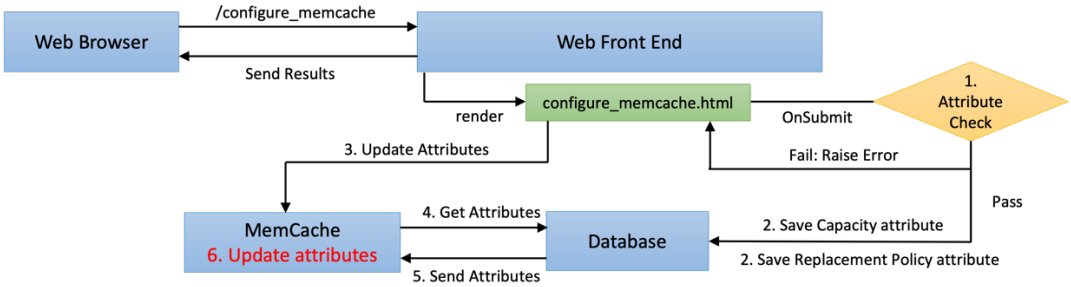


Figure 4. /configure_memcache request workflow

f. Get Memcache stats: Front End requests the database for Memcache stats of the recent 10 minutes. Five stats are being kept track of. We use a child thread to run the statistic tracking concurrently alongside the MemCache flask instance. It updates the stats stored in the database every five seconds.

MemCache Stats	
No_items	The number of items in the MemCache currently.
Total_size	The total size of the MemCache currently.
No_requests	The number of "Get" requests to the MemCache in the recent 10 minutes.
Miss_rate	Cache miss count / No_requests
Hit_rate	Cache hit count / No_requests

Table 2. MemCache Stats

g. Display current database keys: Front End requests the database for all key-filename pairs stored. There shall be a one-to-one relationship between each of these pairs to the image contents stored in the local file system.

h. Display current Memcache keys: Front End request MemCache for all keys stored in MemCache. The order starting from the top represents the least recently used keys down to the most recently used keys.

3. Database: Details about the database will be discussed in the **Database Schema** session

4. MemCache: The MemCache is a memory object cache system. It would store parts of the data in a "key"-"image content" form according to locality. It can improve the performance of the code by reducing the number of times to retrieve data from the database. Therefore, the MemCache can improve the web app's response time.

   Such a "key"-"image content" is put into MemCache whenever a cache miss happens as shown in Table 1 and Figure 3 step 6. An existing key will be discarded either when an invalidate key request or a clear MemCache request is called. If MemCache reaches maximum capacity, it will also discard keys according to the replacement policy. No cache is possible if implemented when setting capacity as 0. There are two types of replacement policies in this web app: Random Replacement (RR) and Least-Recently Used (LRU)

Attributes	
Capacity	Configure the size of MemCache. The default is set to the maximum value of 1000MB.
0	No Cache: For every GET() request, the app will face a cache miss, search database for "key"-"path", and look into local file system with "path" to retrieve image content. However, the workload is tedious and the response time can be slowed down.
Replacement Policy: *RR*	Randomly discard a key when the current capacity is larger than configured capacity. This is the default replacement policy method.
Replacement Policy: *LRU*	Discard the least recently used key when the current capacity is larger than configured capacity. The access order is kept track of using OrderedDict() to ensure correctness.

Table 3. /configure_memcache request workflow

5. Local File System: Stores the content of the images. Various workflows relate to this can be seen in Figure 2.

Part 3. Database Schema

As the ER diagram shows, there are four tables needed in the database schema. Image, Cache, cache_image and cache_config. The cache_config table stores the information of cache configuration information including the id, capacity, and policy of the cache. Therefore, the cache_config table does not have any relationship with other tables. The image table and cache table store the information of the image and cache separately. Besides, the cache_image is the relationship set between the image and cache sets. This relationship is set up just in case there are multiple cache instances.

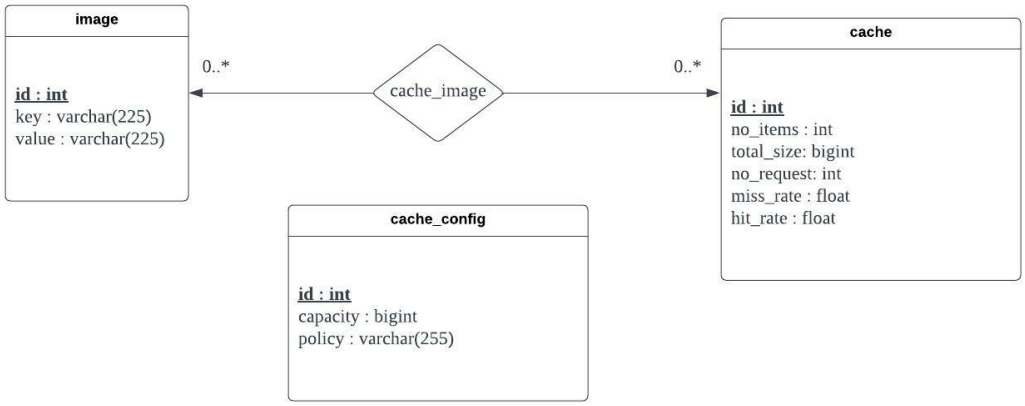


Figure 5 Database ER diagram

Part 4. Reduction to Relationship Schemas:

Strong entity sets:

image(id, key, value)
cache(id, no_items, total_size, no_request, miss_rate, hit_rate)
cache_config(id, capacity, policy)

Relationship sets:

cache_image(image_id, cache_id)

Part 5. Design Decisions

- **Communication between Front End and Cache**

  We used blocking requests for all communications between front end and cache. An alternative method could be to use threading for synchronous processing. This can possibly increase performance when the app is running independent tasks. An example is if the image content is very large, blocking will waste a lot of time when sending the image content from cache to front end during cache hit. However, implementing the synchronous method is tedious, and since most of the communications do not involve such kind of large data, we chose the asynchronous method. In addition, image sizes are rarely that big that it would bottleneck the whole system when blocking.

- **Database**

  We choose to use the MySQL database as the RDBMS of our project. The database is created through the image_db.sql file.

  To develop the website database, there are many alternatives such as ORM (Object-Relational mapping) database, and other RDBMS databases such as SQLite and so on. However, AWS supports MySQL and MySQL is also the requirement of this project. Compared to other DBMS databases such as SQLite, MySQL is more secure due to its authorization server. Therefore, we choose to use MySQL as our database server.

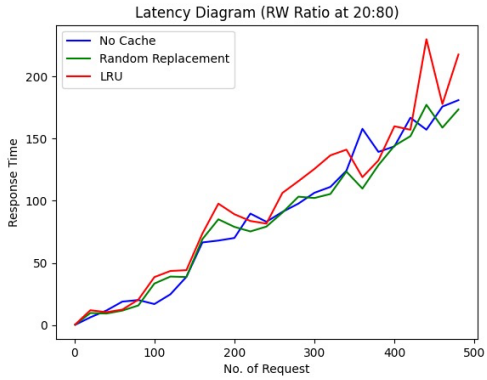
  To facilitate the management and modification of the database, we put all the operations of the database with flask into one python file called db_operations.py.

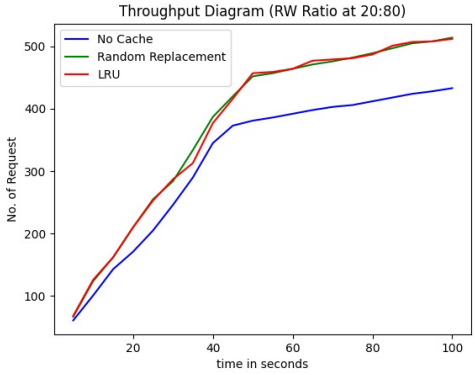
Part 6. Results

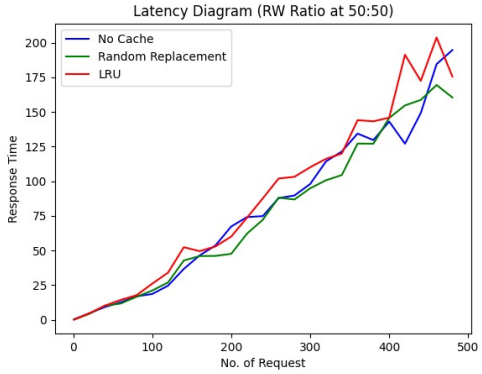
Testing Environments :

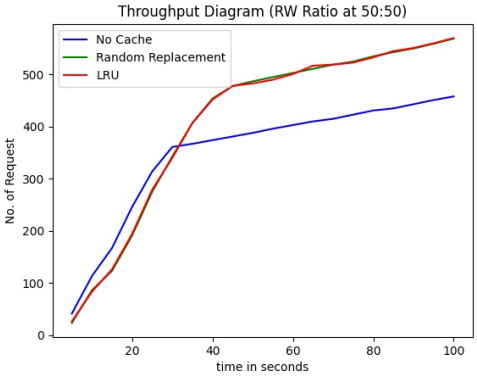
1. Average image size: 100kb per image
2. Cache size during RR and LRU: 30MB
3. R/W testing for latency graph: Finish all writing requests, followed by all reading requests
4. R/W testing for throughput graph: Using 10 requests as a cycle, where the distribution is set according to RW ratio. For example, a RW ratio of 20:80 means we perform 2 reading requests followed by 8 writing requests and recur.

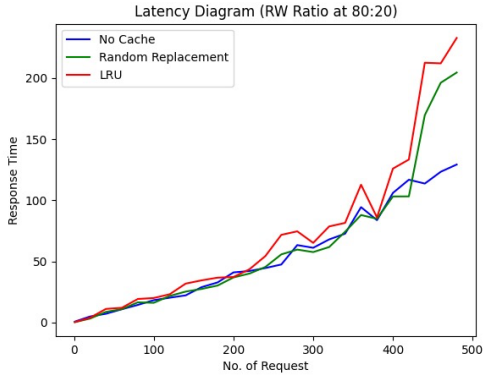
5. Reading requests is done by randomly choosing an existing key as get_key.

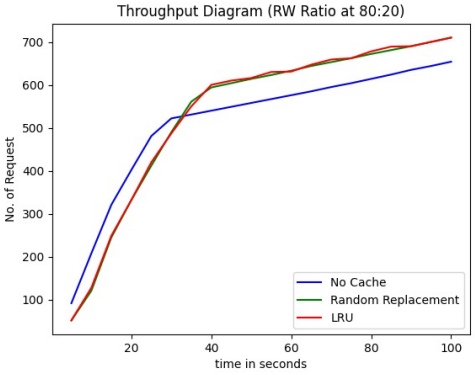












Part 7. Result Discussions

1. As expected, using cache is clearly faster than no cache implementation. Both the throughput and latency can be seen to be improved.
2. Performance tends to be better when reading request count surpasses writing request count. This may be because of more cache hits. Also, it may imply that write request is slower in general compared to our read request.
3. The throughput increase significantly slowed down at around 450-550 requests. After testing, we think it may be because of the heavy workload on the flask instances that is slowing it down.
4. RR and LRU does not seem to differ much, which is expected as we chose to randomly select an existing key as get_key. In real world use case, however, LRU shall be able to perform slightly better in general as it takes better care of locality.
5. After multiple testing, we find that uploading internet speed is also important. Since images are small, bursts in internet speed will effect the latency hugely. This may be the reason for some of the anomalies in the latency graph.

Part 8. Contribution Table

Yih CHENG	Finished MemCache implementation and DB_Operations
JiaWei MA	Finished Database configuration and setup
Ho Wan LUO	Deploy on EC2