ECE1779 Assignment 1 Report - Web Development

Group #5

Yuhan Wei 1000893161

Hsuan-Ling Chen 1002322202

Eric Wang 1002294108

1.0 Introduction

For this assignment, the group was tasked to develop a personal photo gallery where users can upload photos with keywords which can be used to later identify the photos with. We built a storage web application with an in-memory key-value memory cache (mem-cache) in the form of an online album which allows users to upload, search, display images and change configuration settings. The application is implemented using Python, Flask as framework and MySQL as the database, then deployed on Amazon EC2.

Package Required: Matplotlib, flask, mysql-connector-python, numpy, Werkzeug, requests, django

2.0 Web Application Details and Description

The application consists of 5 key components listed as follows (as shown in Figure 1.0): a web browser that initiates requests, a web front end that manages requests and operations, a local file system where all data is stored, a mem-cache that provides faster access, and the relational database which stores a list of known keys, the configuration parameters, and other important statistical values.

To launch the project, run the following command lines:

1. pip install -r requirements.txt
2. ./start.sh

To navigate to the webpage, the server needs to be started on the EC2 instance. Once this is done, the webpage can be accessed via the public IP address of the instance. This will bring the user to the home page where the user can select whether to upload, search, display images, configure settings and show cache statistics.

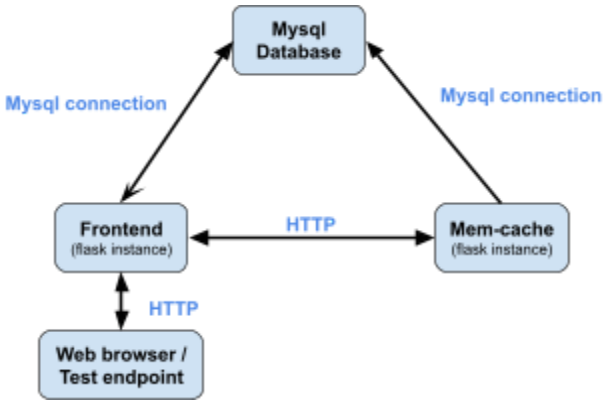


Figure 1.0. Application Architecture.

3.0 Application Architecture

The general architecture and solutions are thoughtfully designed through the key concepts covered in lectures and exercises before development. As an overview, the front-end and mem-cache are two separate Flask-based applications: Front-end (instance 1): port 5000 and mem-cache (instance 2): port 5001. The frontend serves as the client of mem-cache where it would send the request url so the mem-cache can receive and process the request.

3.1 Frontend

As requests initiate from the web browser, the front end is the part of the web application that manages these requests and operations. This part of the project is contained under the ‘frontend’ directory of the project repository. We used html for the route is called and css for the styling of the webpage. There are six pages that each provides different functionalities to fulfill the project’s requirements:

File Name	**Functionality:**
Upload.html	This page allows the user to upload a new pair of key and image. Image given with the same key replaces the original image in the database and is invalidated in mem-cache.
Search.html	This is the page that displays an image associated with a given key.
Display.html	This displays all the available keys and images stored in the database, as well as a button to delete all keys and values from the application (from the file system and mem-cache).
Main.html	This page loads up as the homepage for the application.
Config.html	This page displays cache capacity usage and items (keys, size, and access time) in the mem-cache, allowing parameter configuration and clearing of mem-cache.
Stats.html	This page displays mem-cache statistics over the past 10 minutes in charts.

3.2 Key-Value Memory Cache

The mem-cache is an in-memory cache that is implemented as a Flask-based application to provide faster access. The files for mem-cache are located in the “memcache/cache.py”, “memcache/main.py”. It stores a subset of keys and images and supports the basic operations of PUT, GET, CLEAR, invalidateKey and refreshConfiguration. The mem-cache also supports two cache replacement policies which can be configured in the “Config Settings” page: Random Replacement and Least Recently Used.

The default mem-cache size is selected to be 10.0 MB, which is capable of storing a reasonable number of images in their original format. The user is able to configure the maximum mem-cache size from the page. On an update,

the program first checks that there is space left; if the cache shrinks, keys are dropped according to the replacement policy until it meets the new capacity.

3.3 Database Schema

The database in this project is used to store the list of keys and corresponding images, the configuration parameters of the mem-cache, and the statistics values from the site. Three tables were created named "image_gallery", "config", and "stats". Since there are no direct relationships between these information, the current database schema carries three standalone tables with a primary key each. There is no need for foreign keys because there is no one-to-many relationship in the tables, and no need for junction tables because there is no many-to-many relationship in the tables.

image_gallery
im_key path id

config_param
capacity replace_policy

stats
num_items total_size num_req miss_rate hit_rate time_created

4.0 Design Decisions

During the development process, there were some design decisions made. Here are two of them:
In the mem-cache “Config Settings” page, there were two cache replacement policies: Random Replacement and Least Recently Used. LRU where the least recently used keys and its associated values are discarded first. This is implemented by an ordered dictionary to keep track of what was used when to discard the least recently used key. This algorithm can also be completed with a doubly linked list, if the ordered dictionary is not available data structure. The other alternative that was attempted was to use the Pylru library, which is LRU cache for Python. The code is straight forward, but after diving into it, it was realized that the size that the library returns only gives the number of items in mem-cache, not the actual size of the mem-cache in MB. Therefore, for more flexibility, the LRU is implemented from scratch.

In order to serve the images in json response for the auto-testing API, the version of image sent to the auto-tester is selected to be encoded with base64 since json is text-based format. Another way to encode the images is creating HTML5 canvas or load the image with XMLHttpRequest and use the FileReader API to convert it to dataURL. However this approach lacks browser support and requires more implementation, base64 is the more common method to achieve the intended task. With some design planning, it was decided that the most efficient design for our web application is to store the images in binary form separately from the image paths. This allows the client’s browser to cache the images as its original format in the mem-cache and something that a browser can render as an image for the interactive web interface.

5.0 Evaluation and Performance Graph

The section introduces the design of performance evaluation, detailed evaluation results and some discussion.

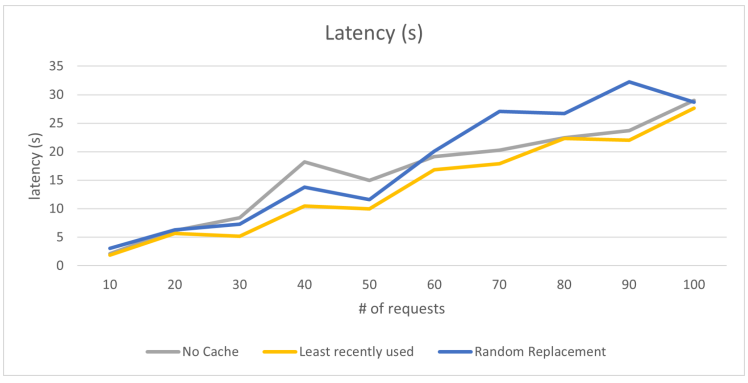
5.1 Evaluation Setup

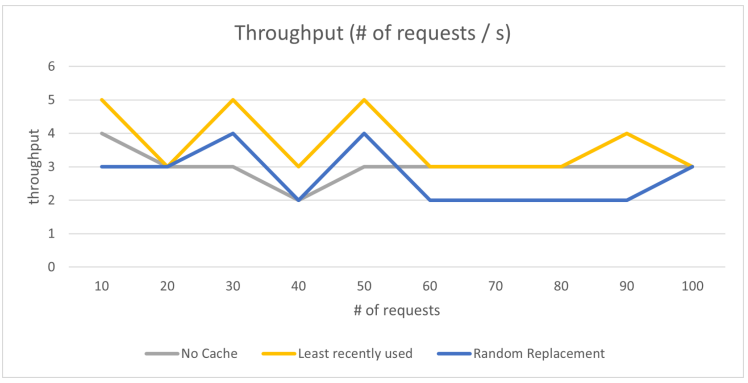
- Set the cache capacity to 10 MB to make sure the replacement will happen.
- Prepare 12 test images with a size range from 5 KB to 2 MB.
- Design 10 request batches from 10 to 100 with a step of 10.
- For each batch, use the read/write ratio to calculate the number of writing and reading operations.
- For the writing operation, automatically generate keys such as ‘test1’, and ‘test2’ and randomly select images from 12 candidate images.
- For the reading operation, randomly select keys which were generated for the writing operations.
- Measure latency and throughput for each request batch.
- Run the evaluation after our application is deployed on AWS.

5.2 Performance Graphs

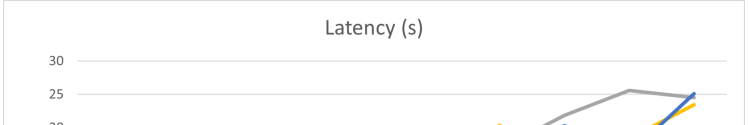
Overall, the performance among three cache settings: no cache, Least recently used and random replacement is very close, especially for the 50/50 read/ write ratio. In theory, the use of a cache will improve the application’s efficiency when the read request is initiated frequently. However, in our application, the frontend and cache communicate with each other via HTTP, which will introduce some processing time costs. As we can see from the following figures, the application with no cache achieved better latency performance when there were a large number of requests for the 20/80 and 80/20 ratios. Some optimization may still be required inside the code. As for the throughput, the performance from the three settings varies with the change in the number of requests. From our results, the cache does not significantly enhance the overall performance. Also, the performance of AWS instances and the network connection introduce some uncertainties to the measurements.

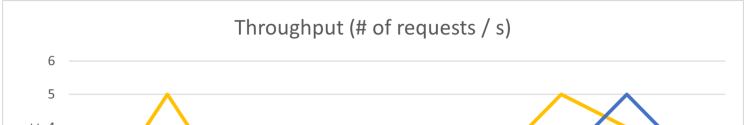
- **20/80 Read-Write Ratio**



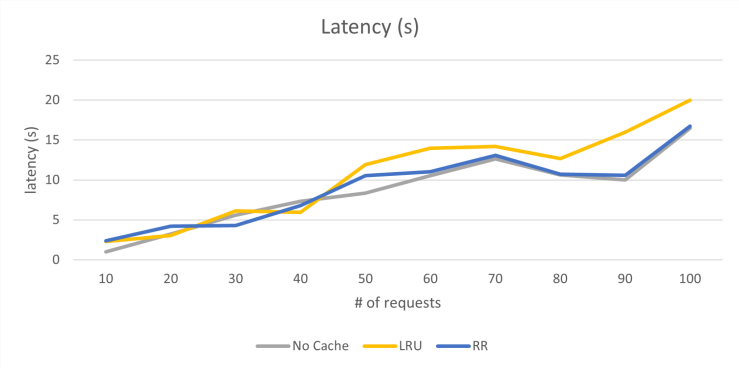


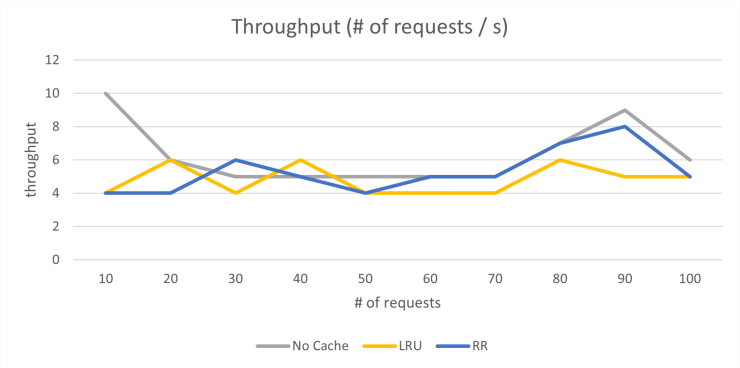
- **50/50 Read-Write Ratio**





❖ **80/20 Read-Write Ratio**





6.0 Conclusion and Future Implementations

In conclusion, the website is able to perform all the functionalities of Assignment 1 requirements. We added two additional features in the frontend: cache capacity usage display, cache item size and access timestamp display. These two features clearly show how the cache works for the user and helped us to debug the application. As for the evaluation section, we designed the experiments based on our own understanding instead of the industry standard protocols. Therefore, the results may not be sufficient to reflect the performance in real-use scenarios.

All of our team members are new to the web development area, and we started this assignment from learning basic techniques. Therefore, we are satisfied with what we have done currently, and will continue to dig into this logic further for the next two assignments. Specifically, we plan to do more research on the typical web project logic to refine our code structure. Also, we will deploy our application with uwsgi in the next two assignments.

Attribution Table

Team Members	Tasks
Eric Wang	● Created MySQL database that supports calls from frontend as well as storing statistics from the mem-cache ● Developed APIs for accessing the database ● Supported deployment process to EC2 instance
Yuhan Wei	● Designed webpages layout, and implemented related HTML,CSS, and JavaScript ● Integrated frontend, mem-cache and database together and conducted the overall testing ● Implementation the evaluation script, summarized and analyzed the results ● Deployed the application to EC2 instance.
Hsuan Chen	● Worked on development of mem-cache ● Replacement Policies ● Documentations