Group number 26
Group members:
Shiyu Xiu, 1004724872
Yuangan Zou, 1004761778
Qian Tang, 1005228656
Assignment 1
General Architecture of application:

The general architecture of our system consists of a frontend, a backend, a database named “memcache” and also a dictionary also named “memcache” with structure {‘key_value’: (path, time)}. The frontend takes care of all user requests and includes five pages: upload, Showimage, display keys, configure memcache and display statics of memcache. Backend refers to the flask instance of memcache and it has functions that do the configuration of memcache, as well as doing operations to guarantee a successful running of the system. (invalidate keys, replacement policy, etc). Both frontend and backend can get access to the database.

The workflow of user storing pictures:

1. User uploads pictures with keys, each key specifically identifying a picture, which is a file path.
2. The frontend writes the pictures to the local file system(on the cloud) and stores the list of keys and their corresponding paths in the database. Frontend will also invalidate the key in memcache. This completes the process of storing.

The workflow of the user retrieving pictures:

1. User provides a key in order to retrieve the corresponding image.
2. If the key is in memcache, then memcache returns ‘ok’ and frontend will display the image.
3. if the key is not in Memcache, memcache returns ‘miss’ and frontend consults database with the key value in order to get the path associated with the key. After getting the path, frontend will go to the path to obtain the file and write it to the local file system. After that, frontend puts the key value and its associated path into the memcache through a PUT request.

The following graphs show the general structure of our application in different stages:

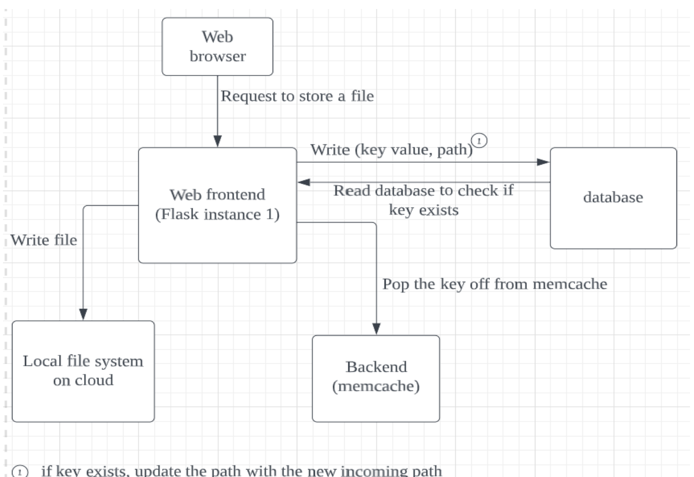

Web browser
Request to store a file
Write (key value, path)
Web frontend (Flask instance 1)
Read database to check if key exists
database
Write file
Pop the key off from memcache
Local file system on cloud
Backend (memcache)
if key exists, update the path with the new incoming path

Figure1: Workflow Design of a user uploading files

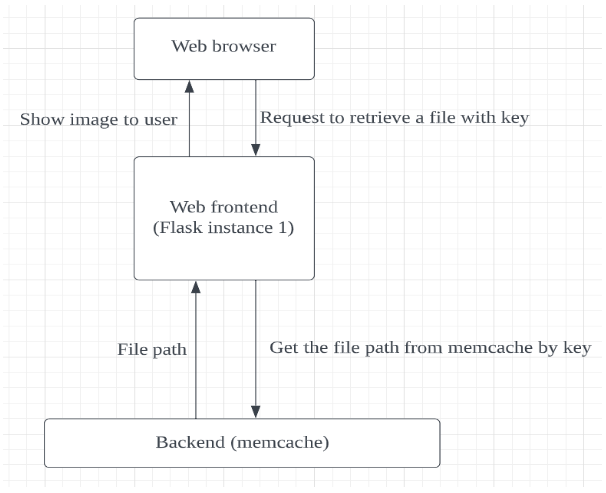

Web browser
Show image to user
Request to retrieve a file with key
Web frontend (Flask instance 1)
File path
Get the file path from memcache by key
Backend (memcache)

Figure 2: Workflow of Retrieving a File: Cache Hit

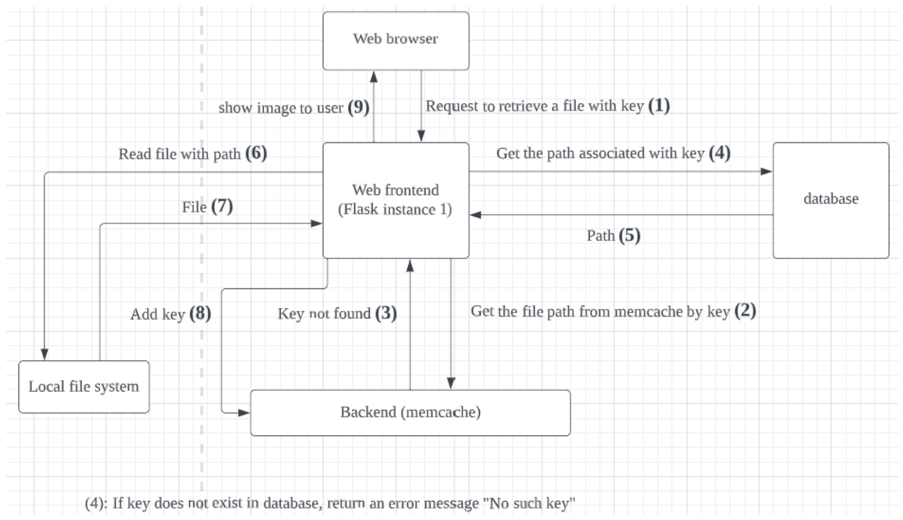

Web browser
show image to user (9)
Request to retrieve a file with key (1)
Read file with path (6)
Get the path associated with key (4)
database
File (7)
Web frontend (Flask instance 1)
Path (5)
Add key (8)
Key not found (3)
Get the file path from memcache by key (2)
Local file system
Backend (memcache)
(4): If key does not exist in database, return an error message "No such key"

Figure 3: Workflow of Retrieving a File: Cache Miss

Database Schema:

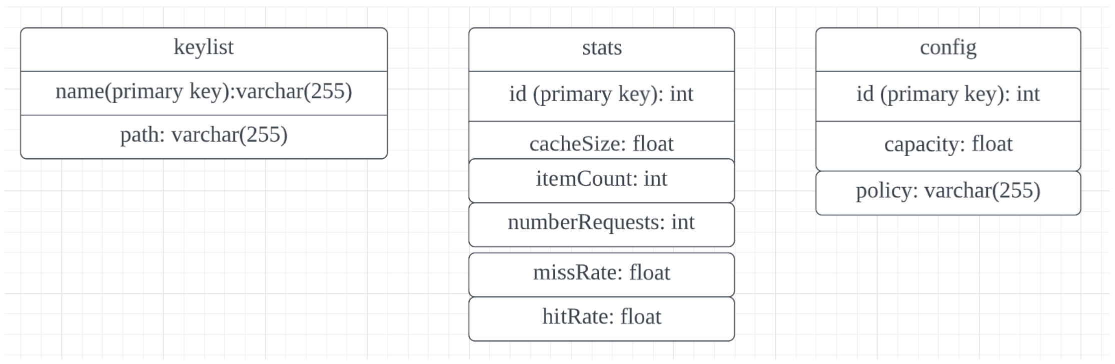


Figure 4: Database diagram

We designed our database so that it has 3 tables: a table “keylist” to store the values of keys as well as their corresponding paths; a table “stats” to store the cachesize in MB, number of items in memcache, number of requests received as well as the missrate and hitrate; a table “config” to store the capacity of replacement policy of memcache, this table will be used by our backend to read configuration data and to modify memcache’s parameters.

Key Design Decisions:

Dictionary design of memcache: A design of structure {‘key’: (path, time)}, time is recorded by using datetime.now(). When we need to invalidate a key, we compare the values of time in order to drop the least used key.

Number of requests in memcache: Invalidate operation is not counted since this is not an operation requested by the user.

Approach to calculate miss rate and hit rate: $miss/hit\ rate\ =\ \frac{num(miss/hit)}{num(miss)+num(hit)}$, alternative is to use $miss/hit\ rate\ =\ \frac{num(miss/hit)}{num(requests)}$. However, num(requests) also includes the number of clear requests from the user which will make the result less accurate.

Approach to go from page to page: User goes back to the main page, then navigates to other pages.

How to configure cache: user sets the value for capacity in MB and chooses a replacement policy, the frontend stores the config info into database, backend reads the table “config” and sets global-variables *capacity* and *policy* to complete the configuration of memcache. Backend also clears the cache after each configuration.

Save space in the local file system: When the user uploads two files with the same key, our system will not only update the path in the database but also will go to the local file system and delete the previous file associated with the key in order to optimize the usage of storage space.

Encode files during transmitting: pass images between API classes in base64 encoding to increase security. The alternative is to transmit image files directly, but this approach includes a risk of images being leaked during transmission.

Randomize file names: use uuid to randomize file names so that every file uploaded to local file system has a unique filename (path). This is to deal with the case that the user uploads 2 different pictures with 2 different keys but with the same filename. For example

(dog.jpg: picture of a dog, key = 1) and (dog.jpg: picture of a cat, key = 2)

In this case, the local file system will delete one of the dog.jpg files to make sure the filenames are unique. To avoid losing files, we randomize all file names.

Performance Graphs:

Config used to test: 10MB memcache, set memcache size to 0 when measuring no cache

We use 20 images for the testing, file size is around 2MB per image.

80:20 read/write ratio:

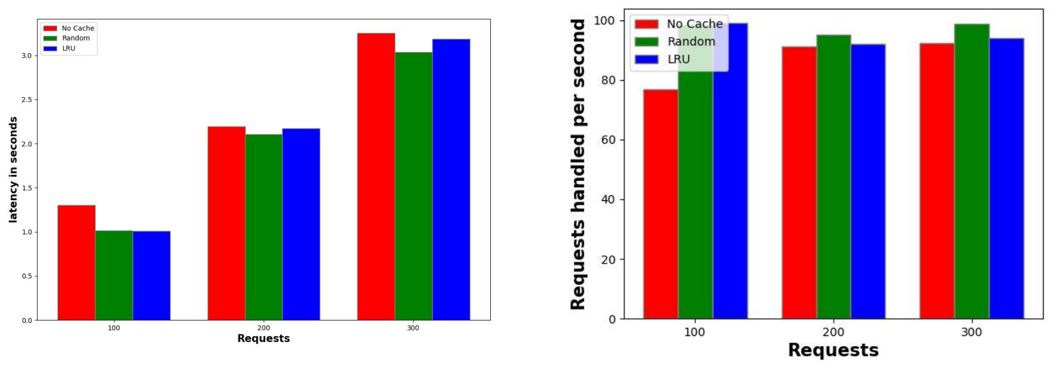
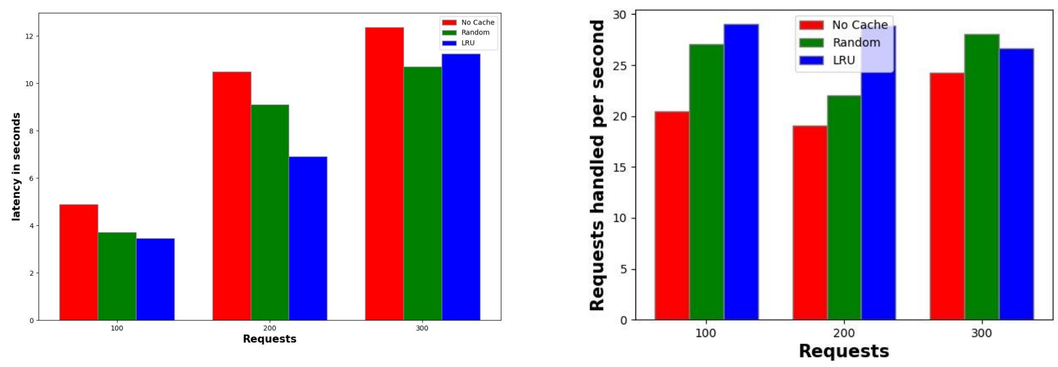


Figure5: Latency and throughput graph for 80:20

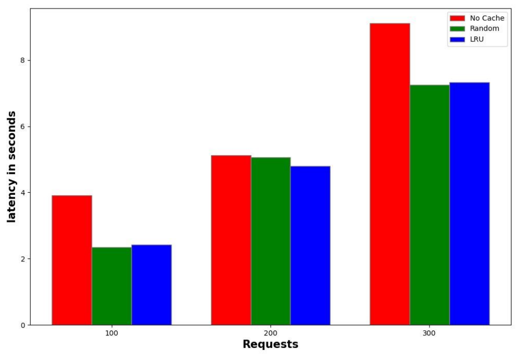
When read/write ratio = 80:20, the chance of cache hit is high. We can observe that our memcache is speeding up our read process. The latencies of both policies are lower than the latency of ‘no cache’. In this case, most of the requests are read, so our system can take about 100 requests per second.

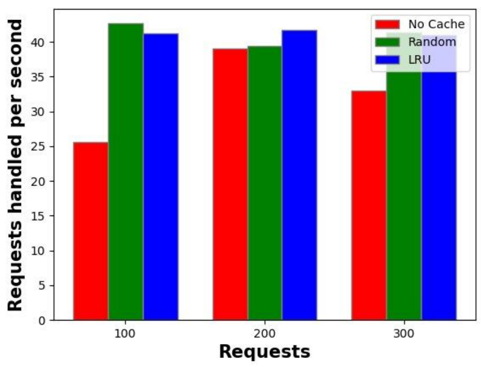
20:80 read/write ratio:



When read/write ratio = 20:80, the % of cache hit is low. In this case, the memcache does not help much in speeding the read process since most of the read requests encounter a cache miss. Since in this case most of the requests are GET, our system takes about 30 requests per second.

50/50 read/write ratio:





When read/write ratio = 50:50, we observe that the three latencies are similar to each other. It is not evident that our memcache is speeding up the read process. This is because instead of removing the memcache, we set the size of memcache to 0. In this case with a balanced read/write ratio, our system takes around 45 requests per second.

Attribution table

Group member	Milestone	Signature
Yuangan Zou	Display Keys, Configure Memcache, construct report	Bill Zou
Shiyu Xiu	deploy on AWS, build out the whole initial structure and build multiple endpoints both front end and backend	Shiyu Xiu
Qian Tang	Show picture page and its API, show statistics page, debugging and generate testing graphs for the system, making design decisions	Qian Tang