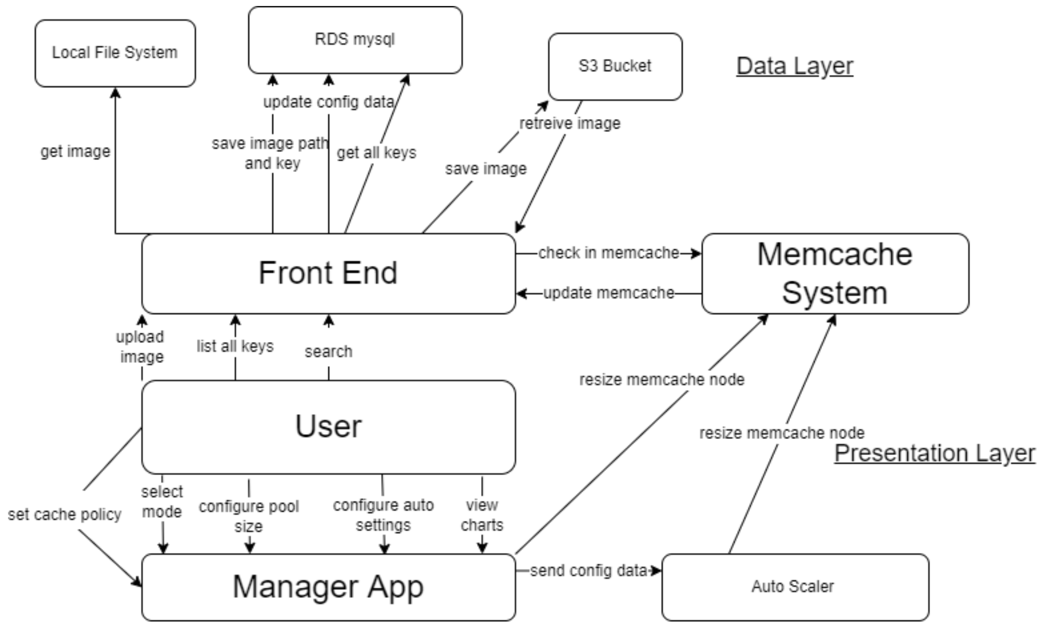
ECE 1779 Assignment 1

Group 21
Yachen Wu 1004232804,
Di Wu 1009758816,
Ziqian Qiu 1004723383

November 23, 2022

1 General Architecture

1.1 Architecture Diagram

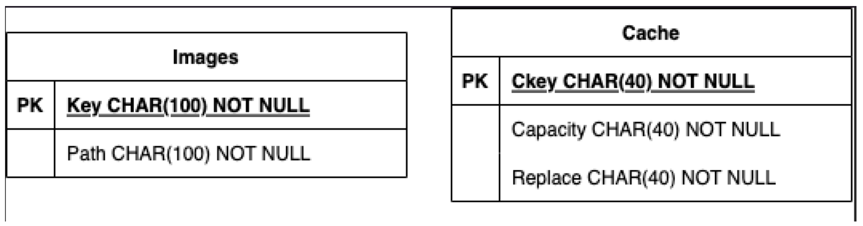


1.2 Explanation

Our design follows the multi-tiered architecture. The basic request flow can be shown in the architecture diagram. Users interact with the presentation layer, which is the Manager app and Front end. There are basically 5 use cases that generate requests at the browser.For the usage, the user will be able to upload image from the local file system, then save the image path to RDS mysql database. It also communicates with the mencache for a cache update. Manager app will be available for users to select manual or automatic mode to generate memcache node. For the list-all-key case, the front end system will get all keys from the database and show on the browser. For the searching case, the front end system will firstly communicate with memcache system to check if the corresponding image is in the system. If not, then the front end system will communicate with database and get the file path. For the cache configure case, the manager app system communicates with various nodes and upload the new configurations respectively. Finally, for the cache charts case, the manager app system communicates with the CloudWatch function and get the latest statistics average of all running nodes.
Overall speaking, when the user generate an event from the browser, the front end communicates with the memcache and the data layer in order to process the requests and finish the event.

2 Database Schema

2.1 ER-Diagram



2.2 Explanation

As shown in our ER-Diagram, we used three independent tables to store image path, cache configurations, and cache statistics changed over time. Three primary keys from these three tables are key, ckey, and time. Our design is properly normalized since the normalization rules are not broken. To be more specific, we will use the five rules of Normal Forms to prove our normalization:

- First Normal Form (1NF): It can be easily shown in our ER-Diagram that each column only includes one type of data. The column names are different if they are different in one table. Also, we will make sure in the python program that there will be no multiple values inserted into one column. We will be using SQL command to get the corresponding data, so the order of the data does not matter. Therefore, our design satisfies the requirements of 1NF.

- Second Normal Form (2NF): We have proved that our design satisfies the requirements of 1NF, so we only need to show there are no partial dependencies in our design. Partial dependency only exists when there are more than one primary keys in a table, and one of the columns depends on only a part of this primary key group. In our design, we only have one primary key in all three tables. Thus, there are no partial dependencies in our design. The requirements of 2NF are also satisfied.

- Third Normal Form (3NF): We have proved that our design satisfies the requirements of 2NF, so we only need to show there are no transitive dependencies in our design. Transitive dependency exists only if one normal column in the table depends on another normal column instead of primary key. In the "image" table, the primary key is "key", and "path" depends on "key". In the cache table, the primary key is "ckey", "capacity" and "replace" depend on "ckey". In the cachechange table, the primary key is "time", and all the statistics data changes with the changing of time, so all the data columns depend on the primary key. Therefore, there are no transitive dependencies in our design. The requirements of 3NF are also satisfied.

- Boyce and Codd Normal Form (BCNF): We have proved that our design satisfies the requirements of 3NF, so we only need to show that "A" is always the primary key for each dependency (B depends on A). In our case, there are no transitive dependencies since the primary keys are independent in these three tables. All the other columns depend on the primary keys. Therefore, the requirements of BCNF are also satisfied.

- Fourth Normal Formal (4NF): We have proved that our design satisfies the requirements of BCNF, so we only need to show there are no multi-valued dependencies in our design. Multi-valued dependency exists only if one single value from column "A" has more than one values from column "B" with the dependency of "B" depends on "A". In our design, all the normal columns depend on the primary key. However, there will be no duplicated primary keys in our tables. In other words, one primary key value will always find only one value from each other column. Therefore, there are no multi-valued dependencies in our design. The requirements of 4NF are also satisfied.

3 Design Decisions

- Configuration file:
  As we extend the project on top of A1, we created a configuration file to avoid redundancy in code. Since we need to call database many times, we also created a file that calls mysql.connector.connect using the relative information imported from config file. The config file also included the aws and rds related credential information. We add this file to .gitignore to prevent it from leaking personal information while pushing to github.

- Single responsibility principle:
  We separated the functions that deal with internal logic while load/push a file to/from cache. Such as load image to database and get an image. Therefore, in the API we only consider the requests and call the functions we wrote in another file. In this case, each part of the flask instance does their own job, hence it's easier to extend and elaborate on top of it later.

- Memcache pool design:
  Considering about the Manual and automatic mode specified in the document. We decided to deploy the frontend, memcache, manager app, autoscaler as four flask instance. When users manually select the pool size from minimum 1 memcache node to maximum 8 memcache node, new instance will be created accordingly and by the use of built-in feature within ec2.createfunction all the instance will be initialized and running the memcache portion. In this way, all the memcache nodes will be running on different public Ip address but the same port number.

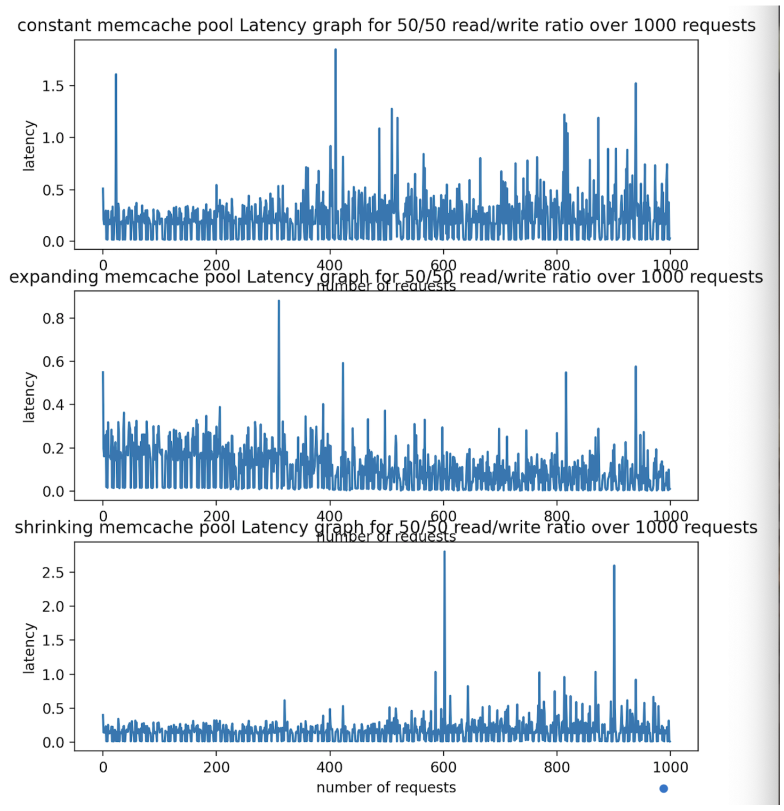
- RDS and S3 transferring data:
  After transferring the data from mysql to RDS and S3 bucket, we decided to store the images uploaded from the users themselves into S3 bucket that we created meanwhile the path and object value of the image will be stored into the RDS mysql Database accordingly. Every single time the user retreive or upload the data, based on the image name s3 bucket will do the operation to get it from the server.

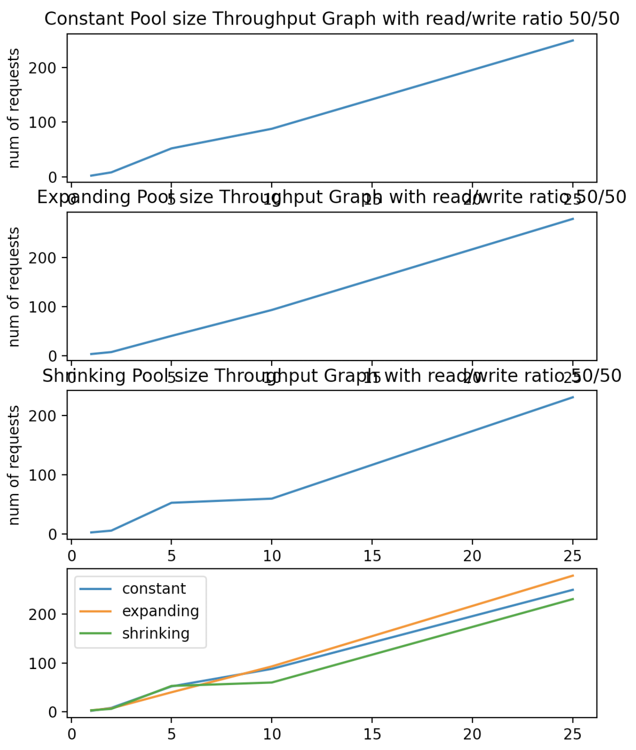
4 Results

4.1 How does Auto Scaler Work

In our auto scaler design, we wrote a function to run the basic function of auto scaler, and we have a flask route to update the miss rate thresholds and ratios as global variables in the auto scaler python file. When users change config in the manage app front end, this route will be called and the corresponding thresholds and ratios will be updated. The basic auto scaler function includes a while loop, in which compares the average miss rate with the thresholds every 60 seconds. If the average miss rate is greater than the max threshold, it will expand the memcache pool by expand ratio. If the average miss rate is smaller than the min threshold, it will shrink the memcache pool by shrink ratio.

4.2 Graph Analysis - Latency



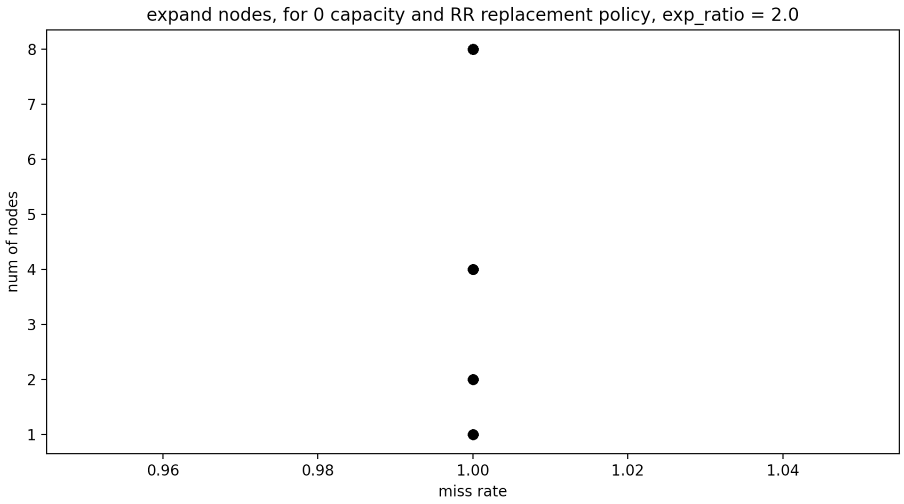


Note for the following tests, we have a 50/50 read write ratio within 1000 requests. We used random replacement policy with 2MB cache size, and the test image size are approximately 10KB each. The time needed to retreive data from cache is approximately 0.005 and Image dataset found on Kaggle: https://www.kaggle.com/datasets/vishalsubbiah/pokemon-images-and-types?select=images

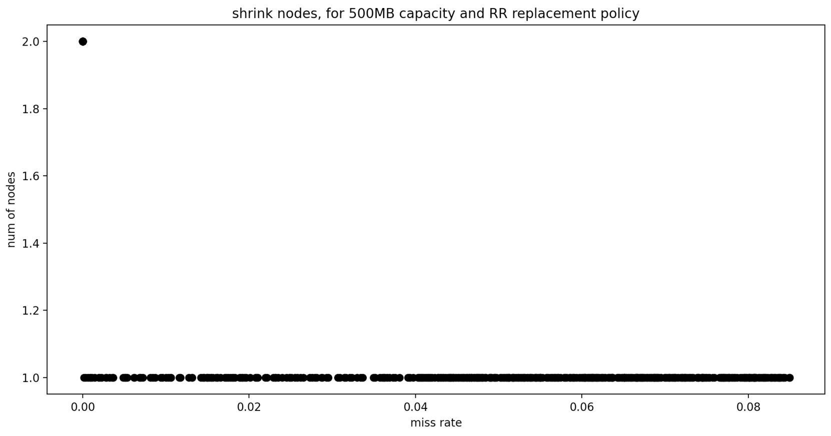
The three graphs demonstrated the pool latency in manual mode when 1.pool size fixed, 2.pool size growing, 3.pool size shrinking; these are plotted respectively in the graph above from top to bottom. By observation, the latency started off with a value approximately 0.35 and oscillates after that in all three groups. When the pool size is fixed, as the number of request increases, the memcache is gradually filled up. However, since it can only contains a certain amount of images, the get requests begins to miss hence need to retrieve the corresponding data URL saved in RDS and then from S3. Therefore, the cache latency gradually increases as the number of requests goes up when the size of memcache pool is fixed. In the other hand, the latency graph of expanding memcache pool size is different. By observation, the latency is gradually decreasing as the number of memcache are added into the pool. This is because as more memcaches become available, the more images can be stored in cache pool. In this case, less get requests are directed to S3 hence it decreased the overall latency with expanding memcache pool. In contrast, as memcache pool size shrinks, more requests are directed to S3 since there are less images saved in memcache. Therefore, the latency of graph 3 is gradually increasing. Noted all three graphs have a few outliers, which might be caused during connection of are affected by instance initiating at request 300, 600 and 900.

As for throughput graph, we can observe the performance from the bottom one including the comparison between all three cases. At seconds = 5, the number of requests initiated by the shrinking memcache pool is the largest amongst three. However, after seconds= 7, the expanding memcache pool always initiated the most requests. This is because after the number of memcache increases, more requests are directed to cache but not s3. This saves plenty of time when retrieving data as well as decreased the latency. Therefore, the number of requests are higher than the other two.

4.3 Graphs for Auto-Scalar



The scenario I choose for auto scaler to automatically expand the node size is: 0MB capacity and RR replacement policy, and my max miss rate threshold is 80 percent. In the first 60 seconds, the miss rate keeps at 100 percent since nothing can be stored in memcache pool. After auto scaler detects the current average miss rate and compares it with the max miss rate threshold, new memcache instance are created based on the expand ratio (2.0), which means there are 1 * 2.0 = 2 memcache instances now. The new memcache instances carry a miss rate of 0, however after several searchings, the miss rate will immediately become 1. After the second 60 seconds, auto scaler detects the current average miss rate and compares it with the max miss rate threshold and new memcache instance are created based on the expand ratio. This time, there are 2 * 2.0 = 4 memcache instances in total. The rest of the graph are based on the similar logic. Finally, there will be 8 instances in total, the miss rate for all 8 instances will be 100 percent.



The scenario I choose for auto scaler to automatically shrink the node size is: 500MB capacity, RR replacement policy, max miss rate threshold 10 percent, and shrink ratio 0.5. To begin with, there are two memcache instances in the memcache pool. Due to the large capacity of the memcache nodes, our testing images can all stays in the memcache, thus the miss rate stays at 0. After the first 60 seconds, auto scaler detects the current average miss rate and compares it with the min miss rate threshold, then one of the instances terminated. After that, the miss rate rises slowly from 0 to a relatively low level. This is because one 500MB node can still considered as a very large cache.

Di Wu	Yachen Wu	Ziqian Qiu
auto_scaler.py	manager app	Change web front-end component
MD5 hashing	Delete all application data button	Change key-value memory cache
start.sh	Clear memcache data button	rewrite web frontend API
Connect auto scaler with manager app	Memcache pool configure (capacity and replacement policy).	write script to retrieve the latency for latency and throughput graph statistics
Deploy on ec2 instance	Resize policy: Manual and automatic mode	a2latency.py
Create scaler functions	Chart show the statistics for the memcache pool	data cleaning and renaming of image dataset found on Kaggle
Explain how the auto scaler works in the report	Design decisions on report	db connect simplification
Draw the auto scaler graphs	bash script to ease the web app running	Plot 6 Performance Graphs for report
Testing	Testing	Testing

[illegible]　　Yachen Wu　　Ziqian Qiu