Praktikum: Cloud Data Bases (IN0012, IN2106, IN4163)
Milestone 3 Report

Group No. 13; David Silva, Krisela Skenderi, Lukas Bernwald

15/12/2021

1 Introduction

This report contains the experimental evaluation of the 3rd Milestone of the Cloud Databases *Praktikum* project.

With the introduction of the ability to add multiple servers to handle client requests, there is a need to measure how this increase affects the system's performance. In addition, stress testing the system is an important tool to evaluate the correctness of the communication protocols and their implementation.

Although the present evaluation is not thorough, we argue that such an evaluation falls out of the scope of the current milestone, which intends to bootstrap the development of scripts and tools that will aid us in the evaluation of the system, and that will be improved during the following milestone and the project. As such, we also discuss the current limitations of our tools and experiments and how we intend to improve them going forward.

The following section contains a description of our experimental setup, and a discussion of the experiment's results.

2 Experiments

To evaluate the system's performance, we created a *Java* application [1], that simulates a real-life scenario. The application starts by launching the *ECS* at a given port.

It is also able to **create a configured amount of servers** that are created with certain parameters (cache strategy, cache size and B-Tree node minimum-degree). The servers may be created at the start of the run, or late through the use of *timed events*. These events can also be used to shutdown servers after a given time delay.

The application also **starts a configured number of clients** that connect to a random server from the list of created ones and that execute *get*, *put* and *delete* operations from the *Enron email dataset*. Clients continuously execute these operations in a cycle of *put-get-delete*. This is done in order to try to balance the number of operations of each time. However, deletes only occur after 30% of the client's emails have been sent, in order to allow for the growth of the amount if data in the servers. The clients may also be created using *timed events*.

Currently, we measure the number of *get*/*put*/*delete* operations and respective fails in one second time-steps. We also track the time needed for each operation.

[1]see `src/main/java/de/tum/i13/simulator`

2.1 Setup

To evaluate the caching strategies (section 2.2.1) we create a system with 5 servers and 10 clients and analyse number of operations per second as well as the average operation times for the different strategies (LRU, LFU and FIFO). For this experiment we fix the cache size at 100.

To evaluate the caching size (section 2.2.2) we use the same system used to evaluate the caching strategy but fix the strategy to the best one obtain in the previous experiment. We run the experiment for sizes 50, 100 and 500.

Finally, in order to evaluate the system behaviour (section 2.2.3) we create a system for servers (using the best configuration as per the previous experiments) and clients. We start by creating one server, and periodically creating clients, until twenty clients are created. This is done in order to try to overload the server. Then, we periodically create servers, until ten servers are created. and evaluate the systems response.

2.2 Results and analysis

2.2.1 Caching strategy

As can be seen in figures 1, 2 and 3, all strategies achieved approximately the same performance. We attribute this to the fact that the performance of caching strategies is related to the access patterns of the data (temporal and spatial locality), which is not simulated by our tool (random access). To better understand the best strategy, experiments using real data access patterns would have to be performed. Going forward, we will use the *LFU* strategy in our experiments.

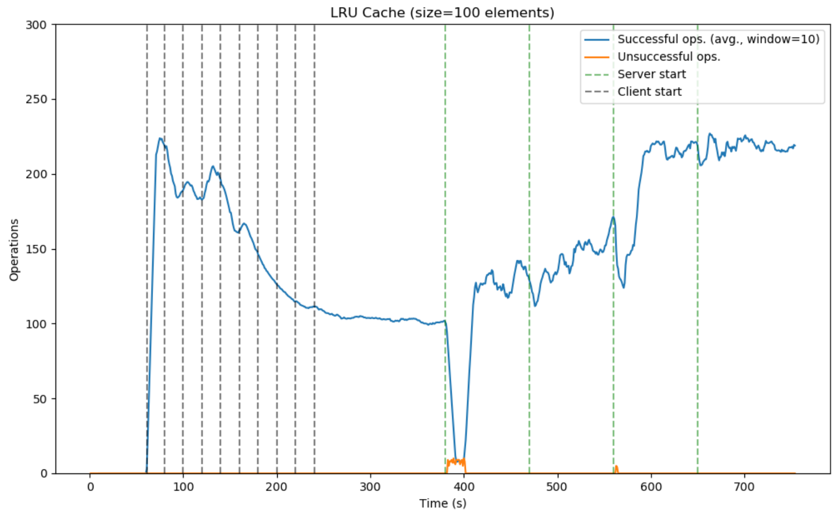

Figure 1: Number of successful (blue) and unsuccessful (orange) operations per second using LRU caching strategy performance. Cache size of 100 elements. System of 10 clients (client starts are marked by dotted gray line) and 5 servers (server starts are marked by dotted green lines).

2.2.2 Cache size

As can be seen in figures 2, 4 and 5, larger cache sizes were able to achieve greater performance. This is due to faster accesses to main memory than to persistent storage. Once again, we argue

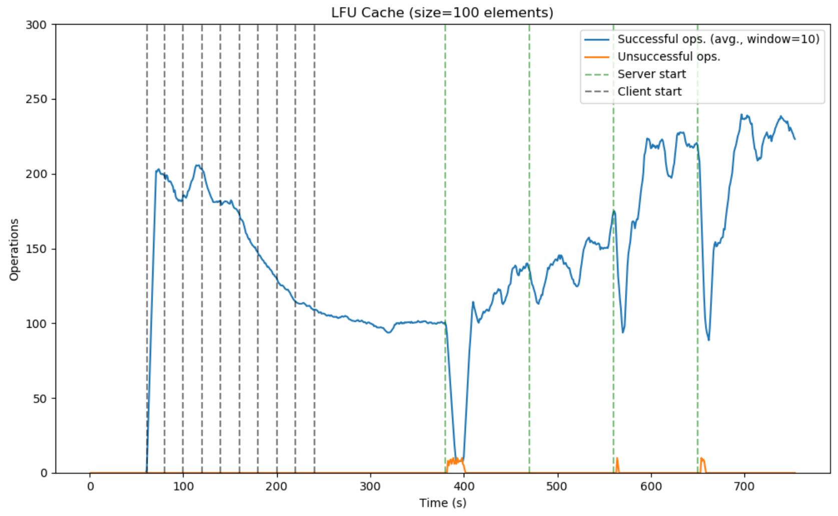

Figure 2: Number of successful (blue) and unsuccessful (orange) operations per second using LFU caching strategy performance. Cache size of 100 elements. System of 10 clients (client starts are marked by dotted gray line) and 5 servers (server starts are marked by dotted green lines).

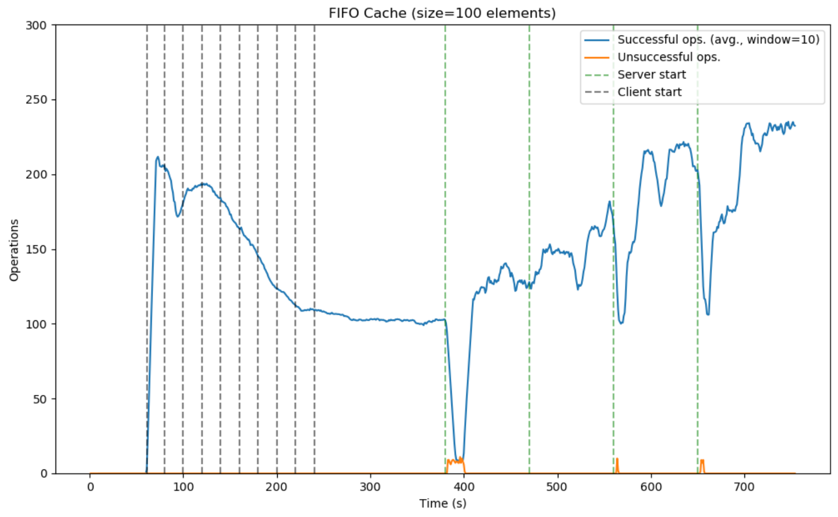

Figure 3: Number of successful (blue) and unsuccessful (orange) operations per second using FIFO caching strategy performance. Cache size of 100 elements. System of 10 clients (client starts are marked by dotted gray line) and 5 servers (server starts are marked by dotted green lines).

that this results would be more divergent if access patterns had spatial and temporal locality. In the next experiment, a cache size of 500 elements will be used.

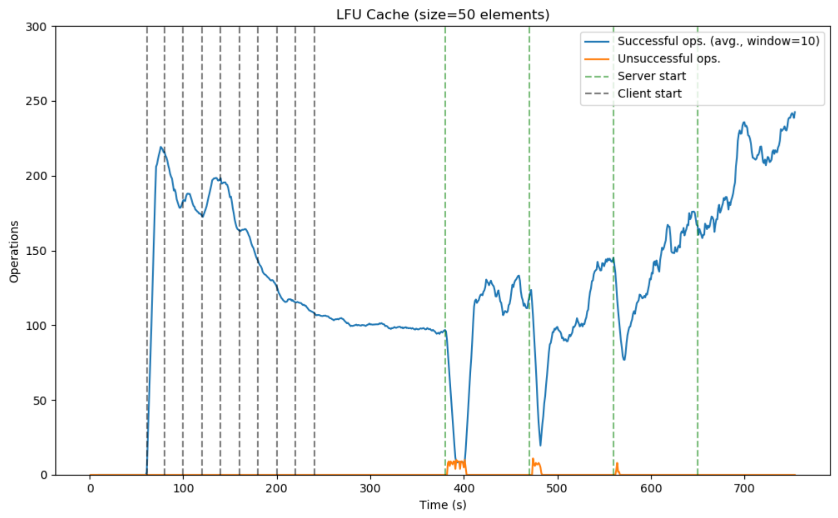

Figure 4: Number of successful (blue) and unsuccessful (orange) operations per second using LFU caching strategy performance. Cache size of 50 elements. System of 10 clients (client starts are marked by dotted gray line) and 5 servers (server starts are marked by dotted green lines).

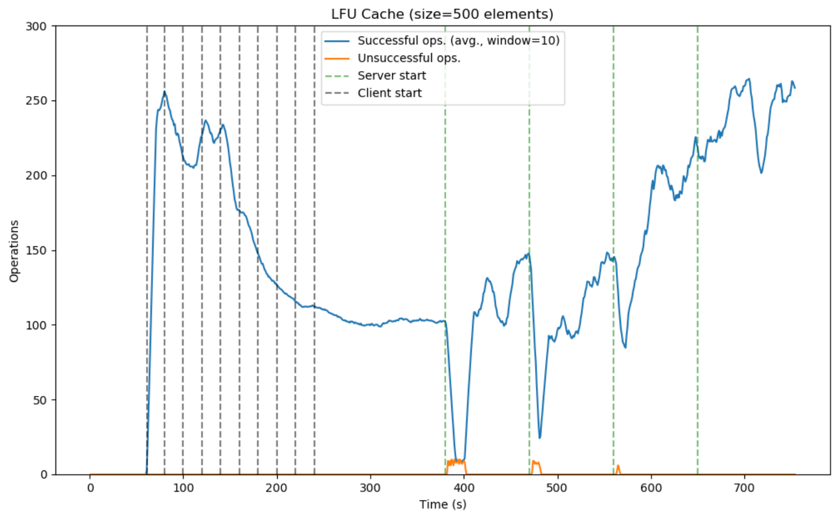

Figure 5: Number of successful (blue) and unsuccessful (orange) operations per second using LFU caching strategy performance. Cache size of 500 elements. System of 10 clients (client starts are marked by dotted gray line) and 5 servers (server starts are marked by dotted green lines).

2.2.3 System behaviour

The system behaved as expected. In figure 6, we can see that as the number of clients increased (dotted grey lines), the number of requests that could be processed per second decreased (server overload). When servers were added, the number of successful operations dropped and the number of unsuccessful operations increased, because the servers are reorganizing and aren't able to process the incoming requests. We can also see that with the addition of new servers there is an increase of the stable throughput (system can handle more requests). For example, at the end of the test, the system can handle about 200 requests per second by the twenty clients whereas at the with only one server, the system could only handle about 75 requests per second. In the future, we would like to be able to use the tool to measure the average time needed for system reorganization.

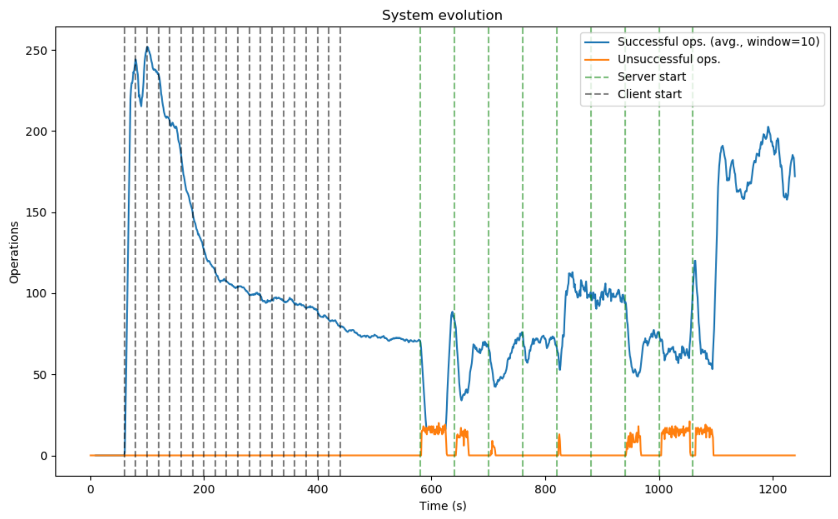

Figure 6: Number of successful (blue) and unsuccessful (orange) operations per second using LFU caching strategy performance. Cache size of 500 elements. System of 20 clients (client starts are marked by dotted gray line) and 10 servers (server starts are marked by dotted green lines).

Table 1 shows the average latency to perform each of the operations (get, put and delete). As expected, get operations performed significantly better, since these operations don't require writing to persistent storage, and aren't affected by system reorganization. Put and delete operations have similar average times because they perform essentially the same operation in the server.

Table 1: Average operation latency (seconds)

Get	Put	Delete
0.09743759555	0.2119050575	0.2130135604

2.3 Limitations and future work

The current simulations could be improved to allow **gathering extra metrics**. Namely, we would like to get **measurements** on the **number of cache misses** and persistent/cache-storage **access times**. This would allow us to better investigate the **optimal size** for the cache. Measuring the best **caching strategy** is also affected by the access patterns so the obtain values can be skewed by the simulation mechanisms. We would also like to collect events from the servers and *ECS* in order to better estimate system re-organization (hand-off, start, shutdown) times.

These metric are currently not collected due to persistent storage design choices, which limit the amount of servers we can create in a single process (we are currently using one process per server, instead of one thread per server). We intend to tackle this limitation in the future.

It would also be interesting to evaluate the performance of the B-Tree persistent storage implementation, regarding the size of the blocks.