Performance Evaluation Study

CSE 503S

Austin Tolani

# Introduction

For my creative project for CSE330S, my partner and I developed a full-stack web application that allowed users to rate the food at dining halls on Washington University’s campus. This web application was hosted on an Amazon t2.micro EC2 instance and used Apache to serve our web application and MongoDB as a database solution. When configuring the Amazon EC2 instances, I was overwhelmed by the options that one has to choose from: different options for CPUs, memory and network performance as well as different options for the type of instance. Naturally, I was curious as to the effect of these configurations on web server and database performance.

In this study, I will explore the performance of different Amazon EC2 instances. More specifically, I will compare the performance of Apache HTTP Server and MongoDB on the different instances of the same “family” of EC2 instances (t2.micro,t2.small,t2.medium,t2.large,t2.xlarge) as well as across similarly priced instances in different “families” including general purpose instances and compute optimized instances (T2,T3,M4,C4).

The tables below show the features of the instances included in this study as well as their on-demand pricing in the US East (Ohio) region as of December 2020. Amazon classifies all of the instances except for the C4 instance as “General Purpose Instances” while they classify the C4 instance as a “Compute Optimized Instance”

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Instance** | **vCPU** | **Memory** | **Network Performance** | **Price ($ per hour)** |
| T2.micro | 1 | 1 | Low to Moderate | 0.0116 |
| T2.small | 1 | 2 | Low to Moderate | 0.023 |
| T2.medium | 2 | 4 | Low to Moderate | 0.0464 |
| T2.Large | 2 | 8 | Low to Moderate | 0.0928 |
| T2.Xlarge | 4 | 16 | Low to Moderate | 0.1856 |
| T3.Large | 2 | 8 | Up to 5 Gps | 0.0832 |
| M4.Large | 2 | 8 | Moderate | 0.10 |
| C4.Large | 2 | 3.75 | Moderate | 0.10 |

# Experimental Setup

A total of 8 instances were used to conduct experiments. All were mounted with the Amazon Linux 2 machine image with the default configuration. Once initialized, the same process was followed to set up each instance. First, all packages were updated and Development Tools were installed. Then, Apache HTTP Server (version 2.4) and MongoDB (version 4.4) were installed and configured to start upon reboot. For MongoDB, the configuration file was edited so that connections could be made from any IP address. Next, the respective ports for Apache and MongoDB were opened by editing the security groups of the instances in the AWS console.

Before any tests were run for either of the experiments, each instance was always stopped and then restarted. This was done to normalize the effect of CPU credits across the burstable instances (T2 and T3). Burstable instances have a baseline level of CPU usage available with the ability to “burst” above this level. The ability to burst is dependent on CPU credits that are earned every hour. Stopping and then starting a burstable instance sets CPU credit to 0.

All testing was done on a separate EC2 instance. The benchmarking tools used in this study can be resource intensive, so running them on a separate instance ensured that the benchmarking tools did not affect the resources available to what service was being tested. Additionally, when testing both Apache HTTP Server and MongoDB, the Private IPv4 address of the instance being tested was always used to avoid contention from other networks.

## Apache Web Server Performance

To measure the performance of Apache HTTP Server across instances, Apache Benchmark was used. This is a command line tool that allows you to benchmark web servers. Apache Benchmark takes two key parameters: the number of requests and the concurrency level, or the number of requests to perform at a time. Exploratory testing found that increasing the number of tests generally had a negligible effect on the outcome of Apache Benchmark. Changing the concurrency level had a significant effect on outcomes. To explore this effect further, my experiment involved running Apache Benchmark on each of the instances at a fixed number of requests (10,000) and at increasing concurrency level. 10 concurrency levels were tested for each instance, starting at no concurrency level (only one request at a time) and then increasing the concurrency level by 100 until a level of 1000 was reached.

Apache Benchmark returns many values that reflect the performance of the web server. In this experiment, the two values that I considered were the number of requests per second and the average time spent per request. These two values represent two of the most important measures of server performance which is how many requests can be handled and how fast is a request handled, respectively.

Running 10 Apache Benchmark tests for each of the 8 instances meant that the tool had to be run a total of 80 times. Starting, waiting, and parsing the response given by the command line tool is a tedious and timely process. To automate this process, a Bash script was created to automate the process of calling Apache Benchmark repeatedly from the command line. This script allowed the 10 tests for each instance to be initiated with a single keystroke. Additionally, it allowed the intervals between each test to be standardized across all of the instances. The primary output of Apache Benchmark is printed to the terminal, meaning that after running the Bash script there were hundreds of lines of text in the terminal. Instead of searching through this text by hand to find the relevant values, a Python script was created to automate this process and find relevant values using Regular Expressions. The output of the Python script was then copied to an Excel spreadsheet for analysis.

## MongoDB Performance

To measure the performance of MongoDB across the instances, Yahoo! Cloud Serving Benchmark (YCSB) was used. This is an open-source program to evaluate the performance of databases. It is particularly suited towards benchmarking NoSQL databases, making it a good choice for benchmarking MongoDB. This tool is also run from the command line. YCSB is run in two stages. The first stage involves loading the

initial working set into a blank database. This stage takes two key parameters: record count and threads. The second stage involves actually running read and update operations that simulate an actual workload. This stage also takes two key parameters: operation count and threads. In both stages, the number of threads represents the number of clients connecting to perform the operations. Similar to the web server performance experiment described above, exploratory testing showed that increasing the number of operations (in either stage) had a negligible effect on performance. Increase the number of threads had a significant impact on performance. To explore this effect further, YCSB was run with a fixed number of inserted records/operations (10,000) and with increasing threads for both stages. For each of the 8 instances, YCSB was run for both stages with 1,2,4,8,32,64 and 128 threads respectively.

YCSB produces many results of interest. In this study, the primary value of interest was the number of operations/seconds or throughput that each instance supported. Each test consists of the two stages (each resulting in a measure of throughput) meaning that each test had a throughput value that represented inserts and read/write respectively.

Like Apache Benchmark, YCSB is run in the command line and prints its output to the command line. Running YCSB 56 times is a timely and tedious task. Similar to the web server experiment described above, a Bash script was created to automate the process of calling YCSB repeatedly and a Python script was created to parse the output.

# Results

## Apache Web Server Performance Results

## MongoDB Performance Results

# Discussion

# Limitations

# Conclusion