BirdWatch
Cloud Application Report

Nuno Nogueira, João Sampaio, Pedro Braz

December 5, 2014

1 Introduction

BirdWatch is a cloud application that gathers information about bird sightings on a database. The received logs are processed internally through a MapReduce system to simplify the querying on our database. The application users are then able to access that data through our webserver.
It also supports a load balancing mechanism to be able to scale according to its usage.

The following paper will explain the reasoning behind our cloud application as well as the auto scaling mechanism implemented.

2 Webserver

2.1 Application Server

Our webserver is entirely written in javascript, the server uses Node.js to run our javascript code. What is specific about our webserver is that Node is fundamentally single threaded, so it deals with all requests asynchronously. Being able to run well under load while having to query the database. Node alone provides a very primitive way of building a http server, that is why we use the express.js framework on top. Express simplifies the process behind defining routes for receiving and managing client requests.

The webserver provides 3 HTTP routes for the queries:

- **POST /bigbirds** : Given a date, identify the tower that observed the bird with biggest wingspan, in a raining situation.
- **POST /totalweight** : Given a date and a tower identifier, show the total weight of all the birds seen by that tower on that day.
- **GET /unseen** : List every marked bird that hasn't been seen for more than a week.

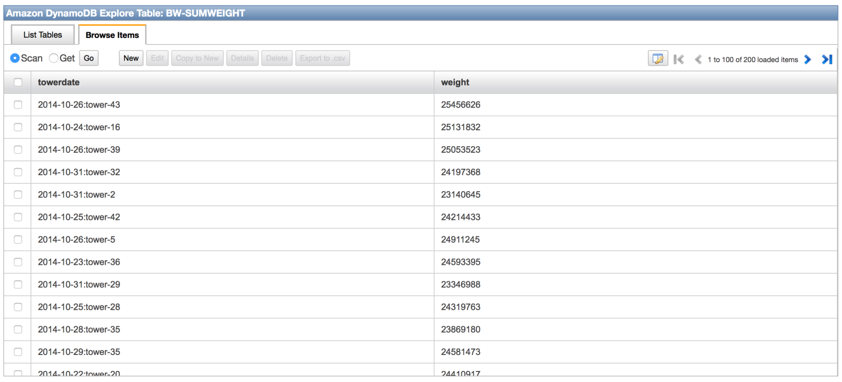
We also introduced another route to test the performance of our servers:

- **GET /test** : Does asynchronous calculations to waste cpu cycles and put load the server.

2.2 DynamoDB

DynamoDB is a NoSQL database provided by Amazon Web Services. This database works also as a Distributed Hash Table, because of the way it separates the database into multiple machines. That is why primary keys can only be of 2 types: hash or hash and range. The use of the hash is to map it to its corresponding location.
For our application, the choice of using DynamoDb is good, because a NoSQL database is more flexible in the way data is stored and queried. Also performance for manipulating the database is not important because the queries we perform are relatively simple.

BW-MAXWINGSPAN To find the maximum wingspan the key is the date, it is mapped to the the tower identifier, that, on that day, had the biggest wingspan value.

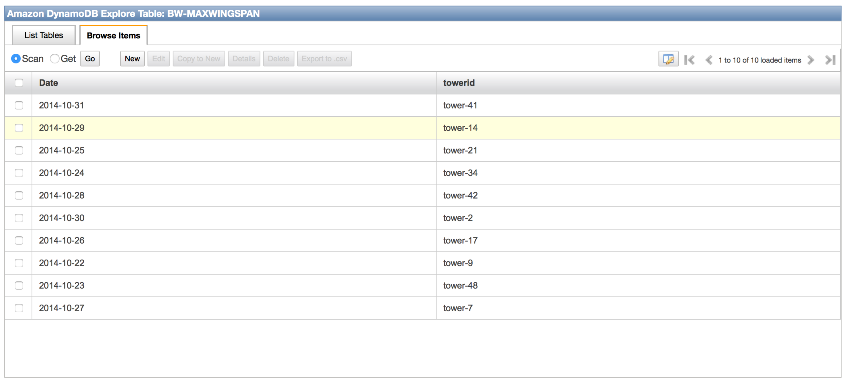


Amazon DynamoDB Explore Table: BW-SUMWEIGHT

1 to 100 of 200 loaded items

towerdate	weight
2014-10-26:tower-43	25456626
2014-10-24:tower-16	25131832
2014-10-26:tower-39	25053523
2014-10-31:tower-32	24197368
2014-10-31:tower-2	23140645
2014-10-25:tower-42	24214433
2014-10-26:tower-5	24911245
2014-10-23:tower-36	24593395
2014-10-31:tower-29	23346988
2014-10-25:tower-28	24319763
2014-10-28:tower-35	23869180
2014-10-29:tower-35	24581473
2014-10-22:tower-20	24410917

BW-SUMWEIGHT To find the total weight, we use as key the concatenated string of the tower identifier and the date.

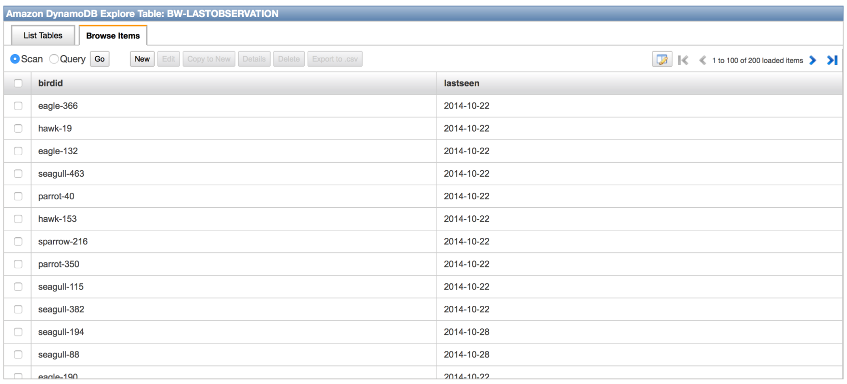


Amazon DynamoDB Explore Table: BW-MAXWINGSPAN

1 to 10 of 10 loaded items

Date	towerid
2014-10-31	tower-41
2014-10-29	tower-14
2014-10-25	tower-21
2014-10-24	tower-34
2014-10-28	tower-42
2014-10-30	tower-2
2014-10-26	tower-17
2014-10-22	tower-9
2014-10-23	tower-48
2014-10-27	tower-7

BW-LASTOBSERVATION The table that allows us to list the birds that haven't been seen for more than a week, uses the bird identifier as a hash key, to make each bird unique in the table, the range is used on the date when it was last seen, so that we can have an ordered list to compare with the date from a week ago.



birdid	lastseen
eagle-366	2014-10-22
hawk-19	2014-10-22
eagle-132	2014-10-22
seagull-463	2014-10-22
parrot-40	2014-10-22
hawk-153	2014-10-22
sparrow-216	2014-10-22
parrot-350	2014-10-22
seagull-115	2014-10-22
seagull-382	2014-10-22
seagull-194	2014-10-28
seagull-88	2014-10-28
eagle-190	2014-10-22

3 MapReduce

3.1 Description

Given a set of logs, our MapReduce application will process them, to efficiently answer the 3 queries from our webserver. All log entries have the following format:

<$tower_{id}$>, <date>, <time>, <$bird_{id}$>, <weight>, <span>, <weather>

3.2 Mapper

For each log entry, the mapper will return **two key-pair values**.
For the first and second queries the output will be the following:

key	value
$< date >$	$< tower_{id} >, < weight >, < bird_{id} >, < wingspan >$

For the third query, the output will be the following:

key	value
$< bird_{id} >$	$< date >$

To create this using just one mapper we had to create a ***MapperOutputWritable*** class to group each mapper's values in the same set and identify each one based on the flag ***querytype***. Where the value is 0 for query 1 and 2, and the value is 1 for query 3.

3.3 Combiner

The goal of the combiner is too reduce the amount of data transferred to the Reducer.

Our mapper output could be of two types, depending on it being query1 and query2, or query3:

In the first and second query, for a record with a $bird_{id}$ (tagged bird), we remove any duplicated entries. For the key $< date >$ and value ($< tower_{id} >, < weight >, < bird_{id} >, < wingspan >$), we send the value that has a wingspan value greater that 0 if possible or 0 otherwise. For the case where the record does not have a $bird_{id}$ the value is written as is, as we need the weight value for all untagged birds.

For the third query, our combiner will only send the key $< bird_{id} >$ and value $< date >$ with the largest date value found for that bird.

3.4 Reducer

The Reducer is able to return 3 types of values as its output. In the case where the ***querytype*** field is 0 the reducer will return the following outputs:

key	**value**
$< tower_{id} >, < date >$	$sum(weight)$
$< date >$	max(wingspan)

Where the **sum(weight)** is the total weight of all birds observed by a tower in a certain date and **max(wingspan)** refers to the biggest wingspan for an observed bird in a certain date. These values are calculated by adding all the weight for the key ($< tower_{id} >$,$< date >$) and by checking the maximum wingspan for the same key $< date >$.
For ***querytype*** with value 1, the reducer's output will be:

key	**value**
$< bird_{id} >$	$< lastseen >$

The field $< lastseen >$ corresponds to the date when the bird was last observed. We verify for the same $< bird_{id} >$ the biggest $< date >$.

4 Load Balancing

The values required for maintaining a stable application will depend on the programming languages and frameworks used. Different architectures will deal differently with multiple requests under load. To find the machines' limits we have to test them in different environments.

1. Linear - machines will be able to handle requests are they are received without the need to scale.

2. Slow Growth - the rate will increase slowly, until it is required for another machine to start working.

3. Fast Growth - the rate will increase faster, another machine needs to start working urgently before the webserver starts losing requests.

In our solution we use AWS Elastic Load Balancing (ELB) to distribute requests to the application servers. This allows us to achieve greater levels of fault tolerance in the application and keep the availability levels in situations of higher incoming traffic. It can detect unhealthy instances and reroute traffic across the remaining healthy instances.

Together with ELB, we also have configured AWS Auto Scaling Groups, which allow to scale automatically EC2 capacity up or down, according to conditions we define. So during demand/traffic spikes the number of EC2 instances can be increased to maintain application performance and availability and decrease capacity during lower demand to reduce costs.

4.1 Elastic Load Balancing Configuration

There are some configuration parameters that we need to set in ELB, to better fit our application requirements:

- **Ping Target** - the route in the webserver for which ELB should send the health checks. This route should be used by requests that access the application database, so this way we also check for database connectivity. In our case the route is **HTTP:80/unseen**;

- **Response Timeout** - time to wait when receiving a response from the health check. We defined it to **5 seconds**;

- **Health Check Interval** - this value should be low enough, so that when an instance goes down the load balancer reacts fast (this is critical during high traffic situations), but also high enough so that there isn't a health check overload to the instances. We setted it to **15 seconds**;

- **Unhealthy Threshold** - the number of consecutive health checks before declaring an instance dead. When setting this value, we must take into consideration that the value shouldn't be very low, because in high load situations, there could be some request loss and we don't want ELB to declare an instance dead after one or two timed out health check. We set it at **4 consecutive health check failures**;

- **Healthy Threshold** - the number of consecutive health checks before declaring an instance healthy. When setting this value, we must take into consideration that this value will affect the time that will take to make the instance available to the ELB. The value should be adjusted, so that during high load scenarios the instance will be quickly available and in fully working conditions. We set it **4 consecutive health check successes**;

4.2 Auto Scaling Group Configuration

The configuration of the Auto Scaling Group is very straightforward. First it is necessary to create a launch configuration. This launch configuration will be used by all instances that will be created in the auto scaling group. We basically used a AMI (instance image) that already contained our application pre-configured, used a security group that allowed incoming HTTP traffic and defined that all our instances would be **m3.medium**.

The second step is to define the Auto Scaling Group parameters:

- **Group name** - bw-autoscale-launch;
- **Group Size** - The number of instances the group should always have at any time. **In our case - 1 instance**;
- **Network** - vpc-73422f1b(172.31.0.0/16)(default);
- **Subnet** - the default ones in us-west-2 region: **us-west-2c, us-west-2a, us-west-2b**;
- **Load Balancing** - Receive traffic from Elastic Load Balancers;
- **Health Check Grace Period** - this value should be approximately two times the machine booting time. We setted it to the default **300 seconds** value;
- **Monitoring** - Enable CloudWatch detailed monitoring;

Finally is necessary to configure the auto scaling policies. This auto scaling polices take advantage of Cloud Watch alarms. In our case we used the following:

- Scale between 1 and 5 instances. (minimum and maximum size of group)
- **Increase Group Size**
  - **Execute policy:** Whenever: **Maximum** of **CPU Utilization** is **>= 70 percent** for at least: **1** consecutive period of **5 minutes**
  - **Take the action:** Add 1 instance
  - **And then wait:** 300 seconds before allowing another scaling activity
- **Decrease Group Size**
  - **Execute policy:** Whenever: **Average** of **CPU Utilization** is **<= 20 percent** for at least: **1** consecutive period of **5 minutes**
  - **Take the action:** Remove 1 instance
  - **And then wait:** 300 seconds before allowing another scaling activity

5 Testing

In our tests we simulate two different situations:

- **Burst Requests**-send several concurrent requests at a burst-rate in the minimum possible time.
- **Growing Rate Requests** - send increasing number of concurrent requests in a given time interval.

5.1 Burst Requests Simulation

In this simulation case, we used Apache Bench (version 2.3) to send several concurrent requests at a burst-rate. This simulation case is useful for testing the Cloud Watch alarms and check if the auto scale is working correctly. This simulation doesn't represent a real world usage, because there is never going to be a case in which a user burst requests without a minimum interval between them (for example, the user needs to read a webpage before requesting another page).

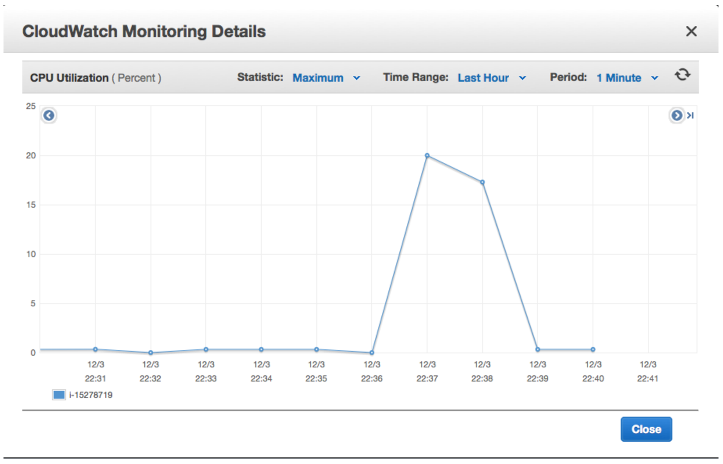
- **Simulation 1: 1000 requests - 100 concurrent users**

```
$ab -r -n 1000 -c 100 http://birdwatch-lb
   -525863057.us-west-2.elb.amazonaws.com/test
```

  - **Results**

Concurrency Level:	100
Time taken for tests:	22.596 seconds
Complete requests:	1000
Failed requests:	0
Requests per second:	44.26 [per/sec] (mean)
Time per request:	2259.615 [ms] (mean)
Time per request:	22.596 [ms] (mean, across all concurrent requests)

  - **Instance CPU Usage**



CloudWatch Monitoring Details

CPU Utilization (Percent) Statistic: Maximum Time Range: Last Hour Period: 1 Minute

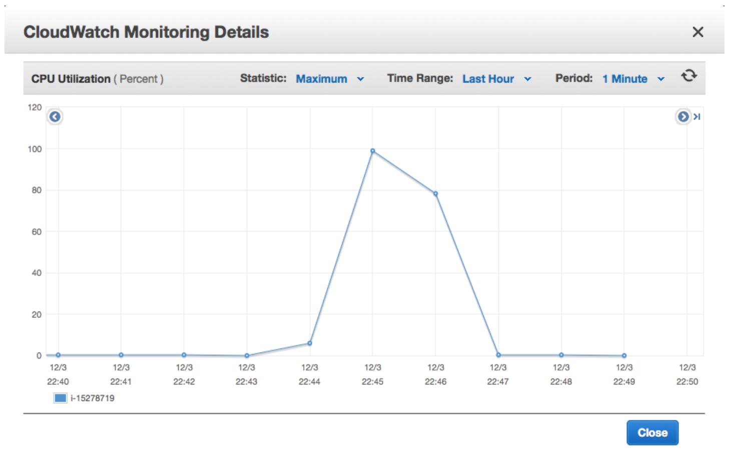
- **Simulation 2: 5000 requests - 100 concurrent users**

```
$ab -r -n 5000 -c 100 http://birdwatch-lb
   -525863057.us-west-2.elb.amazonaws.com/test
```

- **Results**

Concurrency Level:	100
Time taken for tests:	111.260 seconds
Complete requests:	5000
Failed requests:	0
Requests per second:	44.94 [per/sec] (mean)
Time per request:	2225.209 [ms] (mean)
Time per request:	22.252 [ms] (mean, across all concurrent requests)

- **Instance CPU Usage**



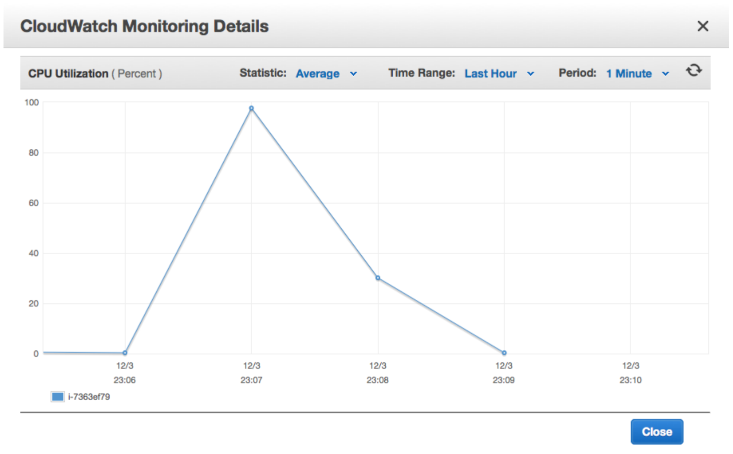
- **Simulation 3: 10000 requests - 1000 concurrent users**

```
$ab -r -n 10000 -c 1000 http://birdwatch-lb
    -525863057.us-west-2.elb.amazonaws.com/test
```

- **Results**

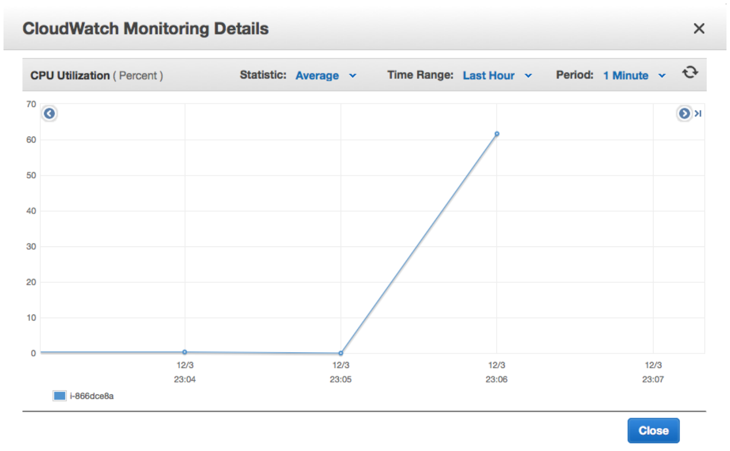
Concurrency Level:	1000
Time taken for tests:	115.769 seconds
Complete requests:	10000
Failed requests:	47
Requests per second:	86.38 [per/sec] (mean)
Time per request:	11576.892 [ms] (mean)
Time per request:	11.572 [ms] (mean, across all concurrent requests)

- **First Instance CPU Usage**


CloudWatch Monitoring Details

CPU Utilization (Percent) Statistic: Average Time Range: Last Hour Period: 1 Minute

– **Second Instance CPU Usage**

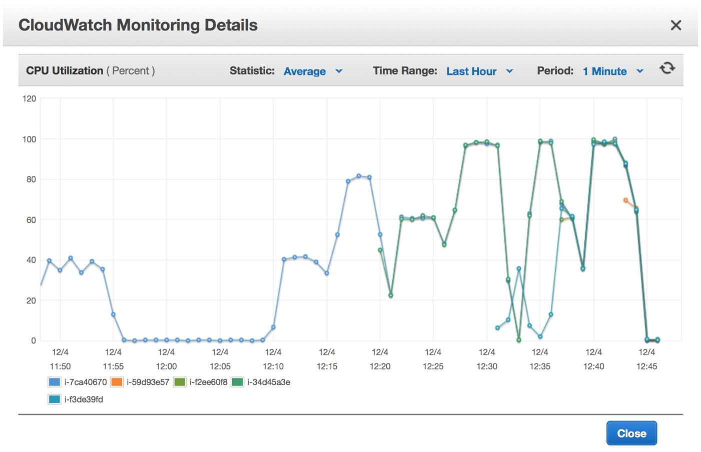

CloudWatch Monitoring Details

CPU Utilization (Percent) Statistic: Average Time Range: Last Hour Period: 1 Minute

5.2 Growing Rate Requests Simulation

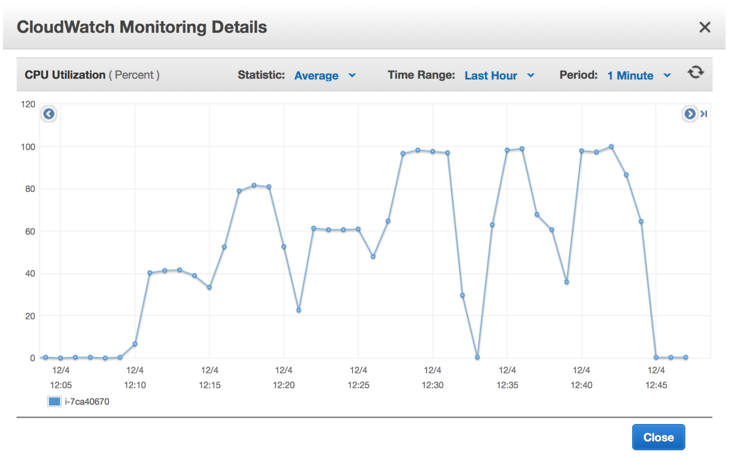
In this simulation we tried to test a more realistic traffic pattern. We used Siege, and have done several requests for 100, 200, 300, 500, 600 and 1000 concurrent users, each during 5 minutes with a delay between each request of 1 to 10 seconds. Before increasing the number of concurrent users in Siege, there is an interval of 30 seconds.

- All Instances CPU Usage


CloudWatch Monitoring Details

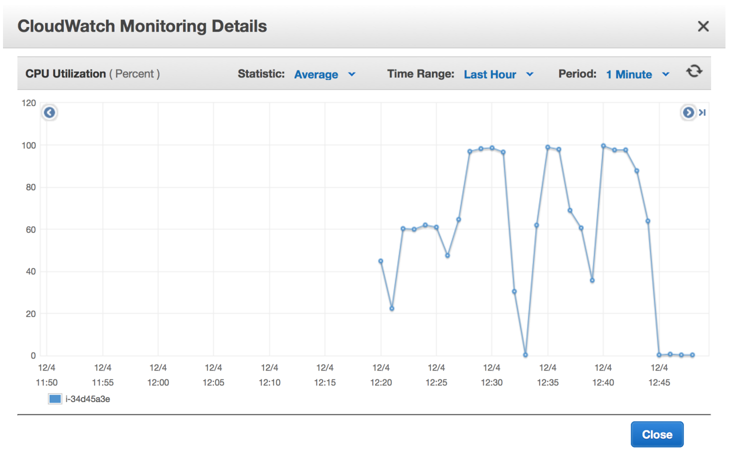
CPU Utilization (Percent) Statistic: Average Time Range: Last Hour Period: 1 Minute

- First Instance CPU Usage


CloudWatch Monitoring Details

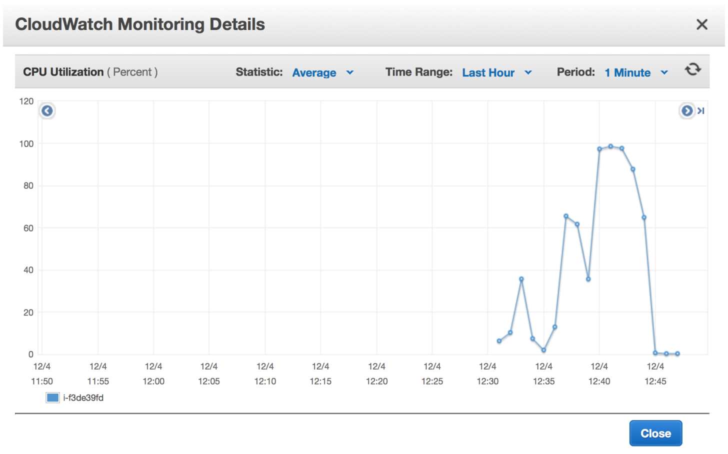
CPU Utilization (Percent) Statistic: Average Time Range: Last Hour Period: 1 Minute

- Second Instance CPU Usage


CloudWatch Monitoring Details

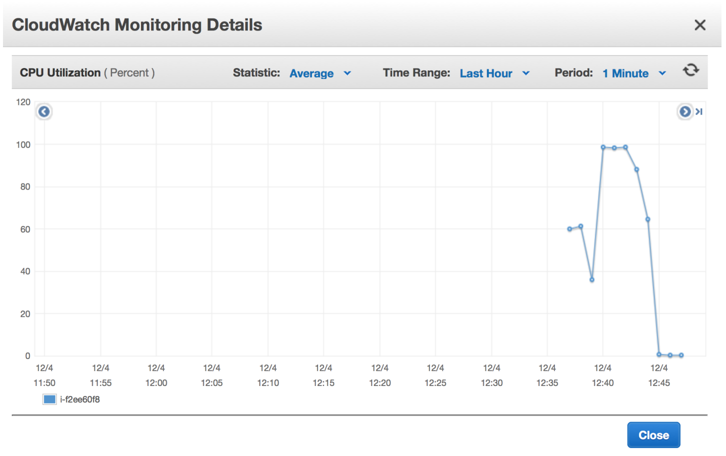
CPU Utilization (Percent) Statistic: Average Time Range: Last Hour Period: 1 Minute

- Third Instance CPU Usage


CloudWatch Monitoring Details

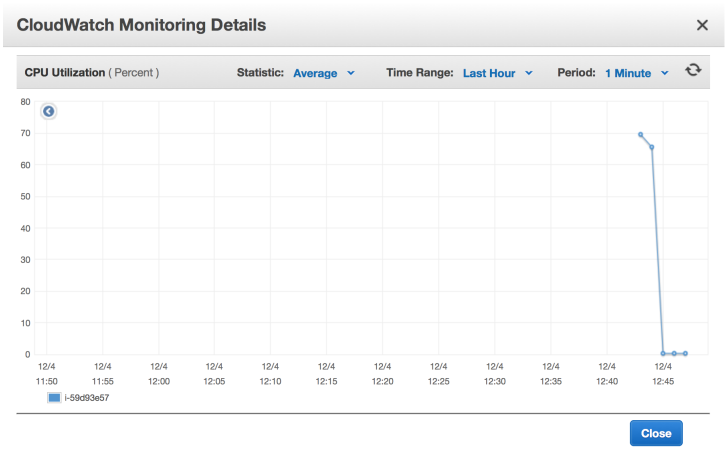
CPU Utilization (Percent) Statistic: Average Time Range: Last Hour Period: 1 Minute

- Forth Instance CPU Usage


CloudWatch Monitoring Details

CPU Utilization (Percent) Statistic: Average Time Range: Last Hour Period: 1 Minute

- Fifth Instance CPU Usage


CloudWatch Monitoring Details

CPU Utilization (Percent) Statistic: Average Time Range: Last Hour Period: 1 Minute

- Siege Log Results

100 concurrent users

Transactions (requests):	5495
Availability:	100.00%
Elapsed Time:	299.32 secs
Response Time:	0.43 secs
Transaction rate:	18.36 trans/sec
Successful transactions:	5495
Failed transactions:	0
Shortest transaction:	0.40

200 concurrent users

Transactions (requests):	10892
Availability:	99.95%
Elapsed Time:	299.98 secs
Response Time:	0.47 secs
Transaction rate:	36.31 trans/sec
Successful transactions:	10892
Failed transactions:	5
Longest transaction:	2.28
Shortest transaction:	0.40

300 concurrent users

Transactions (requests):	16402
Availability:	100.00%
Elapsed Time:	299.97 secs
Response Time:	0.44 secs
Transaction rate:	54.68 trans/sec
Successful transactions:	16402
Failed transactions:	0
Longest transaction:	5.48
Shortest transaction:	0.40

500 concurrent users

Transactions (requests):	29934
Availability:	99.93%
Elapsed Time:	299.50 secs
Response Time:	0.98 secs
Transaction rate:	99.95 trans/sec
Successful transactions:	29934
Failed transactions:	22
Longest transaction:	3.08
Shortest transaction:	0.40

1000 concurrent users

Transactions (requests):	53790
Availability:	99.96%
Elapsed Time:	299.63 secs
Response Time:	0.55 secs
Transaction rate:	179.52 trans/sec
Successful transactions:	53790
Failed transactions:	20
Longest transaction:	5.52
Shortest transaction:	0.40

6 Evaluation

6.1 Burst Request Rate Simulation Evaluation

In this simulation the scaling worked correctly and scaled up to two instances, when there was 1000 concurrent users sending requests. When testing for 100 concurrent users sending in total 1000 requests and 5000 requests a single machine was enough to process all sent requests. But by analyzing the first instance CPU usage in simulation is possible to see that that the instance was under heavy load (more that 80% CPU usage) and triggered the launch of a new instance. As all three simulations where done without any time interval between them, so when the 10000 request and 1000 concurrent users test was done, the second instance hadn't been launched, so there was some failed requests.

6.2 Growing Request Rate Simulation Results

In this simulation scenario there was more accurate results, since it better represents a real world scenario. In some of the Siege executions, it was reported that there wasn't total availability and there was some failed transactions (requests). This is due to the fact that the tests were executed in a single machine, and at some point there were socket creation problems thanks to the fact that is hard for a single machine to handle 1000 concurrent threads sending requests (mainly bandwidth-wise).

In order to do some more intensive load tests with higher fiability, some of kind of distributed load testing tool (like Tsung) could be used that can scale the number of concurrent virtual users to several thousands.

In this test scenario all of the five instances were launched, when it was necessary to handle 1000 concurrent users, and so the efficient load balacing of the system was proved.

7 Conclusion

In this project it was necessary to build a modern web application with several components, that could scale with a growing userbase.

By building a MapReduce application and using Amazon Elastic MapReduce to run it, we were able to quickly process log files and build a database schema with that processed data. This allowed us to query DynamoDB tables with the web application, requiring minimum resources.

When load balancing and auto scaling application instances, we could prove that our solution could efficiently scale to handle (probably) thousands of concurrent users. We also observed, that in order to have more reliable load tests we need to use some kind of distributed load testing tool. We also concluded that Amazon AWS services are an excelent option to build a application that can easily scale with a growing userbase.