TÉCNICO LISBOA

Cloud Computing

Cloud Computing

Group 40
Final Report

77875 – Filippo Campagnaro

46425 – Rodrigo Bruno

67074 – Samuel Bernardo

Master's Telecommunications and Informatics Engineering

December 13, 2013

Contents

Introduction

The first goal of the cloud computing project is to develop a mobile phone network data processing and search system. For achieving that goal we use a MapReduce application to process large logs of information retrieved from phone cell towers. The processed information is then stored on the database that will be latter accessed from a website.

The other goal for this project it so setup a load balancer and configure auto-scaling for the web servers. By configuring a proper load balancer and auto scaling mechanism, we should be able to grow the number of web servers when we detect more requests and drop web servers when there are few requests.

Within the next sections we will explain how we implemented the MapReduce application (detailing the most important factors of our solution), give a fast overview over the database organization and end up explaining the load balancing and auto scaling mechanisms.

The final sections will be dedicated to the evaluation and conclusions. Evaluation will be divided into: evaluating the MapReduce application and evaluating the load balancing and auto scaling.

1 MapReduce Implementation

The main goal for our MapReduce application is to efficiently process large sets of data and sent them to a remote database. In order to achieve it, we need to analyze what information is given as input and what information is expected for the output. The very big concern we had always in mind is that this application should do almost all the work and store an almost ready answer into the database. This is fundamental in order to achieve higher scalability (since the amount of processing inside the webserver and database will be very small).

The specification that we must cope with requests three types of queries:

- get the sequence of cells where the phone was in a particular day;
- get the set of phones present in one cell in a particular day and hour;
- get the number of offline minutes for a particular phone and day.

Before going into implementation details, it is important to note the following: the first and the third queries can be processed in parallel only for different days and the second can be processed in parallel for different cells within one day.

Given the fact just presented, our implementation is very simple:

- **Map**: the map implementation takes all the lines and creates a different keys and values according to the event type. Events are grouped into three different sets of keys (that will be

handled differently by the combiner and reducer) according to the event type;

- **Combiner**: the combiner is used to process information for the second query (the one that can be processed in parallel for different cells);
- **Reducer**: the reducer does the final aggregation of all the values. It also supports incremental logs (where by incremental logs we assume that are new logs from different celss). To provide the correct information, the reducer asks the database (whether it is an SQL or DynamoDB) to give the current values for the query. With this information, the reducer can produce the correct information that will then uploaded to the database.

Another interesting feature that we used was secondary sorting. Secondary sorting enables us to perform value sorting and get all the values for the same key in order. This saves a lot of effort in the reducer since all the events are grouped and all the values inside each group are ordered. Our final implementation is prepared to work in the RNL Hadoop installation and is able to produce output for three output storage systems: local, postgreSQL and Amazon DynamoDB.

2 Web Site

We developed a web application to allow simple queries over the output data of the MapReduce application. The web application is developed using php and javaScript. Those languages are the most used for simple web applications to allow the user to access and querying a database. In fact in our first prototype we store the results of the MapReduce application in a postreSQL database. Anyway with php is also simple accessing to a BigTable on DynamoDB, as we did for the final application. Three kind of queries are permitted:

2.1 User Interface

With the website the user can access the data inserting the request specifications in the input fields. To permit this there are three form blocks in the interface, one for each query. After the right insertion of the specifications in the input fields the request is sent to a php page due to accessing the data stored in a webserver. The web interface is shown in figure 1 available at: server url: `ec2-54-200-205-56.us-west-2.compute.amazonaws.com`, load balancer url: `ec2-54-201-119-18.us-west-2.compute.amazonaws.com/`.

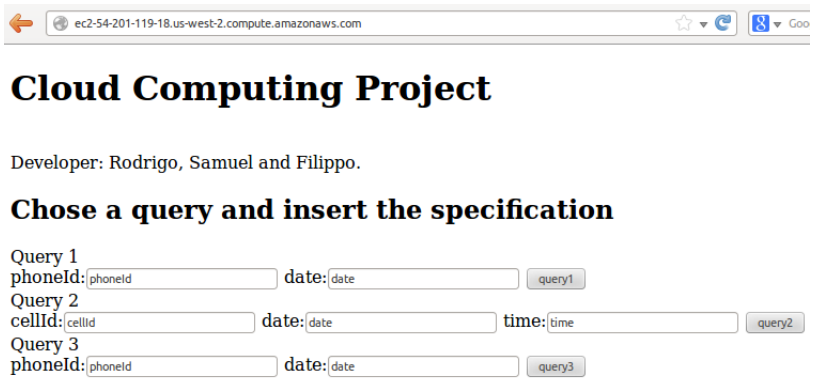

Figure 1: Illustrative screenshot of index.html

2.2 PHP: connection with data

For retrieving the data stored in the web server (in a DynamoDB BigTable) we use a php page. To allow stress test with software like 'siege' or 'jmeter', the querying specifications are passed to this page with the GET method, directly written in the URL of the page. Getting the attributes, the data are retrieved by the server. For the second and the third query the retrieved data are directly shown to the user, instead for the first query the data are stored in an hidden field for the last elaboration at the client web browser. The listing of the php page is attached in the code 2.2 and an example of the web application is shown in figure 2, available using the `http://ec2-54-200-205-56.us-west-2.compute.amazonaws.com/query.php?id=C1&date=2013%2F10%2F22&time=08&submit=query2`

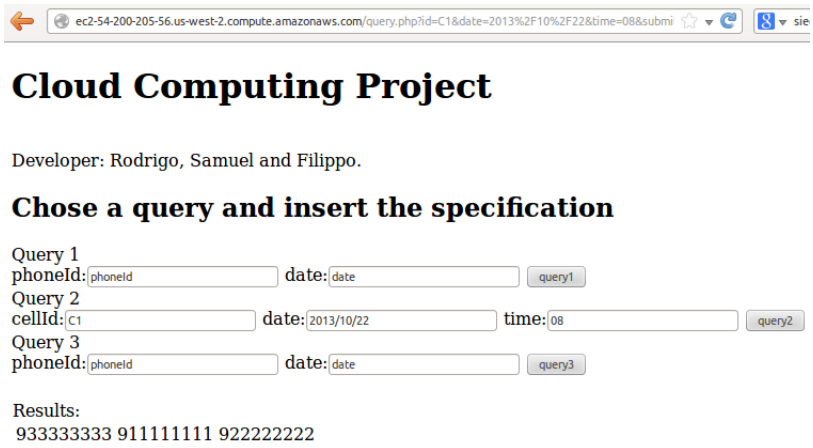

Figure 2: Second query in query.php

```
query.php
<?php
//AWS connection
require '../aws/aws-autoloader.php';
use Aws\DynamoDb\DynamoDbClient;
$client=DynamoDbClient::factory(array(
  'key' => '...',
  'secret' => '...',
  'region' => 'us-west-2'
));
//FORM GET STUFF
$submit = $_GET['submit'];
$date = $_GET['date'];
$id = $_GET['id'];
$dateId=$date."-".$id;
$time = "";
$sql = "";
//DynamoDb getItem
if ($submit!="query2"){
  if($submit == "query1"){
    $result = $client ->getItem(array(
        'ConsistentRead' => true,
        'TableName' => 'CN_logs',
        'Key' => array('date-id' => array('S'=>$dateId)),
        'AttributesToGet' => array('value')
        ));
  $output = $result['Item']['value']['SS'];
  }
  else{
    $result = $client ->getItem(array(
      'ConsistentRead' => true,
      'TableName' => 'CN_logs',
      'Key' => array('date-id' => array('S'=>$dateId)),
      'AttributesToGet' => array('number')
    ));
    $output = $result['Item']['number']['SS'];
  }
}
else{
```

```
$time = $_GET['time'];
$dateId=$dateId.":".$time;
$result = $client ->getItem(array(
      'ConsistentRead' => true,
      'TableName' => 'CN_logs',
      'Key' => array('date-id' => array('S'=>$dateId)),
      'AttributesToGet' => array('value')
      ));
$output = $result['Item']['value']['SS'];
}
echo "<div id='queryResult'>Results: <input type='hidden' id='queryNumber' value='$submit'/>";
echo "<div id='out'>";
if($submit!="query1"){
  echo "<table>";
  for ($i=0;$i<sizeof($output);$i++)
    echo "<tr><td>$output[$i]</td></tr>";
  echo "</table></div>";
}
else{
  echo "</div><input type='hidden' id='campo' value='";
  for ($i=0;$i<sizeof($output);$i++)
    echo $output[$i];
  echo"'/>";
}
echo "</div>";
?>
```

2.3 JavaScript: last elaboration

As we saved the data in DynamoDB in a common table, the first query data needs one more elaboration step before be shown to the user. In fact we decided to elaborate the network logs by an unique MapReduce application and store them in the most generic way in a common BigTable. To do this we decide to store int the fields of the cells visited by a phoneId in a day also the time for having the sequence of the cells order by time. The first query require only the ordered sequence of the cells, not the time. That sequence is stored in a hidden field in the web page and by a javaScript function the time is deleted from the sequence before been shown to the user. This function is called when the page sent by the server is loaded in the web browser of the user, for doing less work as possible in the server.

```
<body onload="init();">
Results:
<input id="queryNumber" type="hidden" value="query1">
<div id="out"> C2 C1</div>
<input id="campo" type="hidden" value="00:24:00;C2 14:12:15;C1">
```

```
function init(){
if(document.getElementById("queryNumber").value=="query1")
document.getElementById('out').innerHTML=elaborateString(document.getElementById('campo').value);
}
function elaborateString(inputString){
  input=inputString.split(' ');
  output="";
  for (index=0;index<input.length;index++){
    item = input[index];
    item=item.split(";");
    output+=" "+item[item.length-1];
  }
  return output;
}
```

3 Load balancing and auto scaling

Regarding the load balancing and auto scaling mechanisms we decided to use Amazon services since it would be impractical to use RNL to launch and terminate machines automatically.

The big challenge here is to be able to grow and decrease the number of worker machines (in this case, web servers) according to the number of requests or load. In order to do this we created an auto scaling group with one load balancer, placed some scaling policies and setted up some alarms.

After some experiments, we noted that the most significant factor is the CPU usage percentage. This is the only information available that we believe that truly explains the high load on the servers. As that being we placed two scaling policies and two alarms that are triggered. The fist policy adds one instance if the average CPU utilization (from the auto scaling group) is bigger than 35% for more than one minute and another to remove a machine if the average CPU utilization is lower than 10% for more than one minute. The time between scaling activities is 30 seconds.

Our scaling policies react very fast and try to add as soon as possible more machines. Although it can be a waste of resources for ephemeral waves of requests we decided to leave it like this for safety (this way we assure that we will loose less requests). For the health status we used 60 seconds interval with an healthy threshold of 2 and an unhealthy threshold of 4 (to avoid loosing machines that are just too busy to respond).

4 Evaluation and Plots

We dedicate this section to testing and evaluating our project. We start with some performance evaluation on our MapReduce application and then we move to the AWS load balancing and auto scaling mechanisms.

4.1 MapReduce

For evaluation purposes we created big logs for stressing our application. Our application was run for multiple time with different log sizes and for different storages: HDFS and SQL DB. For identifying each test sample we use the number of cells (the number of days ans phones is constant for each cell). For tests with more than 8 cells we decided not to run them on the SQL DB since it was taking too long and could be considered Denial of Service.

By analyzing the table, we can two facts:

- the overhead of running MapReduce targeting remote storage is overwhelming;
- the gains for using the combiner are huge (we are able to reduce almost 2/3 of the records).

Table 1: Map Reduce Performance Results

Test Sample (cells)	Map Input (records)	Combine Input (records)	Reduce Input (records)	Reduce Output (records)	Time HDFS (s)	Time SQL (s)
2	40.500	43.740	17.012	12.150	18	285
4	121.500	131.220	51.032	36.450	19	775
8	283.500	306.180	119.072	85.050	21	1737
16	607.500	656.100	255.151	255.151	25	-
32	1.255.500	1.355.940	527.314	376.650	34	-
64	2.551.500	3.827.256	1.071.636	765.450	52	-
128	5.143.500	7.715.265	2.160.285	1.543.050	113	-
256	10.327.500	15.491.283	4.337.583	3.098.250	161	-

5 Load Balancing and Auto Scaling

Regarding the evaluation of the load balancing and auto scaling it is important to note that: 1) we used the siege tool to stress the web site; 2) only micro instances were used; 3) the minimum number of instances was 5; 4) we used a very basic disk image and only added the php and httpd.

Table 2: Stressing Results for the First Test

Test	Transactions	Availability (percentage)	Time (s)	Avg. Response Time (s)
First	15945	99.81	1199.7	0.49
Second	7978	99.86	1199.96	0.49
Third	4028	99.90	1199.80	0.49

Figure 3: First straight test of the load balancer website

Figure 4: Second straight test of the load balancer website

Figure 5: Third straight test of the load balancer website

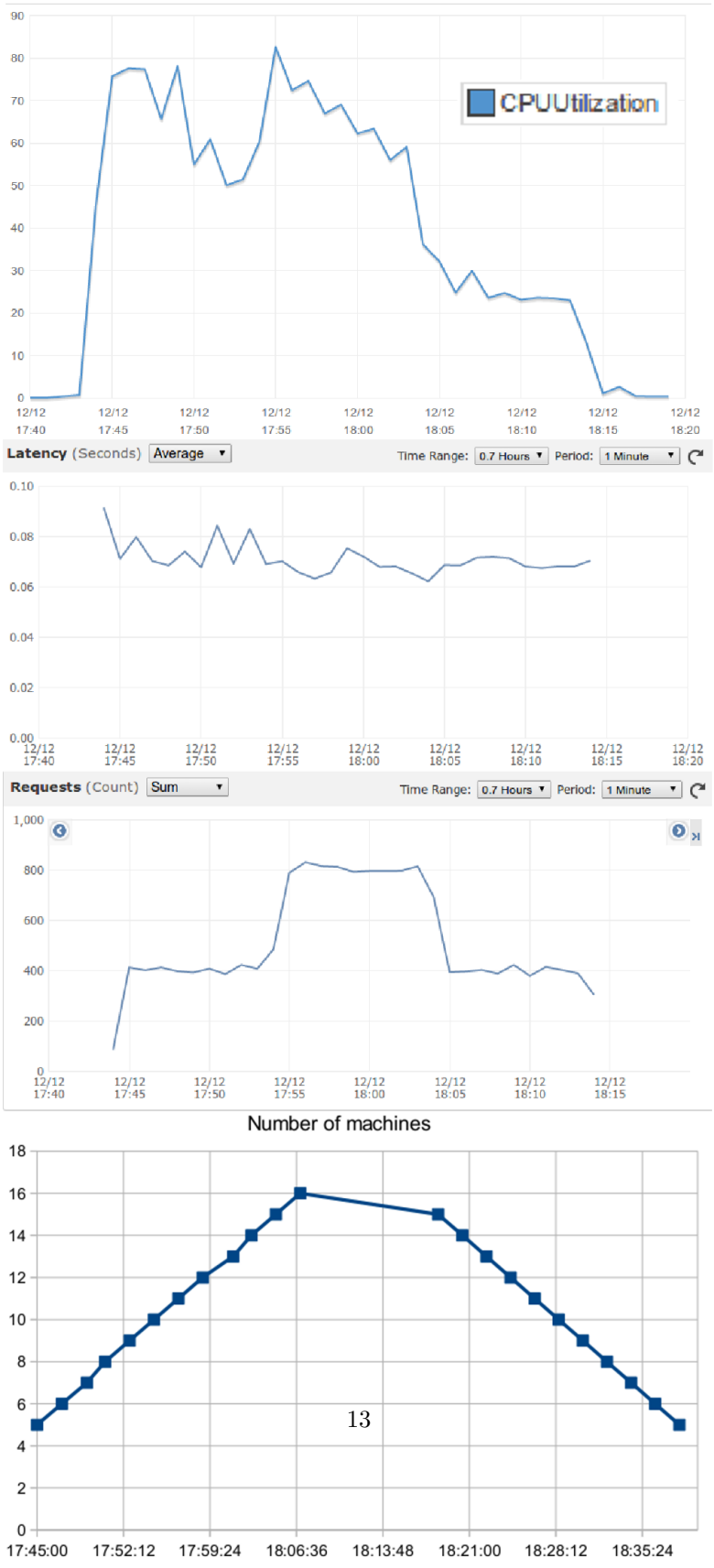

Figure 6: Fourth straight test of the load balancer websitei

All the four tests are very similar, we just increased the number of requests in parallel. The fourth one is composed by three sequences of different request rates. By analyzing all the give information it is possible to state that the mechanisms placed in practise (policies and other scaling rules) produced the expected behaviour by always trying to minimize the CPU load or minimize the number of machines.

6 Conclusion

Finally we would like to say that we believe that to have fulfilled all the goals proposed by the project specification with good results. Both our MapReduce application and the load balancing and auto scaling mechanisms are effective and efficient.