Project - week 2

REST interface and MapReduce

February 23, 2017

Students:
Mees Kalf
10462074

Coordinator:
dhr. L. Torenvliet

Course:
Modern Databases

Catalog number:
5062MODA6Y

1 Introduction

Building a B+ tree in the first week of this YAMR project we are now expanding into a more real database like application. This week we'll be implementing a REST interface that supports the GET, PUT, POST and DELETE requests and proper handling. Due the fact of not working in a pair and needed to make a fast decision, and I never heard of Tornado before, I choose to implement the REST-interface in Flask[1].

2 REST implementation

Installing flask was quite easy, just a sudo apt-get install command. First I've done a lot of reading of documentation on Flask to know with what I'm working, of which much wasn't needed since we already have the functions of the B+ tree and database. I always have quite some problems with starting from scratch, so i just started with getting Flask running and a simple GET request. The Flask app is run on port 8080 and accessible through $http://localhost:8080/documents$, on which set operations are preformed and through $http://localhost/documents/<int:id>$ for separate documents, the id has to be an integer.

2.1 Retrieval, creation, updates and deletion of documents

Retrieval The retrieval of a document is only possible when knowing it's key. The path of the request should look like $/document/<int:id>$,

where $< int : id >$ should be an integer representing the key. So a valid request would be:

curl -i -X GET http://localhost:8080/document/1

The flask application sees the path and a GET request which calls the appropriate function which retrieves the document that's associated with key 1 in the database file, by using the B+ tree. Then the document is inflated so the data is use able.

Creation The creation of a document is done with a POST request using the $-d$ command to add a body to the request. The body MUST be a json, other formats won't be accepted. A POST request look like the following command:

curl -i -X POST -H "Content-Type: application/json" \
-d "'name':'dhr.Torenvliet' http://localhost:8080/document

The json will be compressed and put in the database with a unique key.

Updates Updates are done with a PUT request alsmost the same as a POST request. The key value pair you want to change MUST be in json format, after the REST-interface receives a PUT request for a single document it changes the value of the key and puts it back in to the database. This is done with the following command:

curl -i -X PUT -H "Content-Type: application/json" \
-d "'name':'dhr.Kalf' http://localhost:8080/document/1

If we would now use a GET request to retrieve the name of the first document it will state "dhr.Kalf" instead of "dhr.Torenvliet"

Deletion Deletion is done through a DELETE request. Since we we're not asked to implement a delete in the B+ tree we should alter it or as I did, replacing the entire document with the string "DELETED", this way the document is no longer retrievable. The real deletion will take place in when compacting the database, this will be discussed later in this report. To delete a document I specified the following command deleting the document with id 1:

curl -i -X DELETE -H "Content-Type: application/json" \
http://localhost:8080/document/1

2.2 Retrieval of sets of documents.

The retrieval of the sets of documents is done by sending a GET request without a specification of the key, then we just return every document in the database. The request is specified as:

curl -i -X GET http://localhost:8080/documents

2.3 Execution of map-reduce queries

Being so busy getting every to work I started with the map-reduce implementation without reading the syllabus correctly. This caused quite a headache. After another group pointed out the Asteval package in the syllabus it still wasn't working. Only after literally copying the Asteval file from the syllabus, I got some results. First the dbmap() function is called which checks a document for data specified in this script. This python file should be in the same directory as Flask is running, in my situation env/bin/. The dbmap() function returns a list with the keys, and optionally values, with which a new temporary database is build. This temporary database is used for the reduce function, dbreduce() which returns a dictionary. The dictionary contains all keys with the calculated, by the reduce function, values. The temporary database is then removed. The map-reduce query is specified as follows:

curl -X POST -H "Content-Type: application/json" \
http://localhost:8080 /documents/mapreduce -d 'mapreduce.py'

Make sure the file 'mapreduce.py' is present on the server with the dbmap() and dbreduce() functions.

2.4 Compaction of the database

Compacting the database by removing all deleted entries to get a clean database is done by creating a new database and copying only the entries which aren't equal to "DELETED". When the transfer is completed the temporary database can be copied over the current database and the temporary database can be removed.

3 Database testing

To test the database and REST-interface we are testing it with the NVD, National Vulnerability Database using a SAX parser.

3.1 Install the database

Installing the database is done running app.py, appendix A, which also immediately start the Flask REST-interface at port 8080. I run the database as suggested in the virtual python3 environment.

3.2 Install the NVD database

Installing the NVD database [3] is done using our own just implemented REST-interface. To do this we need to convert the xml file to json which is done through a SAX parser with a little help of smihica's xml to jason script

[4], I used this script a year ago for another course but since we are required to use python 3.4 to run all the installed files mentioned in chapter 2 of the syllabus, I had to make make some major adaptions, looking back it might be better to first investigate what the possibilities are the next time. Using a simple script with pycurl sending a POST request for every exploit, in json form, each document, exploit, is imported to the database. Since importing the NVD crashed every time I first had to decode the entire database so it wouldn't contain strange characters.

3.3 Map-reduce queries

The query for a map-reduce function is previously mentioned. A python file needs to be present at in the same directory as from where the flask application is running. This python file should contain a dbmap() and dbreduce() function. Using the previously mentiond query the file's dbmap() and dbreduce() functions will then be executed on each document.

3.3.1 Top 10 vulnerable Products

The map_reduce_product.py in appendix B shows the code used as map and reduce function for finding the top 10 most vulnerable products. Since I imported every document as a json we can also read every document as a jason, which is automatically a dictionary in python. Looking at the XML file of the NVD we can spot the the entry $vuln : vulnerable - software - list$ which contains a $vuln : product$. These products are needed to be evaluate in the map function. Since every XML key is translated by the XML to json script [2] to a json key we need these keys to access the products in the document in the database.

By first checking if $vuln : vulnerable - software - list$ is a key and then checking the key $vuln : product$ we get a list of all products over which we can loop. At first this didn't work and the error parsing from the map-reduce script didn't prompt any usable information server-side or cliënt side this took me a long time. After trying different simple map-reduce scripts on some self-made simple documents and different functions on the NVD database which did work I couldn't find what went wrong. Inspecting the XML again and putting a lot of write statements I found that when there was only one $vuln : product$ entry the XML to json script [2] coveted it to just a key value pair instead of a key with a list of values, on which the map function crashed. Fixing this by checking if it's a list instance or not did the job. A entry of the product list looks like $cpe : /a : microsoft : jscript : 5.8$ splitting on $' :'$ we can see the fourth item is the product. A unique list of these products is then returned, counting an exploit only once for a product.

The reduce function creates a dictionary and simply loops over all lists generated by the map function adding every list entry to the dictionary, when a list entry, key, is already in the dictionary the value is incremented, when

the entry isn't present it's added to the dictionary as key with value 1. By sorting the dict as list and then jsonifying the list it it's returned to the requester.

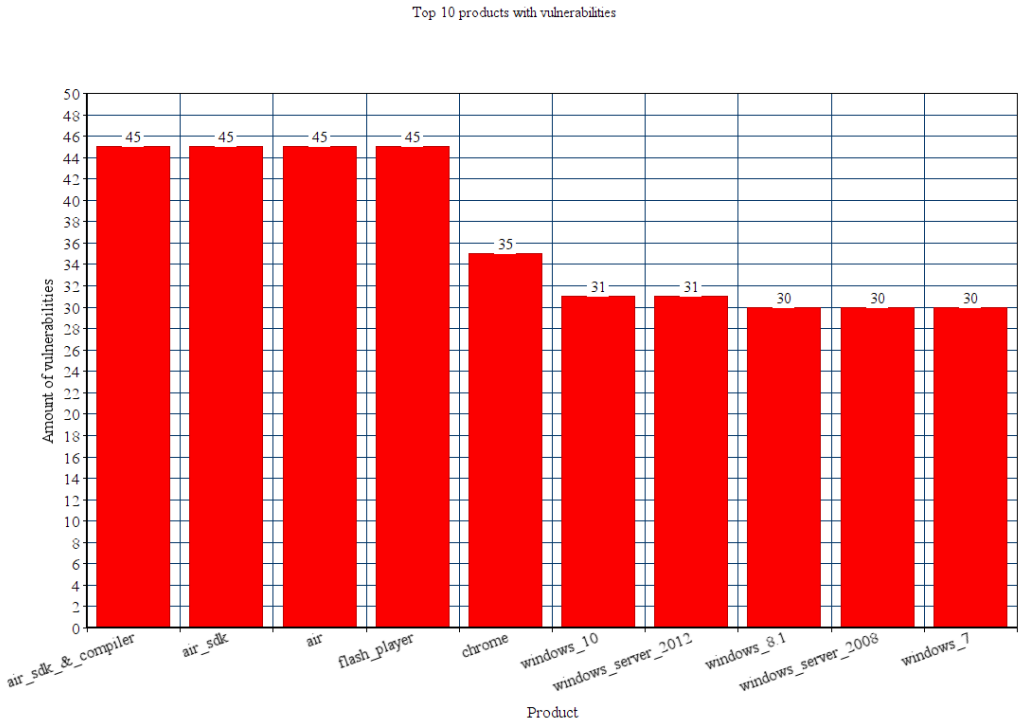

Figure 1: Top 10 products with vulnerabilities

3.3.2 Top 10 vendors with vulnerabilities in products

The map function is actually quite the same as finding the most vulnerable products, only now we have to look at the vendor instead of the product, the map$_{r}educe_{v}endors.pyisshowninappendixC.Lookingagainattheproductlistvalue$cpe:/a:microsoft:

3.3.3 Compare the results with [5]

Our results do differ a bit from the given results [5], although 7 of our top 10 vendors are also in the top 10 of the given results [5]. It looks like we are missing some data, since the results of our top 10 vulnerable products have the same descending order as the given results [5] and the amount of vulnerabilities don't differ a lot. A possible explanation could be the fact that they use multiple sources for the exploits, e.g. http://www.exploit-db.com and Metasplit.

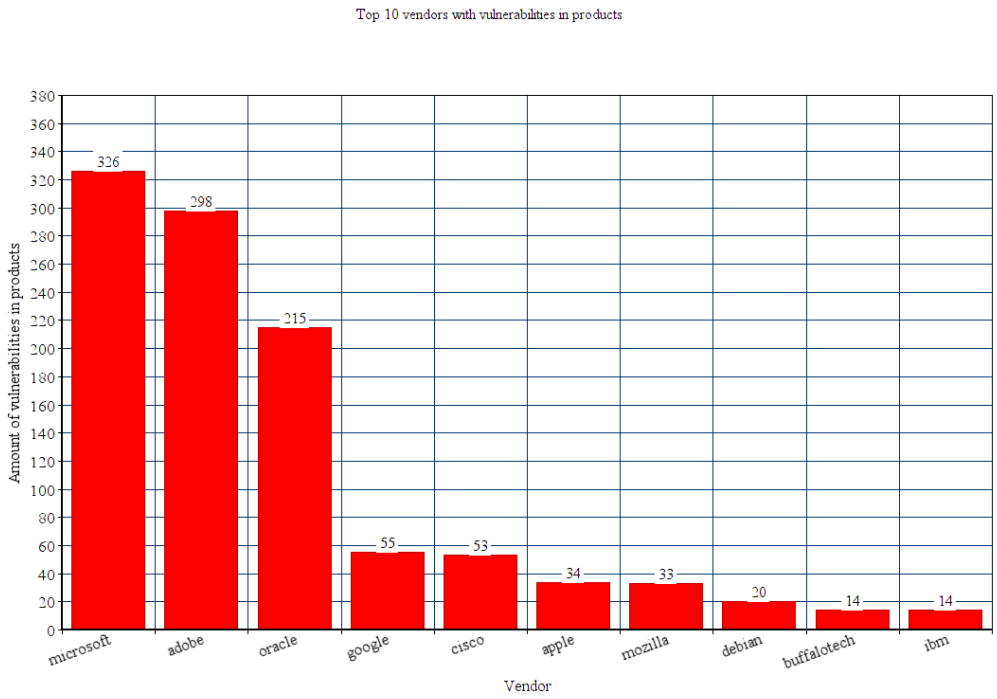

Figure 2: Top 10 vendors with vulnerabilities in their products

4 Conclusion

Looking back at the implementation a lot went wrong. Loading the database of 2015 took so long and used way to much space I used the database of 2016. Still restarting the database went slow. Using snappy as mentioned in the syllabus gave a huge compression rate, from 820MB to 320MB, which made restarting the database a lot faster. The Astreval package didn't work, only when exactly copied from the syllabus and after getting it to work I had problems retrieving the errors from the map and reduce functions which took a lot of time and print statements to fix. I think just one "werkcollege" would've helped a lot, and maybe a partner, now I frequently had to ask vague aspects of the questions and/or Flask, although most groups were using Tornado, to other groups.

5 References

[1] http://flask.pocoo.org/
[2] https://gist.github.com/smihica/2792662
[3] http://nvd.nist.gov/download.cfm
[4] https://gist.github.com/andreif/6088558
[5] http://www.cvedetails.com/top-50-products.php?year=2016