REST interface + MapReduce

Bär Halberkamp, Steven Raaijmakers

March 2017

1 Inleiding

Na het implementeren van de b+tree in een documentstore, is het deze week tijd de database verder uit te bouwen. Hierop bouwen we deze week verder door een interface te maken, waarmee gecommuniceerd kan worden met de documentstore. Daarnaast voegen we ook andere functionaliteit toe aan de database in de vorm van een mapreduce

2 Probleemstelling

Het doel is het maken van een interface zodat er gecommuniceerd kan worden met de database. Dit zal gaan door middel van http-request. Hierdoor ontstaat de mogelijkheid voor mensen met enige kennis van CURL om onze database te gebruiken. Het toevoegen van de mapreduce biedt de gebruikers de kans om over grote hoeveelheden data zinnig informatie te onttrekken.

3 Theorie

3.1 REST api

Het interface dat we gebruiken is een Representational State Transfer (REST) interface. Via dit interface kan er gecommuniceerd worden met de database dankzij gebruik van het HTTP protocol.

De documenten zullen beschikbaar worden door links te bezoeken in de webbrowser in de vorm van `http://localhost:PORT/documents/` om bijvoorbeeld de database te bezoeken, en `http://localhost:PORT/document/DOC_ID` om het document met $\$DOC\_ID$ te bekijken. Hierdoor ontstaat als het ware een soort van grafische interface voor de database.

Het REST interface zal gemaakt worden met behulp van de python library flask.

3.2 MapReduce

Ook kunnen er Mapreduce queries gemaakt worden via de REST interface. Deze queries moeten toegepast worden op alle key:value pairs in de B+ tree, en map en reduce functies inladen uit een bestand. Ook moet het mogelijk zijn meerdere waardes te emitten en moeten de waardes na de Mapreduce in bruikbare vorm gereturnd worden. Om flexibiliteit te verbeteren hoeven waarden niet gesorteerd te worden, aangezien er geen limiet is aan de datastructuur van het queryresultaat (naast de key:value structuur). Voor deze flexibiliteit is gekozen omdat er alleen code gerund kan worden die al in de (sub)directories van de database staan. Het is dus onmogelijk om als niet-admin eigen mapreduce queries in te laden, en er hoeft dus geen rekening gehouden te worden met eventuele security risks.

4 Experimenten

Bij het maken van de database hebben we de taken verdeeld. We beginnen met het maken van de REST api waardoor er via een interface kan worden gecommuniceerd met de database (door Steven). Wanneer dit gelukt is voegen we de mogelijkheid toe om mapreduce toe te passen op onze database (door Bär). Vervolgens is het zaak de interface te combineren met de mapreduce, zodat deze gemakkelijk te gebruiken is (door Steven). Ook moet er een test-environment gemaakt worden om de database te benchmarken en data te plotten (door Bär)

4.1 REST api

We hebben gekozen om flask te gebruiken in plaats van tornado om de REST api te maken. Omdat we voorgaande week ook met python3 hebben gewerkt, downloaden we flask voor python3. Dat gaat met behulp van het volgende commando:

```
sudo python3 -m pip install flask
```

De flask module kan nu geopend worden door de volgende regel toe te voegen:

```
app = Flask(__name__)
```

En daadwerkelijk gerund worden via:

```
app.run(port=PORT, debug=True)
```

De port is een variabelen en kan door de gebruiker zelf ingesteld worden. Deze mogelijkheid hebben we toegevoegd omdat het afsluiten van flask soms ging doordat er een error in de code stond. Hierdoor werd flask wel afgesloten maar de port niet, waarnaar het nu dus gemakkelijker werd om flask op een andere port te draaien. Daarom duiden we de port waarop flask draait aan als $PORT$.

Nu “app” de flask module is geworden kunnen de POST en GET requests afgehandeld door de volgende functie op te stellen:

```
@app.route('/documents/', methods=['GET'])
def get():
    tmp = []
    for k in t:
        tmp.append({k: t[k]})
    return jsonify(tmp)
```

Wanneer nu flask gerunt wordt (door in de terminal het .py bestand te runnen) zal er voor GET requests op het adres “localhost:PORT/documents” een overzicht zijn van alle keys en values die zich op dat moment in de tree bevinden:


```
[
  {
    "0": {
      "software": [
        [
          "microsoft",
          "windows_8.1:-"
        ],
        [
          "microsoft",
          "windows_rt:-:gold"
        ],
        [
          "microsoft",
          "windows_8:-"
        ],
        [
          "microsoft",
          "windows_server_2012:r2::~~~x64~~"
        ],
        [
          "microsoft",
          "windows_rt_8.1:-"
        ],
        [
          "microsoft",
          "windows_server_2012:-:gold"
        ]
      ],
      "time": "2015-01-13T17:59:00.050-05:00"
    }
  },
  {
    "1": {
      "software": [
        [
          "microsoft",
          "windows_server_2008:r2:sp1"
        ],
        [
          "microsoft",
          "windows_8.1:-"
        ],
        [
          "microsoft",
```


Andere type requests zijn ook geimplementeerd, te vinden in het volgende overzicht. We maken hierbij onderscheid tussen een set van documenten (collections) en één enkel document (element).

Collections De request voor collections kunnen gestuurd worden naar `localhost:PORT/documents/`. Het is hierbij mogelijk om gebruik te maken van de volgende requests.

GET	Bij deze request zal de gehele database worden weergegeven
POST	Bij een POST request wordt het meegegeven object toegevoegd aan de database
DELETE	Bij een delete request zal de gehele database geleegd worden.

Elements De request voor enkele elementen kunnen gestuurd worden naar `localhost:PORT/document/DOC_ID` waarbij *DOC_ID* vervangen dient te worden door de key van het element. Vervolgens kunnen er de volgende requests naar gestuurd worden:

GET	Door een get request te doen op een enkel document zal de browser weergeven wat de de key was en zijn bijbehorende values
PUT	Bij een PUT request op een single doc zal de bestaande content geupdate worden naar de context die meegegeven wordt bij de PUT request
DELETE	Bij een delete request op een document zal het betreffende document uit de database verwijdert worden.

Bij het toevoegen van een document hebben we ook de bulk load optie toegevoegd. Hierdoor is het mogelijk meerdere documenten toe te voegen in één request. Dit gaat door middel van het meegeven van een lijst van documenten (in de vorm van json-objecten).

4.2 Mapreduce

De Mapreduce is geïmplementeerd door middel van de `asteval` library, zoals meegegeven in de voorbeeldcode. Deze library runt de mapreduce functies uit pythonbestanden. Deze pythonbestanden moeten de volgende functies bevatten, met de genoemde return types:

```
def mapper(key, value):
    return [(key, value),(key, value)]

def reducer(key, values):
    return (key, value)
```

Voor de tijdelijke opslag van de de waardes tussen het mappen en reducen wordt gebruik gemaakt van een `defaultdict(list)` object. Dit maakt het gemakkelijk om waardes aan values toe te voegen zonder zorgen te hoeven maken om key errors. Na de reduce worden de waardes opgeslagen in een python dictionary en gereturnd.

4.3 Mapreduce + REST

Van de MapReduce kan gebruik gemaakt worden door POST request te sturen naar `localhost:PORT/mapreduce`. In deze request moet de naam van de python file meegeven worden waarin map en reduce functie zich bevinden.

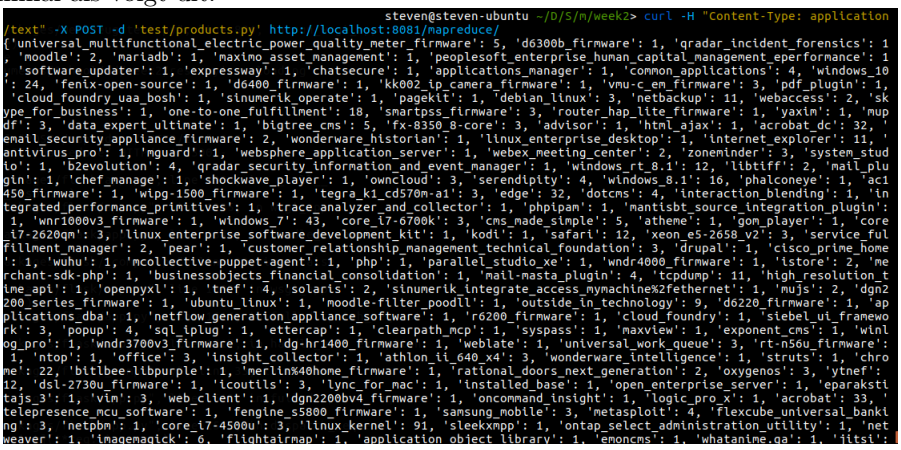
Dat kan als volgt:

```
curl -H "Content-Type: application/text" -X POST \
    -d '_MR.py' http://localhost:8081/mapreduce/
```

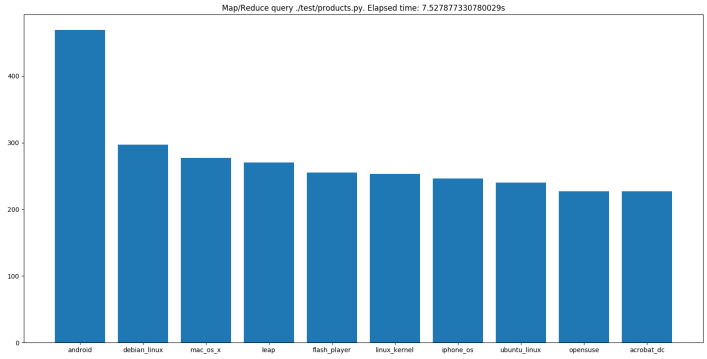
5 Resultaat

Met het inladen van de National Vulnerability Database van 2015 (`https://nvd.nist.gov/download.cfm`) kunnen we onze mapreduce functie testen.

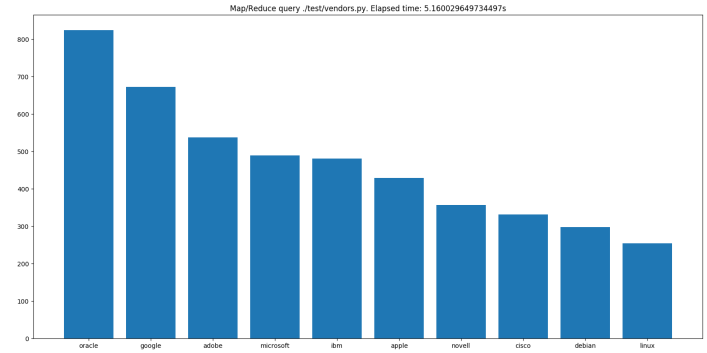
Door het runnen van product.py in de mapreduce van onze database (waarop de NV database is ingeladen) kunnen we een output generen. Dit ziet er in de terminal als volgt uit:



Over deze data is nu nog niet veel te zeggen, maar door er een counter en een plot functie over deze data te runnen we een plot waarin de top 10 van meest kwetsbare producten van 2015:



Ook checken we de output van onze andere mapreduce functie vendors.py, waardoor we uiteindelijk de top 10 software vendors vinden op basis van het totale aantal kwetsbaarheden:



De code voor het maken van deze plots is de vinden in de appendix.

5.1 Benchmarks

Naast het benchmarken van de mapreduce queries is er ook gekeken naar de snelheid van het parsen van de dataset in het geheugen en het inladen van de B+ tree. Het resultaat van deze benchmarks is als volgt:

```
Parsing dataset: 7.43780517578125s
Parsing B+ tree: 0.17426109313964844s
```

6 Conclusie en Discussie

Uit de resultaten kunnen van de top 10 producten en de top 10 vendors kunnen we concluderen dat de database naar behoren werkt. We vergelijken de resultaten van de top 10 producten met de officiele top 10, vrijgegeven door CVE (te

vinden op: `http://www.cvedetails.com/top-50-products.php?year=2015`). Als we deze vergelijken met onze plot zien we dat deze overeenkomt. Hieruit kunnen we opmaken dat het importeren van de database correct is afgehandeld (waaronder het parsen), en dat uit het .xml bestand alle zinnige informatie is ontrokken. Ook blijkt hieruit dat de mapreduce zijn werk correct doet.

Doordat we het zeker voor het onzekere kozen hebben we van het begin af aan gebruik gemaakt van het framework dat Nicky Kessels ons toestuurde. Hierin bevond zich echter geen (werkende?) documentstore. We hebben daarom een static versie gemaakt van de btree waarin we de documenten standaard inladen. De database beschikt enkel over deze data zolang het flask script in run.py runt. In de toekomst hadden we nettere en duidelijkere code kunnen schrijven als we gebruik hadden gemaakt van een werkende documentstore.

Voor het verbeteren van de snelheid van de database zijn er enkele technieken waar in de huidige implementatie (nog) geen gebruik van is gemaakt, maar de performance van de database erg ten goede kunnen beïnvloeden:

- Er zou bij het parsen van de .xml file én het runnen van de mapreduce queries gebruik gemaakt kunnen worden van multithreading. Multithreading is zelfs één van de grootste selling points van het gebruik van mapreduce voor queries. Parralelism is ook essentiëel voor de scalability van de database op meerdere cores, nodes of clusters.

- Een andere mogelijkheid om performance te verbeteren is om een andere taal te gebruiken dan Python. Hoewel Python bekend staat als een erg efficiënte taal voor de programmeur, is de daadwerkelijke performance van pythoncode ondermaats, en zal een database geschreven in een snellere taal, zoals C(++), Go, Rust of Haskell, hogere performance kunnen leveren. Hier staat natuurlijk tegenover dat het omschrijven van de huidige database veel werk is, en voor dit project niet haalbaar. Bij het ontwikkelen van een toekomstige database is de keuze van taal wel belangrijk.

Kortom, hoewel de omstandigheden niet perfect waren, presteert de database toch naar behoren, hoewel er veel ruimte is voor verbetering,

7 Appendix

7.1 rest.py

```
from flask import Flask, jsonify, request
from static_tree import tree
import json
from mapreduce import Script
from collections import Counter

app = Flask(__name__)
```

```
# Retrieval of all docs
@app.route('/documents/', methods=['GET'])
def get():
    tmp = []
    for key in tree:
        tmp.append({key: tree[key]})
    return jsonify(tmp)

# Add docs
@app.route('/documents/', methods=['POST'])
def post():
    obj = request.json

    # bulk loading
    if isinstance(obj, list):
        tot = 0
        for i in obj:
            key = max(tree, key=int)
            tree[key+1] = i
            tot += 1
        return "Succesfully added " + str(tot) + " documents"
    # single dict
    elif isinstance(obj, dict):
        tmp = []
        for k, v in obj.items():
            tmp.append({k: v})
        return jsonify(tmp)

# Deletion of whole database
@app.route('/documents/', methods=['DELETE'])
def delete():
    for key in [reversed([key for key in tree])][1:]:
        del tree[key]
    return "Delete succesful"

@app.route('/mapreduce/', methods=['GET'])
def show_mapreduce():
    return "post a pythonfile to this url, plz"

# MAPREDUCE:
@app.route('/mapreduce/', methods=['POST'])
def run_mapreduce():
    try:
        obj = request.data
        wrapper = Script()
        wrapper.add_file(obj)
```

```
        return str(wrapper.run_mapreduce(tree))
    except:
        return "Something went wrong"

# SINGLE DOC RETRIEVAL
@app.route('/document/<_id>/', methods=['GET'])
def get_single(_id):
    id = int(_id)
    try:
        tmp = {id: tree[id]}
        return jsonify(tmp)
    except:
        return "No entry found for doc " + str(id)

@app.route('/document/<_id>/', methods=['PUT'])
def get_single(_id):
    id = int(_id)
    try:
        obj = request.json
        tree[id] = jsonify(obj)
        return "updated " + str(id) + " to: " + jsonify(obj)
    except:
        return "No entry found for doc " + str(id)

# SINGLE DOC RETRIEVAL
@app.route('/document/<_id>/', methods=['DELETE'])
def delete_single(_id):
    # cast to int, important!!!!!!!!!!
    id = int(_id)
    try:
        del tree[id]
        return "Delete document " + str(id)
    except KeyError:
        return "No entry found for doc " + str(id)
```

7.2 parser.py

```
import xmltodict

def parse(file_name):
    with open(file_name) as fd:
        docs = xmltodict.parse(fd.read())['nvd']['entry']

    store = []
```

```
    for doc in docs:
        entry = {}

        entry['time'] = doc['vuln:published-datetime']

        entry['software'] = []

        if not 'vuln:vulnerable-software-list' in doc:
            continue

        if isinstance(doc['vuln:vulnerable-software-list']['vuln:product'], str):
            spl = doc['vuln:vulnerable-software-list']['vuln:product'].split(':')
            entry['software'].append((spl[2], ':'.join(spl[3:])))
        else:
            for s in doc['vuln:vulnerable-software-list']['vuln:product']:
                spl = s.split(':')
                entry['software'].append((spl[2], ':'.join(spl[3:])))

        store.append(entry)

    return store
```

7.3 test.py

```
# Steven Raaijmakers, Baer Halberkamp
# Run mapreduce and show plots

import matplotlib.pyplot as plt
from collections import Counter
from time import time

from btree import Tree
from parser import parse
from mapreduce import Script

start_time = time()
store = parse('nvdcve-2.0-2016.xml')
print("Parsing dataset: {}s".format(time() - start_time))

start_time = time()
tree = Tree('./storage.boefdb')

for i in range(len(store)):
        tree[i] = store[i]
print("Parsing B+ tree: {}s".format(time() - start_time))
```

```
mapred_files = ['./test/vendors.py', './test/products.py']

for f in mapred_files:
    wrapper = Script()
    wrapper.add_file(f)

    start_time = time()
    results = Counter(wrapper.run_mapreduce(store))
    top10 = results.most_common(10)
    elapsed_time = time() - start_time

    plt.title("Map/Reduce query {}. Elapsed time: {}s".format(f, elapsed_time))
    plt.bar(range(10), [a[1] for a in top10])
    plt.xticks(range(10), [a[0] for a in top10])
    plt.show()
```

7.4 Mapreduce Queries

7.4.1 vendors

```
def mapper(_, value):
    output = []
    for vendor, _ in value['software']:
        if not vendor in output:
            output.append(vendor)
    return [(v, 1) for v in output]

def reducer(key, values):
    return (key, sum(values))
```

7.4.2 products

```
def mapper(_, value):
    output = []
    for _, v in value['software']:
        product = v.split(':')[0]
        if not product in output:
            output.append(product)
    return [(v, 1) for v in output]

def reducer(key, values):
    return (key, sum(values))
```