**Elassandra**

**Introduction**

Elassandra utilizes Elasticsearch in order to perform near real time searching on a Cassandra virtual data center. When a document gets indexed, the JSON document is stored as a row in a Cassandra table and synchronously indexed in Elasticsearch.

Any additional information or clarification can be found on the Elassandra documentation

http://elassandra.readthedocs.io/en/latest/index.html

However, the downside to this documentation is that it assumes you know either one or the other. At the time of picking Elassandra as my project we had not gone over Cassandra or Elasticsearch in class. So I will be going through this assignment learning both from a fresh start.

**Cassandra**

Before we talk about Elasticsearch, covering the basics of Cassandra is important, as it is what we will be using to store all of our data.

This is the basic list of definitions:

* Cluster - A collection of datacenters and nodes
* Datacenter - A collection of Cassandra nodes based on similar characteristics
* Node - A portion of the entire data set where the data is determined by its token key range. It has a primary key which is used as a partition key.
* Keyspace - A collection of tables in Cassandra that function similarly to MySQL. Cassandra offers us the Cassandra Query Language (CQL) in order to perform functions similar to SQL.

Cassandra is a linearly scalable non-relational database where all of the nodes work as peer to peer, rather than master to slave. Cassandra provides the ability to continually add nodes to improve load distribution in the datacenter and allows us to scale out horizontally.

Replication

Replication factor in Cassandra refers to the amount of copies of data that are in a cluster. So a replication factor of one refers to one copy of each row in a cluster, replication factor of two is two copies of each row, etc. This ensures reliability and fault tolerance.

Consistency

Instead of the traditional consistency of ACID properties, Cassandra offers tunable data consistency across a database cluster. This means we can adjust the level of consistency for a particular transaction. If we want more consistency, we would check across multiple replications to ensure our transaction is okay.

Advantages

* Linear performance
* Continuous availability
* Tunable data consistency
* CQL - SQL-like language
* Flexible data model
* Fault detection and recovery

Disadvantages

* No joins with CQL
* No aggregate functions in CQL
* Where clause in CQL is limited to primary keys.

**Elasticsearch**

Elasticsearch is a near real time search platform, meaning that there is only a slight delay from when you index a document until the time it becomes searchable.

This is the basic list of definitions:

* Cluster - A collection of one or more nodes (servers) that all together hold your entire dataset while providing indexing and search capabilities across all nodes.
* Node - A single server that is part of the cluster. Stores data and participates in the cluster's indexing and search capabilities.
* Index - A collection of documents that have similar characteristics.
* Type - A category for the data in your index.
* Document - A basic unit of information that can be indexed expressed in JSON.
* Shard - A subdivision of an index. Each shard is a fully functional and independent "index" that can be hosted on any node in the cluster.
* Replica - A copy of an index's shard.

What are Shards and Replicas used for?

Hardware limitations are a major factor when indexing data. If a node cannot store all of the data, then it is in our best interest to split that data up into several different nodes. This is where shards come into play. We can horizontally scale content volume and allows us to distribute the workload between several nodes to increase performance when searching.

Replicas are important to have as a failsafe in such a case where a shard or node fails. And although backups are nice to have, Replicas also have the advantage of being able to allow searches to be executed on all replicas in parallel, resulting again in a performance boost.

Advantages

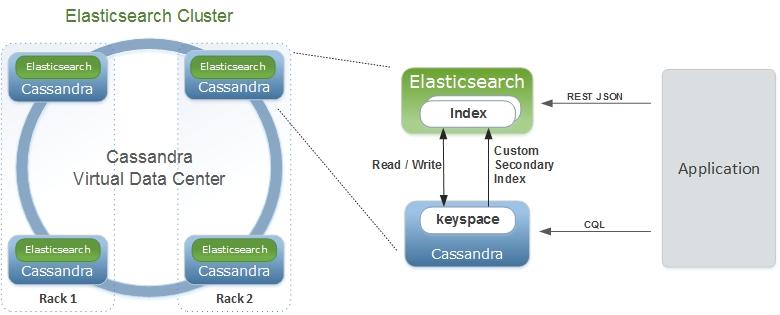
* Full-text search
* Schema-free
* REST API
* Per-operation persistence
* Document-oriented
* Easy to scale out

Disadvantages

* Non-ACID compliant
* No SQL-like queries
* No security
* No join operations across indices

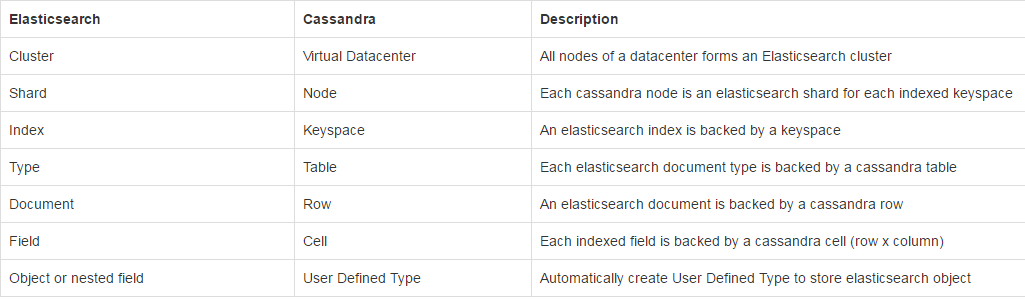
**Elassandra**

With the features of Cassandra and Elasticsearch discussed, we can finally move onto Elassandra. Again, Elassandra is simply the integration of Elasticsearch within Cassandra as a way of searching through data in near real time. Elasticsearch acts as a secondary index for the Cassandra keyspace, and both Elasticsearch and Cassandra can read or write to the stored data using CQL, REST API or JSON. The following image illustrates this process.



Source: http://elassandra.readthedocs.io/en/latest/architecture.html

Below is the concept map that relates Elasticsearch terminology with Cassandra.



Elassandra gives us the benefit of using the features of both Cassandra and Elasticsearch. We can load and store data into Cassandra with SQL-like queries and then map an Elasticsearch index to a Cassandra keyspace in order to query our data. On the other hand, if we are given JSON data, we can also use the REST API to either do a single insertion or bulk insertion.

Differences

* Sharding depends on the number of nodes in the datacenter, and the number of replicas is defined by the keyspace replication factor.

**Installation/Setup**

As a start, it must be noted that there is no way of setting Elassandra up on Windows unless it is installed through a Virtualbox. In my case, I already have a setup of Arch Linux installed on my laptop and will be going through the main steps with that in mind. The prerequisites for Elassandra are Java 8, python, and pip.

While there are five ways of installing Elassandra, I will be focusing mainly on installation using Tarball. Because I am using Arch Linux, the package managers that I use (yaourt and pacman) do not have Elassandra on their download list, whereas apt for Debian and Ubuntu do. If you are on either of those two Linux distros, there's a section on how to install using a DEB package at:

http://elassandra.readthedocs.io/en/latest/installation.html#deb-package.

There are other methods such as RPM installation, using a Docker image, or building from source, but Tarbell is by far the easier route for my specific Linux distro.

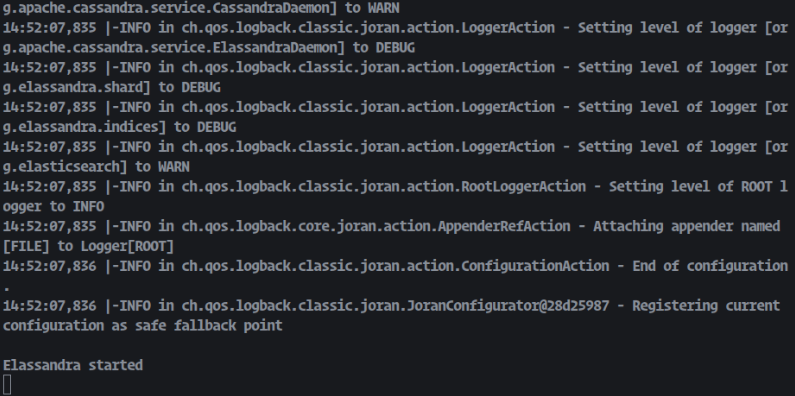
To begin, assuming all prerequisites are installed, we have to download the tarball file from github.

https://github.com/strapdata/elassandra/releases/

This link will have all of the past and current versions. For this installation, we will be using version 2.4.2.15. Initially I didn't think to check for new updates and ended up installing version 2.4.2.10, which caused me to run into issues with mapping the data as I will discuss later.

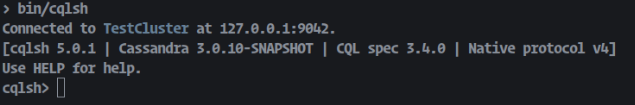
Once the tarball is downloaded, extract it's contents with 'tar -xzf elassandra-2.4.2.15.tar.gz'

To run Cassandra, point your terminal to the Elassandra directory with 'cd Elassandra-2.4.2.15' and then type in 'bin/cassandra -f -e' to start Cassandra in the foreground with Elasticsearch enabled. If no errors occur, Elassandra will give a message saying it has started.



Next we need to install the Cassandra driver for python to run cqlsh using the command 'sudo pip install Cassandra-driver'

Once that is finished installing, launch a new terminal and change directory to your elassandra's bin folder. We can then connect our elassandra node with cqlsh by running 'cqlsh'.



To verify our Elassandra cluster is working, opening up a new terminal and inputting 'curl -X GET http://localhost:9200/' should return our cluster information.

{

"name" : "127.0.0.1",

"cluster\_name" : "Test Cluster",

"cluster\_uuid" : "7cb65cea-09c1-4d6a-a17a-24efb9eb7d2b",

"version" : {

"number" : "2.4.2",

"build\_hash" : "b0b4cb025cb8aa74538124a30a00b137419983a3",

"build\_timestamp" : "2017-04-19T13:11:11Z",

"build\_snapshot" : true,

"lucene\_version" : "5.5.2"

},

"tagline" : "You Know, for Search"

}

Once this is confirmed to be running, we can now move onto loading our data into Cassandra, and then map our Elasticsearch index to the Cassandra keyspace.

**Extracting Data**

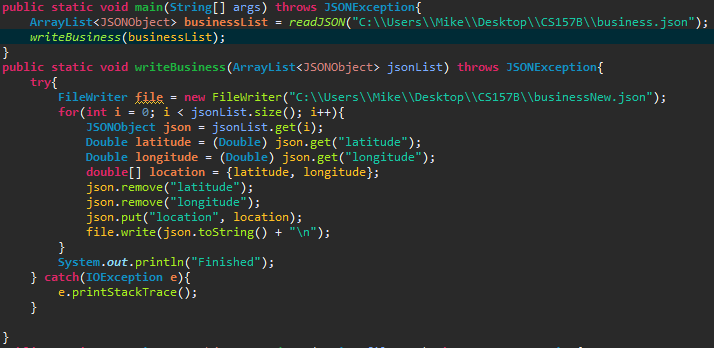
The data that we will be using is the yelp dataset given in class or here: https://www.yelp.com/dataset\_challenge

Bulk extraction - Capture a snapshot of the source data at a point in time.

Incremental extraction - Capturing changes that have occurred since the last bulk extract.

**Transforming Data**

A major hiccup I ran into while trying to transform the data into something workable was the fact that I couldn’t actually get a large enough dataset for Users and Review before my Java program would crash. I had problems with getting Python to load the original JSON files on my Desktop, which unfortunately has more memory than my laptop, so I opted with Java to edit the JSON files. While working on compiling all ‘compliment\_\*’ attributes from User table into one list, I found out that the upper limit of the amount of data that I could load in was roughly 500,000. Anything after that and my program would just run out of memory.



This image for example shows combining latitude and longitude into one column ‘location’. Business was able to run completely since it only has about ~140,000 rows, but for other big files like User, Review, or Tip, they have at least a million rows. So I opted to stick with the original JSON files and upload it without transforming the data directly into Cassandra using a Python script, which oddly works on my laptop but not my Desktop.

**Loading Data**

It is important to note that at the base of Elassandra, we are using Cassandra as our storage. There are two methods of bulk loading in Cassandra, through CSV files or through JSON objects. Although it’s possible to use the CSV files, I decided to use a python script to read the JSON file and upload each JSON object into the keyspace one by one. The original python script that I found was located at: <https://github.com/pobed2/cassandra-yelp-dataset> where I modified the table definitions and modified the script to be Python 3.\* compliant.

The code for creating the tables and uploading the data is provided below:

**Create tables (queries.cql)**

CREATE TABLE IF NOT EXISTS business(

business\_id text PRIMARY KEY,

address text,

hours list<text>,

is\_open int,

categories list<text>,

city text,

review\_count int,

name text,

neighborhood text,

postal\_code text,

longitude double,

latitude double,

state text,

stars double,

attributes list<text>,

type text

);

CREATE TABLE IF NOT EXISTS review(

review\_id text PRIMARY KEY,

user\_id text,

business\_id text,

funny int,

useful int,

cool int,

stars double,

date text,

text text,

type text

);

CREATE TABLE IF NOT EXISTS user(

user\_id text PRIMARY KEY,

name text,

yelping\_since text,

review\_count int,

friends set<text>,

fans int,

average\_stars double,

type text,

useful int,

funny int,

cool int,

compliment\_photos int,

compliment\_list int,

compliment\_funny int,

compliment\_plain int,

compliment\_note int,

compliment\_writer int,

compliment\_cute int,

compliment\_more int,

compliment\_hot int,

compliment\_profile int,

compliment\_cool int,

elite list<text>

);

CREATE TABLE IF NOT EXISTS tip(

tip\_id uuid PRIMARY KEY,

user\_id text,

text text,

business\_id text,

likes int,

date text,

type text

);

CREATE TABLE IF NOT EXISTS checkin(

checkin\_id uuid PRIMARY KEY,

time list<text>,

business\_id text,

type text

);

**Upload data (load\_json.py)**

import json

import uuid

from cassandra.cluster import Cluster

def import\_data(table\_name, json\_filename):

cluster = Cluster()

session = cluster.connect()

insert\_statement = 'INSERT INTO %s JSON %s' % (table\_name, '%s')

with open(json\_filename) as data\_file:

i = 0

for line in data\_file:

i = i + 1

if(i > 3927911):

corrected\_line = line.strip().replace('\n', '\\n')

session.execute(insert\_statement,[corrected\_line])

cluster.shutdown()

def import\_data\_without\_primary\_key(table\_name, json\_filename, name\_of\_id):

cluster = Cluster()

session = cluster.connect()

insert\_statement = 'INSERT INTO %s JSON %s' % (table\_name, '%s')

with open(json\_filename) as data\_file:

for line in data\_file:

corrected\_line = json.loads(line.strip().replace('\n','\\n'))

corrected\_line[name\_of\_id] = str(uuid.uuid1())

corrected\_line = json.dumps(corrected\_line)

session.execute(insert\_statement,[corrected\_line])

cluster.shutdown()

if \_\_name\_\_ == '\_\_main\_\_':

print("Importing review data...")

import\_data('yelp.review','review.json')

print("Finished importing review data")

print( "Importing business data...")

import\_data('yelp.business', 'business.json')

print("Finished importing business data.")

print("Importing review data...")

import\_data('yelp.review','review.json')

print("Finished importing review data.")

print("Importing user data...")

import\_data('yelp.user','user.json')

print("Finished importing user data.")

print("Importing tip data...")

import\_data\_without\_primary\_key('yelp.tip', 'tip.json', 'tip\_id')

print("Finished importing tip data.")

print("Importing checkin data...")

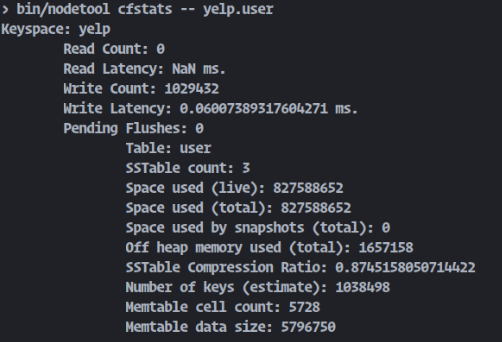
import\_data\_without\_primary\_key('yelp.checkin','checkin.json', 'checkin\_id')

print("Finished importing checkin data.")

This process took around 4+ hours to upload ~4 million rows in review, ~1 million in user/tip, ~100,000 in checkin and business. There were times where my laptop did freeze because I ran out of memory, but it eventually fixed itself.

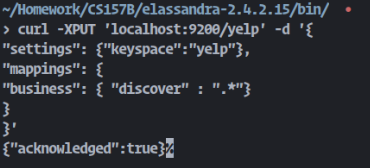
The nodetool tool that’s located inside /bin/ folder of Elassandra provides a useful way of checking how much data we loaded with the following command:

$ nodetool cfstats *--* <*keyspace*>.<*table*>



**Mapping Data**

Below is an example of how we would map our data to Elasticsearch. Elassandra uses the keyspace yelp to map the Business table and discovers all the attributes of the Business table. We would also define the mappings for our other tables too like User or Tip.



However, the issue with the “discover” field is that you are letting Elasticsearch do all of the type assumptions on its own, and you end up getting results like this:



Notice how for the TIP table, both user\_id and tip\_id are considered primary keys, where cql\_primary\_key\_order >= 0 is a primary key. So to fix this, we have to manually define types, collection types, and primary keys.

I ended up settling on creating a shell script to make the mapping indentation look a lot nicer and also for it to make more sense.

**map.sh**

#define json

json='

{

"settings" : {"keyspace" : "yelp"},

"mappings" :

{

"review" :

{

"properties" :

{

"review\_id": { "type" : "string", "cql\_collection" : "singleton", "cql\_primary\_key\_order" : 0 },

"user\_id" : { "type" : "string", "cql\_collection" : "singleton" },

"business\_id": { "type" : "string", "cql\_collection" : "singleton" },

"funny" : { "type" : "integer", "cql\_collection" : "singleton" },

"useful" : { "type" : "integer", "cql\_collection" : "singleton" },

"cool" : { "type" : "integer", "cql\_collection" : "singleton" },

"stars" : { "type" : "double", "cql\_collection" : "singleton" },

"date" : { "type" : "string", "cql\_collection" : "singleton" },

"text" : { "type" : "string", "cql\_collection" : "singleton" },

"type" : { "type" : "string", "cql\_collection" : "singleton" }

}

},

"tip" :

{

"properties" :

{

"tip\_id" : { "type" : "string", "cql\_collection" : "singleton", "cql\_primary\_key\_order" : 0 },

"user\_id" : { "type" : "string", "cql\_collection" : "singleton" },

"text" : { "type" : "string", "cql\_collection" : "singleton" },

"business\_id" : { "type" : "string", "cql\_collection" : "singleton" },

"likes": { "type" : "integer", "cql\_collection" : "singleton" },

"date" : { "type" : "string", "cql\_collection" : "singleton" },

"type" : { "type" : "string", "cql\_collection" : "singleton" }

}

},

"checkin" :

{

"properties" :

{

"checkin\_id": { "type" : "string", "cql\_collection" : "singleton", "cql\_primary\_key\_order": 0 },

"time" : { "type" : "string", "cql\_collection" : "list" },

"business\_id" : { "type" : "string", "cql\_collection" : "singleton" },

"type": { "type" : "string", "cql\_collection" : "singleton" }

}

},

"business" :

{

"properties" :

{

"business\_id": { "type" : "string", "cql\_collection" : "singleton", "cql\_primary\_key\_order": 0 },

"address" : { "type" : "string", "cql\_collection" : "singleton" },

"hours" : { "type" : "string", "cql\_collection" : "list" },

"is\_open" : { "type" : "integer", "cql\_collection" : "singleton" },

"categories" : { "type" : "string", "cql\_collection" : "list" },

"city" : { "type" : "string", "cql\_collection" : "singleton" },

"review\_count": { "type" : "integer", "cql\_collection" : "singleton" },

"name" : { "type" : "string", "cql\_collection" : "singleton" },

"neighborhood": { "type" : "string", "cql\_collection" : "singleton" },

"postal\_code": { "type" : "string", "cql\_collection" : "singleton" },

"longitude" : { "type" : "double", "cql\_collection" : "singleton" } ,

"latitude": { "type" : "double", "cql\_collection" : "singleton" },

"state" : { "type" : "string", "cql\_collection" : "singleton" },

"stars" : { "type" : "double", "cql\_collection" : "singleton" },

"attributes" : { "type" : "string", "cql\_collection" : "list" },

"type" : { "type" : "string", "cql\_collection" : "singleton" }

}

}

}

}'

user='{

"user" :

{

"properties" :

{

"user\_id" : { "type" : "string", "cql\_collection" : "singleton", "cql\_primary\_key\_order": 0 },

"name" : { "type" : "string", "cql\_collection" : "singleton" },

"yelping\_since" : { "type" : "string", "cql\_collection" : "singleton" },

"review\_count" : { "type" : "integer", "cql\_collection" : "singleton" },

"friends" : { "type" : "string", "cql\_collection" : "set" },

"fans" : { "type" : "integer", "cql\_collection" : "singleton" },

"average\_stars" : { "type" : "double", "cql\_collection" : "singleton" },

"type" : { "type" : "string", "cql\_collection" : "singleton" },

"useful" : { "type" : "integer", "cql\_collection" : "singleton" },

"funny" : { "type" : "integer", "cql\_collection" : "singleton" },

"cool" : { "type" : "integer", "cql\_collection" : "singleton" },

"compliment\_photos" : { "type" : "integer", "cql\_collection" : "singleton" },

"compliment\_list" : { "type" : "integer", "cql\_collection" : "singleton" },

"compliment\_funny" : { "type" : "integer", "cql\_collection" : "singleton" },

"compliment\_plain" : { "type" : "integer", "cql\_collection" : "singleton" },

"compliment\_note" : { "type" : "integer", "cql\_collection" : "singleton" },

"compliment\_writer" : { "type" : "integer", "cql\_collection" : "singleton" },

"compliment\_cute" : { "type" : "integer", "cql\_collection" : "singleton" },

"compliment\_more" : { "type" : "integer", "cql\_collection" : "singleton" },

"compliment\_hot" : { "type" : "integer", "cql\_collection" : "singleton" },

"compliment\_profile" : { "type" : "integer", "cql\_collection" : "singleton" },

"compliment\_cool" : { "type" : "integer", "cql\_collection" : "singleton" },

"elite" : { "type" : "string", "cql\_collection" : "list"}

}

}

}'

#remove existing index

curl -XDELETE 'localhost:9200/yelp?pretty=true'

echo $json | curl -XPUT 'localhost:9200/yelp?pretty=true' -d @-

echo $user | curl -XPUT 'localhost:9200/yelp/\_mapping/user?pretty=true' -d @-

I needed to confirm that the data was in fact loaded with the REST API onto Elasticsearch, so I sent in another curl command :

curl -XGET ‘localhost:9200/yelp?pretty=true’

which returns the following response:



with all of the maps that we defined in map.sh.

**Visualizing Data**

To visualize our data, we can use Kibana, which is an open-source platform designed to work with Elasticsearch. Using the indices defined by Elasticsearch, we can search, view, and interact with the data at hand to create graphs, charts, tables, and maps.

Setting up Kibana is relatively simple. Again, just like our Elassandra setup, we have several options to install Kibana: tarball, deb, rpm, or docker. I will be using the tarball installation since it’s relatively easy.

The download links for Kibana can be found at

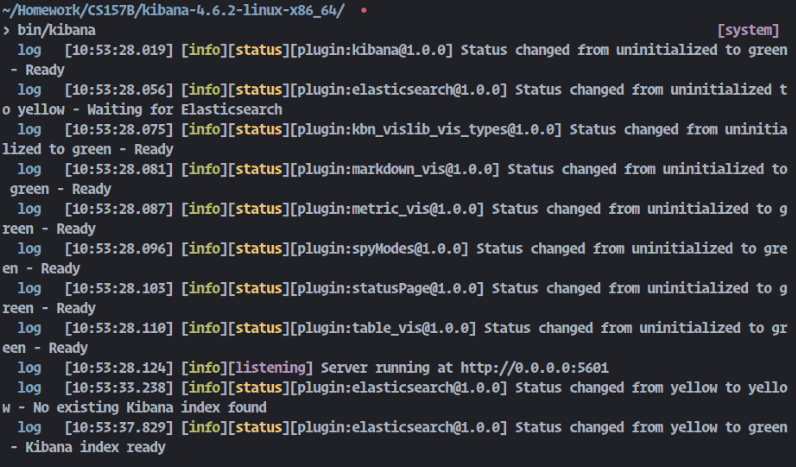
<https://www.elastic.co/downloads/past-releases/kibana-4-6-4>

Once the tarball is downloaded, extract it's contents with

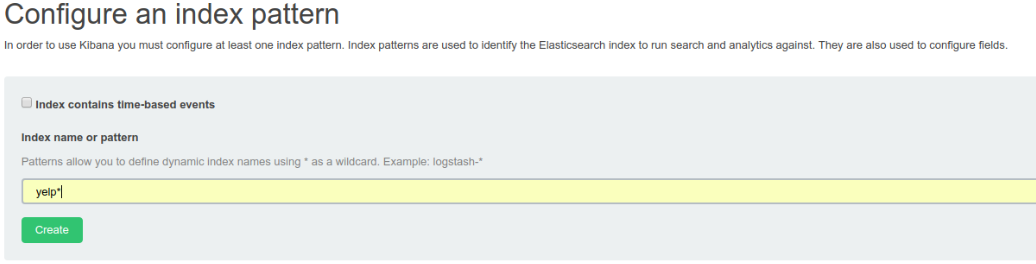
'tar -xzf kibana-4.6.4-linux-x86\_64.tar.gz'

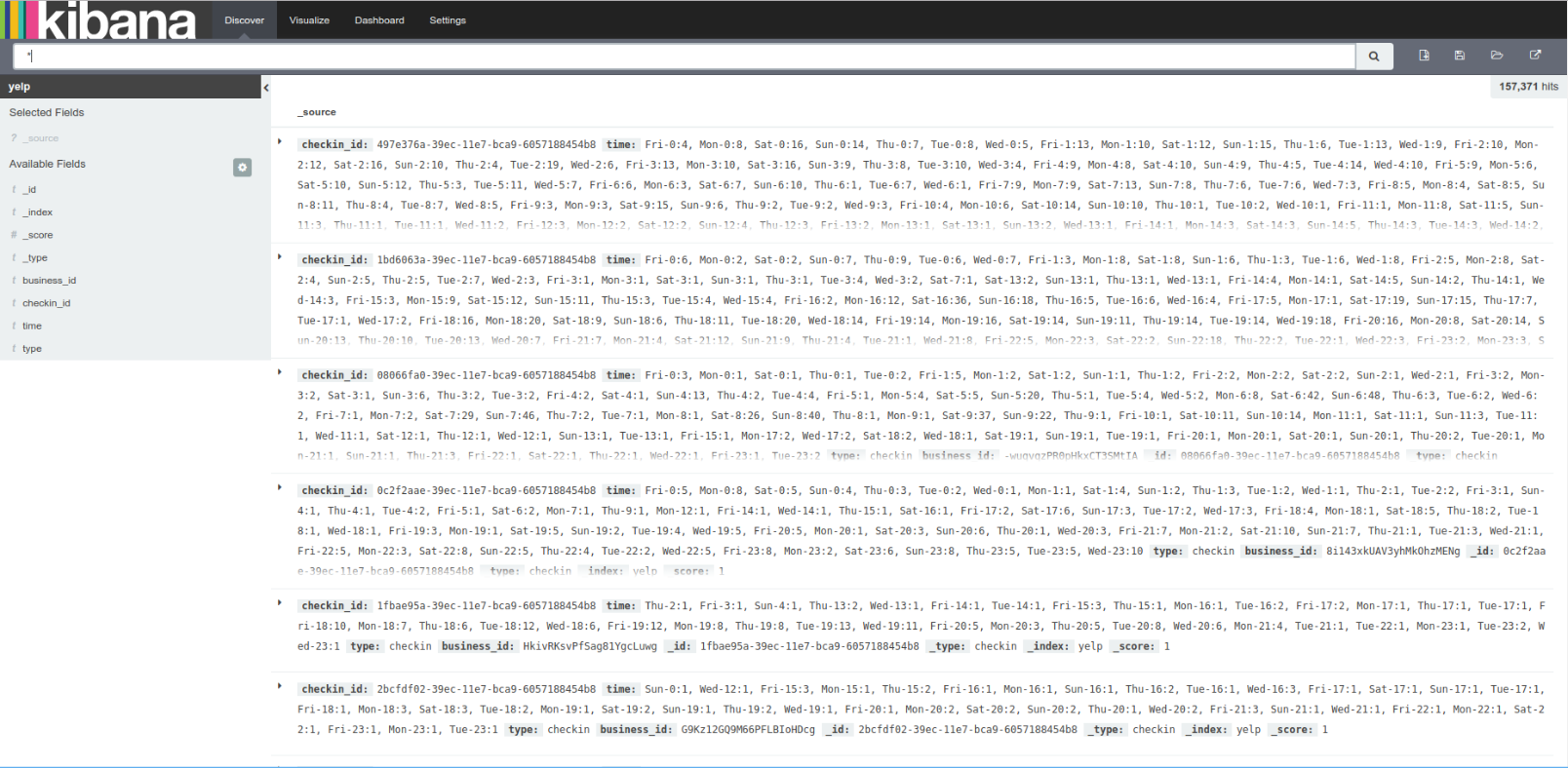
We an Kibana version 4.6.4 because Kibana 5.4 is designed to work with Elasticsearch 5, which is not integrated into Elassandra just yet. Refer to the following git issue comment for more info: <https://github.com/elastic/kibana/issues/8515#issuecomment-252039090>

To run Cassandra, point your terminal to the Kibana directory with 'cd kibana-4.6.4-linux-x86\_64' and then type in 'bin/kibana' to start Kibana. If no errors occur, Kibana will give a message saying it has started and has connected to Elasticsearch indices. Elassandra must be running for this to work.

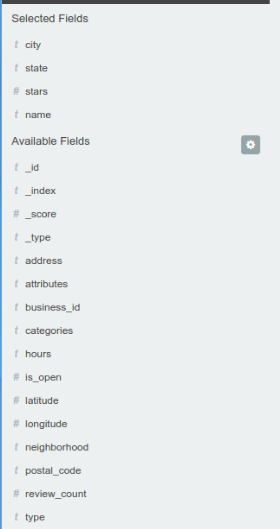
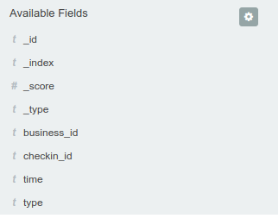


Going to ‘localhost:5601’ in our browser redirects us to the Kibana app page, where it asks us to configure an index pattern:





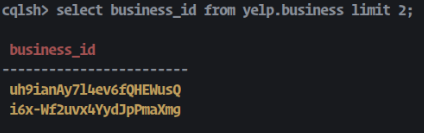
Interestingly enough, and I never figured out whether or not this was fixable, but since there is no timestamp, it only shows the table fields of the LAST table in our first curl map, in this case Checkin. But after swapping the Business table properties and checkin, I was able to get the fields for Business.

 vs 

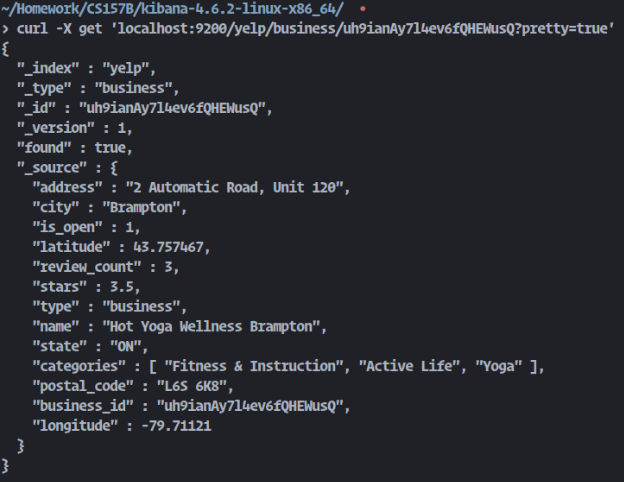
**Queries**

Elasticsearch provides very fast and loosely typed queries. Below I show the difference between a select query with CQL, a select query with Elasticsearch REST API, and a select query in Kibana.

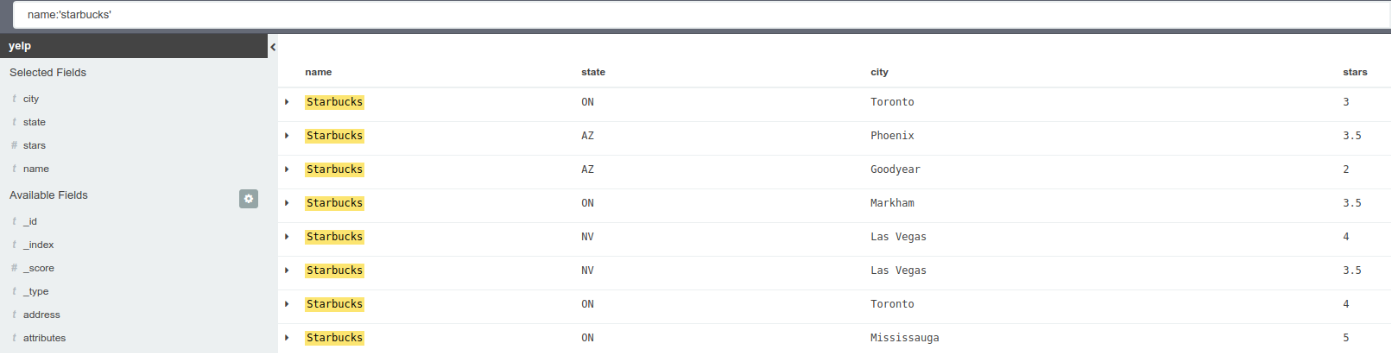
**CQL:**



**REST API:**

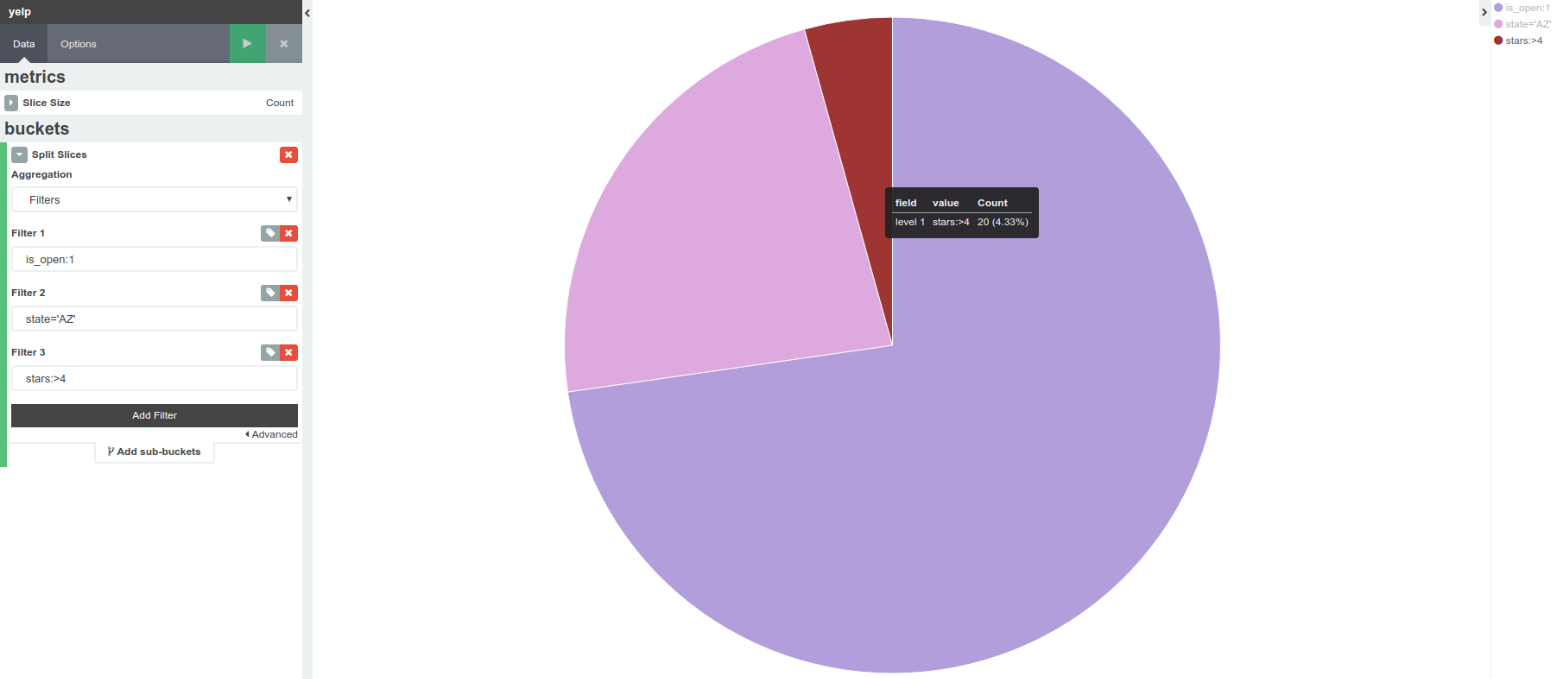


**Kibana:**



Kibana has the advantage of being able to easily highlight and select which fields we want to display without the need to requery again. REST API will always show all of the “\_source” fields we have defined, and CQL just remains similar to SQL.

Kibana also has the very neat tool to display filters with a pie chart



**Optimize**

* Lower replication factor at the cost of reliability
* Lower consistency levels
* Increase RAM significantly. This was the main issue I ran into when running Elasticsearch, CQL, and Kibana all at the same time. Queries were still fast because of Elasticsearch, but my computer froze which was not a very good sign.
* Being able to transform the data properly given more RAM would definitely help with the loading data process, as there would be less attributes to process.
* Cleaning bad rows would also significantly help reduce file size.