**CS157B Project Report**

**Michael Nguyen**

# 

[Project Overview](#_t82ve5bt4s3y)

[Description](#_auiwl7jmldkv)

[Data](#_d83fnddu7qjo)

[Bird’s Eye View](#_wgiyopkfsz30)

[SQL](#_liswtavlpl3w)

[Setup](#_1q17bm9xaen3)

[Database / Schema](#_f86cbrjt73e4)

[ETL](#_k2ohf8q7ce7l)

[Retrieval Queries](#_kl5o8eei4ap6)

[Update Queries](#_cujoob1s2o0)

[Data Quality / Consistency](#_qz8tde6y2d13)

[Performance Optimization](#_nsi50fwstdjh)

[Hive](#_daoe3kku7uq5)

[Setup](#_1q17bm9xaen3)

[Database / Schema](#_f86cbrjt73e4)

[ETL](#_oclx3qgtcq6d)

[Retrieval Queries](#_svoooqt7pybr)

[Update Queries](#_p87inat7juza)

[Data Quality / Consistency](#_xsdvuri5684)

[Performance Optimization](#_tq2a7m4szyev)

[Spark](#_ymjswyx8sfqk)

[Setup](#_5dcn8jevtlu4)

[Database / Schema](#_7duamj58yclj)

[ETL](#_3nr2uvj1hmv7)

[Retrieval Queries](#_v9zfpt0pdpd)

[Data Quality / Consistency](#_jhk2xdvmbvwn)

[Performance Optimization](#_juig3h3ajziz)

[Elassandra](#_wtr9q2nes5mw)

[Setup](#_nofz1t4wndah)

[Database / Schema](#_squixmuxrf3e)

[ETL](#_4mrjn08dnur)

[Retrieval Queries](#_x2wfsd7eq290)

[Performance Optimization](#_aqar0dug7k1x)

[Technology Comparison](#_ti3fto7i9kvw)

[SQL](#_ahwdyruy61ng)

[Hive](#_mp1fwrbc6ven)

[Spark](#_n77b04nil876)

[Elassandra](#_ggi7d5ulhhpu)

# 

# Project Overview

## Description

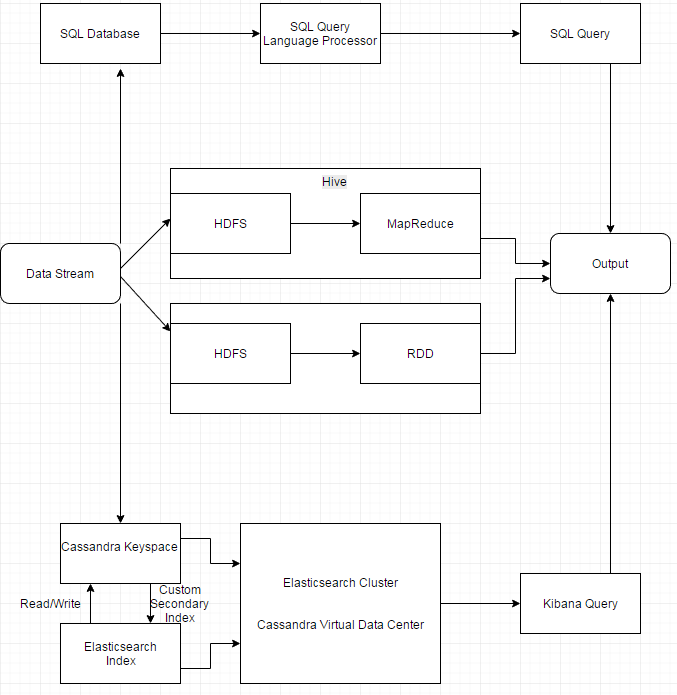
The goal of this project is to be able to perform analytical queries on a given yelp dataset. The project will cover different big data technologies to understand when to use certain technologies for certain queries and workloads, which will then have their strengths and weaknesses compared. For each different technology, the entire pipeline of extracting, loading, and transforming data will be repeated to run OLTP and OLAP-query processing. In the end, data quality and performance optimization will be discussed for each.

## Data

Description of Yelp dataset that shall be used for the project. Provide some general characteristics and discuss how much / which parts of the data was used for your project task.

The Yelp dataset consists of five tables: Business, Checkin, Review, User, and Tip table. Business table contains information about a business such as its location, rating, hours, categories, and attributes (types of parking available, does take out, accepts credit card, etc.). Checkin table contains checkins for a certain business. Review table contains information about the review text, who the reviewer was, what business the review was for, and who liked the review. User table contains all of the user information aside from sensitive account information, such as the amount of reviews they have given, their friends, and the feedback they have gotten from their reviews. Tip table contains the tips that a user has given for a business and when they gave it.

# Bird’s Eye View



# SQL

## Setup

For a Windows setup, I installed MySQL Workbench and proceeded with creating the database and schema.

## Database / Schema

All of the table fields were kept the same and the tables were individually created manually based off of the header row in the CSV files.

## ETL

Tip had to remove ~35,000 bad entries, 930,924 rows total took 5 seconds to enter into database.

Business loaded 142,251 rows total taking 23.182 seconds. Had to change delimiter from ',' to '\t'. Businesses also had some strange names, so I had to re-encode the file to UTF-8 in order for MySQL to read the special characters properly.

Checkin loaded 125,532 rows total taking 1.217 seconds. No modifications to the file

User loaded 1,029,432 rows total taking 289.086 seconds. No modifications to the file

Review loaded 4,153,150 rows total taking 2719.534 seconds. No modifications to the file

The smaller files were opened with Notepad++ to fetch the attributes of the table but the large ones (User and Review) table had to use a workaround to get the attributes for manual table creation. I had to use the import wizard on MySQL Workbench which successfully read the attributes and used that. Luckily it also showed me a little snippet of data so I could also confirm what datatype it was. Every other program that I tried could not get these two opened so it took a long time to figure out what to do.

Loading in the files were done in MySQL Workbench with the following command:

LOAD DATA LOCAL INFILE "C:\\Users\\Mike\\Desktop\\CS157B\\yelp\_academic\_dataset\_review-3.csv"

INTO TABLE review

CHARACTER SET UTF8

FIELDS TERMINATED BY ',' ESCAPED BY '\Z' ENCLOSED BY '"'

LINES TERMINATED BY '\n'

IGNORE 1 LINES

(funny, user\_id, review\_id, text, business\_id, stars, date, useful, type, cool)

With the same command being run for every table, swapping out the header fields listed on the last line to the specific table headers.

## Retrieval Queries

This query orders the top states with reviewed Vietnamese restaurants by State alphabetical order. It is not possible to order by review\_count because of the priority of GROUP BY over SELECT.

SELECT yelp.business.state, count(\*) AS review\_count

FROM yelp.business

WHERE yelp.business.categories LIKE '%Vietnamese%'

GROUP BY yelp.business.state

This query finds the users who use 'Amazing' in their tips and how many tips they have left with that word in it.

SELECT user.name, COUNT(\*) AS c

FROM yelp.tip

JOIN user ON user.user\_id = tip.user\_id

WHERE yelp.tip.text LIKE 'Amazing%'

GROUP BY yelp.user.name

## Update Queries

This query gives all yelp users who have been a member since 2006 elite membership for the year 2017

UPDATE yelp.user

SET yelp.user.elite = '[u\'2017\']'

WHERE user.yelping\_since LIKE '2006%' AND user.elite = '[u\'None\']'

## Data Quality / Consistency

We can introduce a trigger to not allow restaurants to be added into the business table unless they have at least 3 reviews (as is the case right now).

DELIMITER $

CREATE TRIGGER review\_count\_check BEFORE INSERT ON yelp.business

FOR EACH ROW

BEGIN

IF NEW.review\_count < 3

THEN

SIGNAL SQLSTATE '45000'

SET MESSAGE\_TEXT = 'Constraint violation: Not enough reviews';

END IF;

END$

DELIMITER;

## Performance Optimization

Creating indexes on my desktop yielded varying differences. Sometimes it would not change the speed at all, or it would change dramatically. But for the most part indexes to make the queries run significantly faster. For example this query finds all tips with just the word 'Amazing' in it. The initial query started off at 0.612 seconds, and after the index was added, it was 0.000 seconds.

SELECT text

FROM tip

WHERE text LIKE 'Amazing'

create index tipIndex on tip(text(1000))

# Hive

## Setup

Hive was installed with Hortonworks Sandbox VM. Using VirtualBox I imported the Hortonworks provided VirtualBox image containing Hive and followed the rest of their tutorial at <https://hortonworks.com/hadoop-tutorial/getting-started-hdf-sandbox/> to get it running.

## Database / Schema

All of the table fields were kept the same and the tables were individually created manually based off of the header row in the CSV files.

## ETL

I extracted my data from my MySQL Workbench setup, which had already imported and cleaned out all duplicated data. Extracting the data was a big headache, as I discovered that there was a random newline at the final column of each row in every table which must have meant I added one too many extra fields when I imported the data using LOAD DATA LOCAL INFILE. To fix this, I had to use REPLACE(REPLACE(\*FIELD\*, CHAR(13), ''), CHAR(10), '') to replace every rows terminating newline to with nothing.

I chose 500000 as my limit because it felt like a large enough of a dataset for User and Review. It was 50% of the User dataset and 12.5% of the Review dataset.

All of the data was loaded into Hive database using the GUI with no issues other than reserved keywords like 'date' or 'user', which just needed a quick rename. The files were small enough (~70MB to 400MB) that uploading with the GUI was also very simple. It might have been faster to upload it using ssh, but I felt it wasn't very necessary to do since the GUI was intuitive.

## Retrieval Queries

Find the average rating for Greek food in a city

SELECT city, AVG(stars) AS average\_rating, count(\*) AS business\_count

FROm business

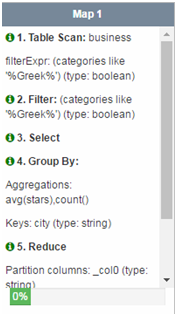
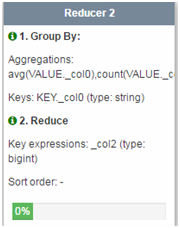
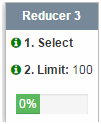
WHERE categories LIKE ‘%Greek%’

GROUP BY city

ORDER BY business\_count DESC

LIMIT 100;

## 

->SHUFFLE-> ->SHUFFLE-> 

## Update Queries

This query closes all H&M businesses in the table.

UPDATE business

SET is\_open = 0

WHERE name = “H&M”

However, there was an issue with my tables where I could not run any update or delete queries because they were non-ACID tables and were not clustered into any buckets.

Upon creating an ACID table clustered into four buckets: create table business (...) CLUSTERED BY(business\_id) INTO 5 BUCKETS, it would no longer be able to upload the data from the CSV into the table since the file was not ORC.

## Data Quality / Consistency

We can clean and filter bad data in small chunks when we partition our table. Whenever possible it is ideal to simply fix the data before Hive has to do any processing with the data.

## Performance Optimization

Partitioning the table and clustering the data into buckets are ways to improve performance.

Partitioning the table allows us to split the data into smaller chunks so that when we query something with a partition key, it would greatly aid the query. Clustering the data into buckets improves join performance if the bucket key and the join keys are common, reducing I/O scans during the join process if the process is happening on the same columns.

# Spark

## Setup

Spark did not require any installation, and instead was a cloud-based setup with Databricks website. Databricks Community Edition was free of charge to run Apache Spark jobs. Once I created an account, I created a Databricks cluster and a ‘notebook’ to run Spark jobs.

## Database / Schema

All of the table fields were kept the same and the tables were individually created manually based off of the header row in the CSV files.

## ETL

I used the same dataset that I extracted for the Hive setup because of the size limitations that Spark has and also because the file's are just too large to upload on my internet speed.

I chose 500000 as my limit because it felt like a large enough of a dataset for User and Review. It was 50% of the User dataset and 12.5% of the Review dataset. All data was loaded into Spark tables using the GUI with no issues. The files were small enough (~70MB to 400MB) that uploading with the GUI was also very simple.

## Retrieval Queries

Find the average rating for Greek food in a city in SQL

SELECT city, AVG(stars) AS average\_rating, count(\*) AS business\_count

FROm business

WHERE categories LIKE ‘%Greek%’

GROUP BY city

ORDER BY business\_count DESC

LIMIT 100;

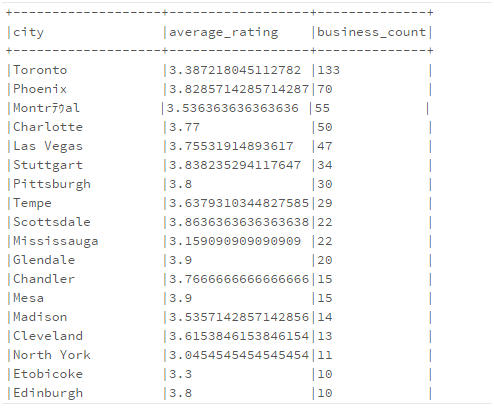
To run it as a Spark job we would insert it the table into a dataframe in Python and run the query in Python as such:

business\_df = sqlContext.read.format("com.databricks.spark.csv").option("delimiter", ",").option("header", "true").option("inferSchema", "true").load("/FileStore/tables/mrr5tmn91492134853356/business.csv")

business\_df.registerTempTable("business")

result = sqlContext.sql("SELECT city, AVG(stars) AS average\_rating, count(\*) AS business\_count FROM business WHERE categories LIKE '%Greek%' GROUP BY city ORDER BY business\_count DESC LIMIT 100")

result.show(result.count(), False)



## Data Quality / Consistency

We can clean and filter bad data in small chunks when we partition our table. Whenever possible it is ideal to simply fix the data before Spark Streaming has to do any processing with the data.

## Performance Optimization

# We can use cache() in order to fetch previously used data faster than reading from disk.

# Even without using cache(), the same query will be able to execute faster than the original query.

# Data serialization is an important factor for Java objects, but I am unsure about Python objects. Objects which consume a large number of bytes will greatly slow down the computation of the query.

# In general, collections of objects will be faster than some standard classes.

# RDDs will use significantly more memory than a Dataset or DataFrame would, so sticking to DataFrames in Python

# Elassandra

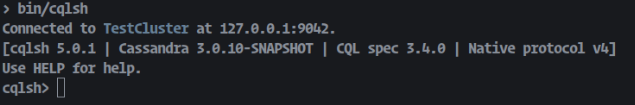
## Setup

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 so I installed Elassandra on that. 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. 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'.

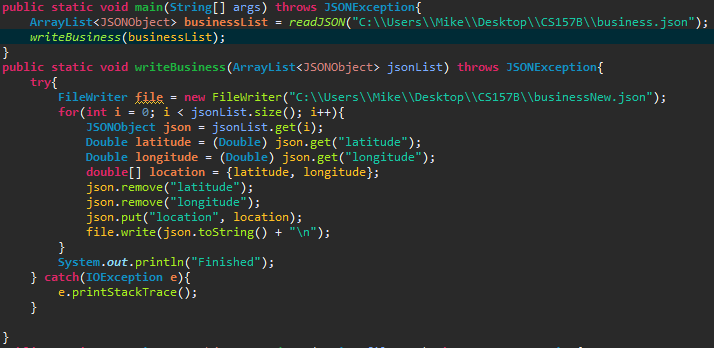


## Database / Schema

All of the table fields were kept the same and the tables were individually created manually based off of the header row in the CSV files.

## ETL

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.

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

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

);

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.")

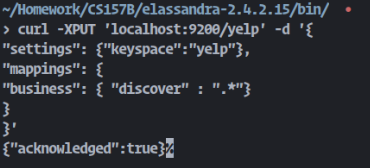
print("Importing checkin data...")

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.

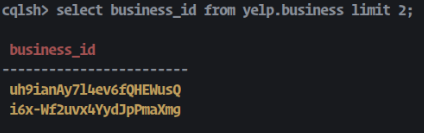
Below is an example of how we would map our data to Elasticsearch. Elassandra uses the keyspace yelp to map the Business table and discovers all the attributes of the Business table. We would also define the mappings for our other tables too like User or Tip.

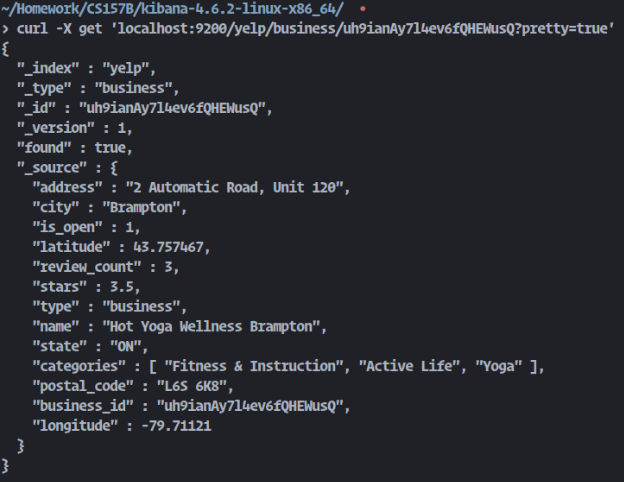


## Retrieval Queries

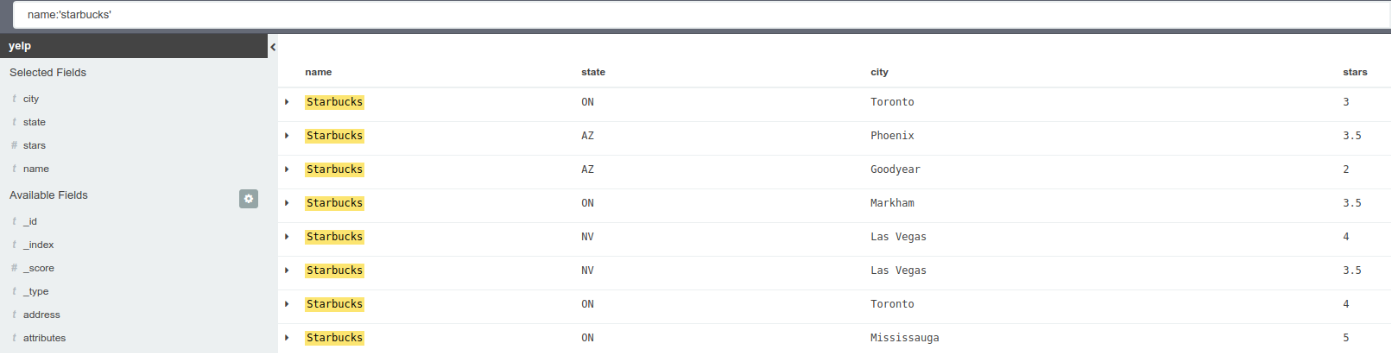
Elasticsearch provides very fast and loosely typed queries. Below I show the difference between a select query with CQL, a select query with Elasticsearch REST API, and a select query in Kibana.

**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.

## Performance Optimization

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.

# Technology Comparison

## SQL

Strengths -

* SQL is very good for simple queries and also for queries where the database is relatively small.
* SQL also has the large advantage of very well written documentation due to being used for so long.

Weaknesses -

* Scales vertically, which means that if new resources for a server need to be added, then downtime is required and there are limits defined by the hardware.
* Cascading updates can be detrimental to optimization due to constraint checks.

## Hive

Strengths -

* Scales horizontally so that it performs well over petabytes worth of data with a distribution across hundreds of clusters.
* Familiar SQL-like language for queries.

Weaknesses -

* Does not perform as well as SQL with simple queries.
* Updating data is complicated because of HDFS.

## Spark

Strengths -

* In-memory processing leads to fast batch processing.
* Jobs can be written in Java, Scala, and Python.

Weaknesses -

* No real-time processing.
* No file management system so it relies on another platform, in this case Databricks cloud based platform.

## Elassandra

Strengths -

* Near real time searching using Elasticsearch indexing.
* Mapping changes and re-indexes cause downtime in Elasticsearch, but using Cassandra as our primary data storage this is not an issue in Elassandra.

Weaknesses -

* Extremely heavy on RAM usage, recommended to have at least 32GB.
* Since Elassandra uses an old version of Elasticsearch, it cannot take advantage of the new version of Kibana to create the mapping for an index, thus all mapping is manual.