Data Management in Apache Cassandra and Apache HBase

Berufsspezifische Schlüsselkompetenzen in einer forschungsbezogenen Projektarbeit
Dr. Lena Wiese
WS 16/17

Institute of Computer Science
Georg-August-Universität Göttingen

Author:	GURYASH KAUR BAHRA
	21432826
	g.bahra@stud.uni-goettingen.de
	Applied Computer Science(Master, 5th semester)
Date:	February 28, 2017

A report on

Data Management in Apache Cassandra and Apache HBase

Submitted in partial fulfilment for the requirement of the course M.Inf. 1809

In

M.Sc. in Applied Computer Science

Submitted by

Guryash Kaur Bahra

21432826

Under the Guidance of

Dr. Lena Wiese

Department of Informatics

Faculty of Informatics and Computer Science

Georg-August-Universität Göttingen

Abstract

Apache Cassandra and Apache HBase are NoSQL databases, designed to run efficiently in distributed environments. While they both rely on the concept of column families in order to store data, their mechanisms for accessing and querying data is quite different. Apache Cassandra comes with the Cassandra Query Language (CQL) that is very similar to SQL. Apache HBase only has simple put and get methods to manipulate data.

Thus, on the one hand individual insertion methods are to be implemented for making the import of large data sets feasible. On the other hand, the format of existing data sets is to be adapted to fit the idea of column families.

In this project the following tasks are implemented: i) Importing the yelp academic data set into Apache Cassandra and HBase using appropriate data structures and import mechanisms, and ii) Benchmarking the performance of the databases and evaluate plausibility of the results with exemplary queries replicated from the queries written for Apache Drill (see Appendix B).

Contents

Introduction

NoSQL systems, as stated in [1], are designed in a way to meet the increasing volume, velocity, and variety of data. It stores and retrieves the data from the database that is not modelled around tables which are used in traditional databases, i.e., relational databases, and they may support SQL-like query languages (see [1]). In this report, we will look into two such databases, Apache Cassandra and Apache HBase, and understand how the data is stored and queried in such systems.

According to [2], Cassandra and HBase both are modelled around the concept of Google's Bigtable, and therefore, are column-oriented datastores. However, they both have different data model and different API (see [3]).

[4] states that the Apache Cassandra is a wide-row-store data store with denormalized model (i.e. any node can perform any operation). And, it consists of following three main objects: i) *keyspace* - a container for tables and indexes, ii) *table* - somewhat like a relational table, however, can store large volumes of data and provides fast row inserts and column level reads, and iii) *primary key* - is used to identify a row uniquely in a table (see [4]).

Apache HBase on the other hand, as in [5], is a key/value store and is defined as *sparse, consistent, distributed, multidimensional, sorted map.* HBase is called *map* as it maintains maps of keys to values (key $\rightarrow$ value) and these keys are used to identify values associated to it (see [5]). It is *sorted* as the values stored are sorted by the key (see [5]). It is *multidimensional* as the key used to store value comprises of row-key, column family name, column name, and time-stamp and so the mapping of the value to the key is, (row-key, column family, column, timestamp) $\rightarrow$ value (see [5]). HBase is *sparse* as it does not store NULL values as in most relational databases, so there will be no cell for a column that does not have any value (see [5]). This is because the value stored in a row is in the form of key $\rightarrow$ value mappings. It is *distributed* as the data can be spread over more than one machine and is *consistent* because the changes to the same row-key are atomic and therefore, a reader will always read the last written (and committed) values (see [5]).

Also, querying in these two systems are different, where Cassandra makes use of CQL (Cassandra Query Language) to store, update and retrieve objects; HBase makes use of get and put operations for manipulating (insert, update, delete and retrieve) the data.

The report, is about how the data is managed in Cassandra and HBase datastores. Section 2 describes what the *yelp* data is, and then, how it is imported and stored in Cassandra and HBase. In section 3, the queries which are written for Apache Drill are discussed, and further, how these queries are replicated for Cassandra and HBase. Section 4 describes the benchmarking for the two datastores, Cassandra and HBase, and conclusion is provided in section 5.

Storing Yelp Data

Before the data can be managed in Apache Cassandra and Apache HBase databases, let us first understand the data and its structure.

Yelp Data

The data is taken from Yelp website (see [6]). The file consists of three different types of json-objects. The first object is of *business* type and it contains information about the businesses, for example, the business ids, name of the businesses, etc. The second json-object is of type *user* and it contains information about the customers, for example, user ids, names, etc. And the third json-object is of *review* type and it contains information about the reviews associated with the businesses and with the users who reviewed those businesses. Therefore, this object contains business ids and user ids linked to the *business* and *user* json-objects respectively, along with review ids, review text, etc. (see Appendix A for the complete structure of these json-objects). There are in total of 474434 objects in the file, out of which, 13490 are *business* objects, 130873 are *user* objects and 330071 are *review* objects.

In the following sub-sections, 2.2 and 2.3, we will see how these three objects were modelled according to the Cassandra and HBase databases respectively.

Apache Cassandra

As mentioned in section 1, storing data in cassandra comprises of three main objects, these are keyspaces, tables, and primary key for each table in that keyspace. Therefore, the yelp data is stored in keyspace named yelpkeys. The three objects, business, review, and user, are stored in three different tables, `businessobjects, reviewobjects`, and `userobjects` respectively, and the attributes of the objects are stored in columns in their respective tables. And lastly, the primary keys, `business_id, review_id` and `user_id`, are defined respectively for each of *business, review* and *user* json-objects, or `businessobjects, reviewobjects` and `userobjects` tables.

From Appendix A.1 it can be seen that the *business* type object mainly consists of string data and therefore, are stored in text datatype columns in Cassandra. Then, there are number type data which are stored in integer type columns, and lastly, string arrays which are stored in list datatypes. As for the *review* type json object (see Appendix A.3), there are more of string data which are stored in text columns, then there is a number datatype stored in an integer column, and an object datatype named *votes*. The *votes* named datatype is associated with three attributes of number type, and to model *votes* attribute, unnesting is done, i.e., the top attribue *votes* is removed, and the nested attributes are stored in separate columns in Cassandra.

Note: The nested object datatypes are easy to model in Cassandra by the use of the concept of *super column*. However, this concept is not supported in HBase, and therefore, is not used in the project.

To put the data in Cassandra, the data store is connected using `Cluster` and `Session` objects available with the java language. To create keyspace and tables, `CREATE KEYSPACE` and `CREATE TABLE` commands from CQL are used respectively with the `Session` object. As for the columns, these are defined in the `CREATE TABLE` command, in it, the datatypes of columns are also mentioned with the column names. And the primary keys for each table are defined in the same, `CREATE TABLE`, command, where the `PRIMARY KEY` identifier is used for the column which is to store primary keys for that table.

Further, the yelp data file which is in .json format, and is read using `BufferedReader` object, which in turn uses `FileReader` object in its constructor call, and the same `FileReader` object uses `File` object which is used to locate the .json file stored on the system. The *votes* attribute is unnested using `replaceFirst` method of the `String` class.

Finally, the data is put into the data store using `INSERT INTO` command of the CQL. In it, the table name is specified to store its respective json object.

Note: Refer Appendix E for the complete implementation.

Apache HBase

For HBase, the connection is established using `Configuration, Connection` and `Admin` objects provided in java API. Then, as there is no concept of keyspaces as in Cassandra, the data is directly stored in the tables. Also, instead of creating three tables to put three different json objects, as done for cassandra, only one table is created in HBase (for this project) and the three different json objects are put in three different column families defined for that table. The table, `yelpkeys`, is created by using `HTableDescriptor` object (provided by java API) and by using the same `HTableDescriptor` object, the column families are added to the newly created table.

Before the data can be stored in the table, it is read from the .json file. And as mentioned in section 2.2, a `BufferedReader` object is used in the same manner as it is used for Cassandra.

As the data is stored in key-value pairs in HBase, therefore, the json objects are broken down into these pairs. Every attribute (and its value) of json objects forms key-value pairs, where attribute acts as a key and its value as key's value. The key-value pairs of a json object belong to the respective column family (in HBase) where the keys form the columns of that family and the value is stored as the value of that column. Every json object is associated with a row-key and the same row-key is used to access different key-value pairs of the same object.

Note: The row-key in HBase is similar to the primary key in Cassandra. For example, the `user_id` row-key in HBase of the *user* json object is similar to the `user_id` column, suffixed with `PRIMARY KEY` identifier, in Cassandra.

Similarly for HBase, the *votes* attribute is unnested using the `split` method of the `String` class and the *put* object is used to put the data in the data store.

Note: Refer Appendix E for the complete implementation.

Querying Yelp Data

In this section, the queries mentioned in Appendix B are discussed in correspondance with Cassandra and HBase data stores. The queries mentioned in Appendix B are used to analyze Yelp Dataset and is done using Apache Drill data store. In this section, we will also see why some of those queries are easily replicated for Cassandra and HBase, and why some aren't.

However, before queries are discussed with respect to Cassandra and HBase in sub-sections 3.2 and 3.3 respectively, the queries mentioned for Apache Drill in Appendix B are discussed in the sub-section 3.1.

Queries in Drill

The Appendix B lists all the queries which helps in analyzing the Yelp data and these are described as follows:

Query 1 is about reading all the columns of the first *business* object.

Query 2 uses an aggregate function, `SUM`, to add all the values of the `review_count` column of the `businessobjects` table.

In **query 3**, the `GROUP BY` keyword is used to count the number of records that are present for every city belonging to every state of the *business* object in the datastore. The data is displaced by sorting the value of the count in the descending order, with the use of the `ORDER BY` keyword to the `COUNT` function. And only the first ten values are displayed as an output by using the `LIMIT` keyword followed by a number (which decides the number of rows displayed in the output).

In **query 4**, the `GROUP BY` keyword is used to average the values of `review_count` column belonging to a particular star values (i.e., `stars` column of `businessobjects` table). And are arranged in the descending order of `stars` column values by using `ORDER BY` clause.

Query 5 is about reading values of `name, state, city` and `review_count` columns of *business* object. However, only those values are displayed where the value of `review_count` is greater than 1000, and is done by using `WHERE` clause. The result-set is arranged in descending order of `review_count` values by using the `ORDER BY` clause for the same, and the first ten values are displayed.

Query 6 is about reading values, name of the businesses and their Saturday opening and closing timings from the `businessobjects` table, and displaying only first ten values.

Query 7 uses the `COUNT` function to count the rows which contain restaurants as one of the categories for the *business* object.

Query 8 is about reading values of `name, state, city` and `review_count` columns of *business* object. However, only those values are displayed where one of the values of `category` column is restaurants, and is done by using `WHERE` clause. The result-set is arranged in descending order of `review_count` values by using the `ORDER BY` clause for the same, and the first ten values are displayed.

Query 9 is about reading business names (i.e. `name` column), to which fields those businesses are part of (i.e. `categories` column), and the total number of the categories present in the categories list of every business. To count the categories, `repeated_count` functionality of Apache Drill is used. Only those values are displayed which contains restaurants as one of the categories in that column. And only first ten values are displayed in the descending order of categories count.

In **query 10** `GROUP BY` function is used to group the first category from the categories list of every record, and to count the total number of records per category by using `COUNT` function. The output is displayed in the descending order of the counted number of the categories and only ten values are displayed.

Query 11 is about reading all the columns of the first *review* object.

Query 12 is divided into two parts, the main query and the sub-query. The execution starts first with the sub-query, then, the result of the sub-query is used as a condition in the `WHERE` clause of the main query, and finally, the result from the main query is displayed. In the sub-query, `business_id` is read from the `reviewobjects` table for which the sum of the cool votes is greater than 2000, and the ids are arranged in the descending order of the value of the total number of cool votes per business id. The main query is about reading the name of the businesses from the `businessobjects` table which belong to the result-set of the sub-query.

Query 13 is about creating view named `businessreviews` which combines data in `businessobjects` and `reviewobjects` tables.

Query 14 uses the `COUNT` function to count the number of rows in the `businessreviews` view created in the query 13. In it, `name, stars, state` and `city` columns are taken from business table, and `funny, useful, cool` and `date` columns are taken from review table. And, only those records are part of the result-set for which the business id of the business table is same as the business id of the review table.

Query 15 is about reading the business names and every category associated to it, from the `businessobjects` table. However, in the output, the categories are listed one at a time for every business name, and is done by using `FLATTEN` keyword to the `categories` column of the business table, and only 20 values are displayed.

Query 16 is divided into main query and the sub-query. In the sub-query, the `FLATTEN` keyword is used for the `categories` column of the business table to list every category of the record separately. And then, in the main query, the list is reduced by grouping the same category of different records together by using `GROUP BY` keyword. This list is then arranged in the descending order of the total count of the each category. And finally, top 10 categories along with their count are displayed as the output.

Queries in Cassandra

The Appendix C lists all the queries that are replicated for Cassandra and are discussed as follows:

Queries 1, 2 & 11 for **Apache Drill** (as in Appendix B) is replicated as **Queries 1, 2 & 7** for **Cassandra** (in Appendix C) respectively, and the only difference is the use of table name in the case of Cassandra, i.e., `businessobjects`, instead of path to .json file of yelp dataset for Drill.

Query 3 for **Drill** (as in Appendix B) is replicated as **Query 3** in Appendix C. However, it is implemented incompletely and differently than what is done for Apache Drill. To start with, the `GROUP BY` keyword cannot be used in Cassandra as it can for Drill. To implement group by functionality in Cassandra, user defined function(s) in combination with user defined aggregate are used. And group by can be used only for one column, so in the query it is used for `state` column and not for `state` and `city` columns together . The user defined function `review_states` is created to implement group by functionality, and in it, the total number of records for each state is calculated. The function takes *map* datatype as one of the parameters to count the records; *tuple* datatype as one of the parameters could have been a possible solution to implement group by for more than one column, however, user defined functions does not take *tuple* as a parameter. Then, the aggregate `review_states` with *map* datatype as its parameter, which is same as *map* datatype defined in `review_states` function, is created, and through this aggregate, the user defined function is called. Then, the `ORDER BY` keyword is not used in the query for Cassandra. This is because the clustering columns are not defined for the table, and so the `ORDER BY` functionality cannot be implemented. Note that the clustering columns are the ones for which the `ORDER BY` keyword can be used at the time of querying, and not for any other columns. Hence, query 3 returns the states with its total number of records as an output.

Query 4 for **Apache Drill** (as in Appendix B) could not be implemented for Cassandra as it is not possible to use `AVG` function along with `GROUP BY` functionality. Although `AVG` function can be used in UDFs or for *number* datatype columns. Also, `trunc` keyword is not there for Cassandra, and cannot with achieved through some manipulations of CQL like, using user defined functions.

Query 5 for **Apache Drill** (as in Appendix B) is implemented as **Query 4** for **Cassandra**. However, there are some differences is the implementation, and these are: i) instead of path to json object, the table name is provided for Cassandra, ii) `ORDER BY` clause cannot be used as there are no clustering column(s) in the table, and iii) `ALLOW FILTERING` keyword is used in Cassandra which is not used in Drill, this is required so as to filter the records according to the mentioned condition in the `WHERE` clause.

Query 6 for **Apache Drill** (as in Appendix B) could not be replicated for Cassandra as the yelp data does not include open and close timings on Saturday for businesses.

Query 7 for **Apache Drill** (as in Appendix B) is implemented as **Query 5** for **Cassandra**. However, the implementation is different from Apache Drill,, i.e., the index `categories_idx` is created on the `categories` column of the `businessobjects` table before the `SELECT` statement is executed. This is done in order to read and display only those records where `categories` column contains restaurants as one of the options. Note that `ALLOW FILTERING` keyword cannot be used for queries which use `CONTAINS` keyword, therefore, index is created. Although, with the index, `ALLOW FILTERING` can be used, however, this is not used in the query as the desired output is achieved without it.

Query 8 for **Apache Drill** (as in Appendix B) is replicated as **Query 6** for **Cassandra**. However, `ORDER BY` clause is not implemented for Cassandra as there are no clustering column(s). And this is implemented as query 5 for Cassandra.

Query 9 for **Apache Drill** (as in Appendix B) could not be implemented for Cassandra. This is because the number of elements in the `categories` column cannot be calculated for each record.

Query 10 for **Apache Drill** (as in Appendix B) could not be implemented for Cassandra. This is because first element of the `categories` column, which is of *list* datatype, could not be read.

Queries 12 & 16 for **Apache Drill** (as in Appendix B) could not be replicated as sub-querying is not handled in Cassandra. One possible solution could have been to execute the sub-query and the main query separately, i.e., the sub-query is executed first, then, the result of that query is used in the main query, and finally, it is executed. However, the approach did not work for implementing such type of queries.

Query 13 of **Apache Drill** (as in Appendix B) could not be replicated for Cassandra. This is because columns from two tables could not be combined in Cassandra. However, **query 8** for **Cassandra** shows that views can be created, and as an example, in query 8, a view `viewUserObjects` is created which include columns `name, type` and `review_count` from the table `userObjects` where `review_count` and `user_id` forms a primary key and `user_id` acts as a clustering column.

Query 14 of **Apache Drill** (as in Appendix B) could not be replicated for Cassandra. This is because the view could not be created in query 13, and so the data cannot be read. However, `viewUserObjects` view is created for Cassandra, and so, accordingly the data can be read, and is shown in **query 9** for **Cassandra**. In this, the data is read from all the three columns in the view, however, only those records are displayed in the output for which the review count value is equal to 23. And as the clustering column `user_id` is defined in query 8, the `ORDER BY` clause is used for the column, and hence, the output is arranged in the descending order of the `user_id`. In the output, only first five values are displayed.

Query 15 of **Apache Drill** (as in Appendix B) could not be replicated for Cassandra as *categories* attribute of *business* json-object is of list datatype (see Appendix A.1) and so, cannot be flattened, either by using `FLATTEN` keyword (as in for Drill) or by some manipulation of CQL, in Cassandra.

Queries in HBase

The Appendix D lists the queries written for HBase, however, these are different from the queries written for Apache Drill or Cassandra. This is because HBase doesnot support SQL like functionality or CQL functionality of Cassandra, it mainly carries out CRUD (create, read, update and delete) operations. The queries are discussed as follows:

Query 1 is to list all the tables in the HBase. This is done by using `listTables` function of `Admin` class.

Query 2 is to list all the tables which matches the given expression, "yelp.*", in the datastore. This is done by using `listTables` function with "yelp.*" as an argument, and the method is called by using object of the `Admin` class.

Query 3 is to check whether the table, yelpkeys, is disabled, and if not, then, it is disabled. `isTableDisabled` function with yelpkeys as an argument is used to check whether the table, yelpkeys, is disabled or not, and if `false` value is returned by the function then, `disableTable` function with yelpkeys as an argument is used to disable yelpkeys table. Both these methods are of `Admin` class.

Similarly, **query 4** is to check whether the table, yelpkeys, is enabled, and if not, then, it is enabled. And is done by using `isTableEnabled` function with yelpkeys as an argument to check whether the table, yelpkeys, is enabled or not, and if `false` value is returned by the function then, `enableTable` function with yelpkeys as an argument is used to enable yelpkeys table. These two methods belong to the `Admin` class.

Query 5 is to disable table or tables which match the expression, "yel.*". This is done by using function `disableTables` with "yel.*" as an argument, and the method belongs to the `Admin` class.

Similar to query 5, **query 6** is to enable table or tables which match the expression, "yel.*". This is done by using function `enableTables` with "yel.*" as an argument, and the method belongs to the `Admin` class.

Note: There could be many more such queries like, to modify column(s), to create namespace, etc., by using the methods provided by the `Admin` class.

Query 7 is to scan the whole table, yelpkeys, in the data store. This is done by using the `getScanner` function of the `Table` class, and the object of the class holds the value of the table in the data store which is to be scanned.

Query 8 is to read the column value of a column family for a given row-key. This is done by using function `get` of `Table` class called with object of `Get` class as an argument. The table object holds the value of the table to which the value is to be read. The get object in turn uses row-key of the row for which the value is to be read. This reads the complete row for that row-key, and is mentioned in part a in the appendix. Part b reads the value of a column of a column family. This is done by using `getValue` function to the result object, and to it, the column family and its column, for which the value is to be read, are passed as arguments.

Query 9 is to delete a particular row. This is done by using function `delete` of `Table` class called with object of `Delete` class as an argument. The delete object in turn uses row-key of the row that is to be deleted. And the table object holds the value of the table to which the delete operation is to be performed.

Query 10 is to update the value of a column. This is done by using function `put` to table object called with object of `Put` class as an argument. The table object holds the value of the table to which the value is to be updated. The put object in turn uses row-key of the row for which the value has to be changed. Then, the column family, the column and the new value are provided to the `addColumn` function of the `Put` class. The column family and the column values are provided to the `addColumn` function for which the value is to be updated.

Benchmarking

In this section, the performance of the queries in the two datastores are discussed. To calculate the time taken by the queries to run, the *stopwatch* package[1] is used. Start and stop functions of the `Stopwatch` class are used to calculate the time. The `start` function is called before the query is executed, this is when the timer starts, and as soon as query is executed, the `stop` function is called, this is when the timer stops. The time is measured in milliseconds.

Note: The time measured for the queries give rough estimations about their executions. This is just to know which queries took more time and which took less comparatively.

Queries are run on Ubuntu 16.04.2 LTS system with 8gb RAM and 1tb of hard disk. Cassandra's version is 3.9 and HBase's version is 1.0.3.

The time taken by queries in Cassandra is between 4 milliseconds to 561 milliseconds depending on their functionality. Query 1 takes around 7 to 70 milliseconds to execute, and so are queries 7, 8 & 9. These queries are simple and fastest of the lot. Query 2 took around 200 milliseconds, and queries 4, 5 & 6 took around 120 milliseconds to run. The maximum time is taken by query 3 and is around 520 milliseconds. This is because the user defined aggregate is used to get the output.

[1]com.google.common.base.Stopwatch

The least time is taken by query 2 in HBase, and is around 9 milliseconds. Then, it is query 9 which took around 13 milliseconds to execute, after this, queries 7, 8 & 10 took around 50 milliseconds to run. The maximum time is taken by queries 3 & 5, and is around 1200 milliseconds. And as for the queries 1, 4 & 6, they took around 400 milliseconds for their execution.

Note: Performances of the two datastores cannot be compared as both of them run different set of queries.

Conclusion

To conclude the report, the data is read from the .json file using java API, and is stored directly into their respective tables for Cassandra and as for HBase, json objects are broken down into key-value pairs and are stored in their respective column families in the table. Then, the queries written for Apache Drill are replicated for Cassandra and not for HBase. This is because Cassandra uses CQL for querying which is similar to SQL, whereas for HBase, it does not have any such querying language. It uses get, put and delete commands for CRUD operations. Also, not all queries could be replicated for Cassandra as there are some functionalities which could not be implemented. And finally, benchmarking is done in order to know which queries took more time than the others.

References

[1] Birendra Kumar Sahu. A real comparison of nosql databases hbase, cassandra and mongodb. `https://www.linkedin.com/pulse/real-comparison-nosql-databases-hbase-cassandra-mongodb-sahu`, 14 June 2015.

[2] Rick Grehan. Big data showdown: Cassanddra vs. hbase. `http://www.infoworld.com/article/2610656/database/big-data-showdown--cassandra-vs--hbase.html?page=2`, 2 April 2014.

[3] Paolo Atzeni, Francesca Bugiotti, and Luca Rossi. Uniform access to nosql systems. In *Information Systems*, volume 43, pages 117–133, July 2014.

[4] A brief introduction to apache cassandra. `https://academy.datastax.com/demos/brief-introduction-apache-cassandra`, 2 January 2015.

[5] Lars Hofhansl. Hbase: Introduction to hbase. `http://hadoop-hbase.blogspot.de/2011/12/introduction-to-hbase.html`, 29 December 2011.

[6] Yelp academic dataset. `http://www.yelp.com/academic_dataset`.

Appendix A

Yelp Academic Dataset

Yelp business data

```
{‘type’: ‘business’,
‘business_id’: (business id),
‘name’: (business name),
‘neighborhoods’: [(hood names)],
‘full_address’: (localized address),
‘state’: (state),
‘city’: (city),
‘latitude’: latitude,
‘longitude’: longitude,
‘stars’: (star rating, rounded to half-stars),
‘review_count’: (review count),
‘photo_url’: photo url,
‘categories’: [(localized category names)],
‘url’: url,
‘schools’: [(localized school names)],
‘open’: True / False}
```

Yelp user data

```
{‘type’: ‘user’,
‘user_id’: (user id),
‘name’: (first name),
‘url’: url,
‘average_stars’: (floating point average, like 4.31),
‘review_count’: (review count),
‘votes’:
    {‘funny’: (count),
    ‘useful’: (count),
    ‘cool’: (count)}}
```

Yelp review data

```
{‘type’: ‘review’,
‘business_id’: (business id),
‘review_id’: (review id),
‘user_id’: (user id),
‘text’: (review text),
‘stars’: (star rating, rounded to half-stars),
‘date’: (date of review, formatted like ‘2012-03-14’),
‘votes’:
    {‘funny’: (count),
    ‘useful’: (count),
    ‘cool’: (count)}}
```

Appendix B

Analyzing the Yelp Academic Dataset[2]

1. **View the contents of the Yelp business data**

   Query 1:

   ```
   SELECT * FROM dfs.‘<path-to-yelp-data>/business.json’ LIMIT 1;
   ```

2. **Explore the business data set further**

   Query 2: Total reviews in the dataset

   ```
   SELECT SUM(review_count) AS totalreviews FROM dfs.‘/<path-to-yelp-data>
   /business.json’;
   ```

   Query 3: Top states and cities in total number of reviews

[2]https://drill.apache.org/docs/analyzing-the-yelp-academic-dataset/

```
SELECT state, city, COUNT(*) totalreviews FROM dfs.`/<path-to-yelp-data>
/business.json` GROUP BY state, city ORDER BY COUNT(*) DESC LIMIT 10;
```

Query 4: Average number of reviews per business star rating

```
SELECT stars, trunc(AVG(review_count)) reviewsavg FROM
dfs.`/<path-to-yelp-data>/business.json` GROUP BY stars ORDER BY stars
DESC;
```

Query 5: Top businesses with high review counts (> 1000)

```
SELECT name, state, city, `review_count` FROM dfs.`/<path-to-yelp-data>
/business.json` WHERE review_count > 1000 ORDER BY `review_count` DESC
LIMIT 10;
```

Query 6: Saturday open and close times for a few businesses

```
SELECT b.name, b.hours.Saturday.`open`, b.hours.Saturday.`close` FROM
dfs.`/<path-to-yelp-data>/business.json` b LIMIT 10;
```

3. **Explore the restaurant businesses in the data set**

Query 7: Number of restaurants in the dataset

```
SELECT COUNT(*) AS TotalRestaurants FROM dfs.`/<path-to-yelp-data>
/business.json` WHERE true=repeated_contains(categories,`Restaurants`);
```

Query 8: Top restaurants in number of reviews

```
SELECT name, state, city, `review_count` FROM dfs.`/<path-to-yelp-data>
/business.json` WHERE true=repeated_contains(categories,`Restaurants`)
ORDER BY `review_count` DESC LIMIT 10;
```

Query 9: Top restaurants in number of listed categories

```
SELECT name, repeated_count(categories) AS categorycount,
categories FROM dfs.`/<path-to-yelp-data>/business.json`
WHERE true=repeated_contains(categories,`Restaurants`) ORDER BY
repeated_count(categories) DESC LIMIT 10;
```

Query 10: Top first categories in number of review counts

```
SELECT categories[0], COUNT(categories[0]) AS categorycount FROM
dfs.`/<path-to-yelp-data>/business.json` GROUP BY categories[0] ORDER
BY COUNT(categories[0]) DESC LIMIT 10;
```

4. **Explore the Yelp reviews dataset and combine with the businesses**

Query 11: Take a look at the contents of the Yelp reviews dataset

```
SELECT * FROM dfs.`/<path-to-yelp-data>/review.json` LIMIT 1;
```

Query 12: Top businesses with cool rated reviews

```
SELECT b.name FROM dfs.`/<path-to-yelp-data>/business.json`
b WHERE b.business_id IN (SELECT r.business_id FROM
dfs.`/<path-to-yelp-data>/review.json` r GROUP BY r.business_id HAVING
SUM(r.votes.cool) > 2000 ORDER BY SUM(r.votes.cool) DESC);
```

Query 13: Create a view with the combined business and reviews datasets

```
CREATE OR REPLACE VIEW dfs.tmp.businessreviews AS SELECT b.name,
b.stars, b.state, b.city, r.votes.funny, r.votes.useful,
r.votes.cool, r.‘date’ FROM dfs.‘/<path-to-yelp-data>/business.json’
b, dfs.‘/<path-to-yelp-data>/review.json’ r WHERE
r.business_id=b.business_id;
```

Query 14: The total number of records from the view

```
SELECT count(*) AS Total FROM dfs.tmp.businessreviews;
```

Query 15: Get a flattened list of categories for each business

```
SELECT name, FLATTEN(categories) AS category FROM
dfs.‘/<path-to-yelp-data>/business.json’ LIMIT 20;
```

Query 16: Top categories used in business reviews

```
SELECT celltbl.catl, COUNT(celltbl.catl) categorycnt FROM (SELECT
FLATTEN(categories) catl FROM dfs.‘/<path-to-yelp-data>/business.json’)
celltbl GROUP BY celltbl.catl ORDER BY COUNT(celltbl.catl) DESC LIMIT 10;
```

Appendix C

Querying in Cassandra

Query 1: View the contents of the Yelp business data

```
SELECT * FROM businessobjects LIMIT 1;
```

Query 2: Total reviews in the business dataset

```
SELECT SUM(review_count) AS totalreviews FROM businessobjects;
```

Query 3: Count total number of reviews for particular state in Yelp business data

```
CREATE OR REPLACE FUNCTION review_states(state_review map<text,int>, state
text) CALLED ON NULL INPUT RETURNS map<text,int> LANGUAGE JAVA AS $$ Integer
count = (Integer) state_review.get(state); if (count == null) count = 1; else
count++; state_review.put(state, count); return state_review; $$;

CREATE OR REPLACE AGGREGATE review_states(text) SFUNC review_states STYPE
map<text,int> INITCOND {};

SELECT review_states(state) FROM businessobjects;
```

Query 4: Businesses with high review counts (> 1000)

```
SELECT name, state, city, review_count FROM businessobjects WHERE review_count
> 1000 ALLOW FILTERING;
```

Query 5: Number of restaurants in the business dataset

```
CREATE INDEX categories_idx ON businessobjects(categories);

SELECT COUNT(*) AS TotalRestaurants FROM businessobjects WHERE categories
CONTAINS 'Restaurants';
```

Query 6: Restaurants in number of reviews

```
SELECT name, state, city, review_count FROM businessobjects WHERE categories
CONTAINS 'Restaurants';
```

Query 7: View the contents of the review data

```
SELECT * FROM reviewobjects LIMIT 1;
```

Query 8: Creating Materialized view of userobjects

```
CREATE MATERIALIZED VIEW IF NOT EXISTS viewUserObjects AS SELECT name, type,
review_count FROM userobjects WHERE review_count IS NOT NULL PRIMARY KEY
(review_count,user_id);
```

Query 9: Reading values from Materialized view of userobjects

```
SELECT name, review_count, user_id FROM viewUserObjects WHERE review_count = 23
ORDER BY user_id DESC LIMIT 5;
```

Appendix D

Querying in HBase

Query 1: List all the tables in the data store

```
admin.listTables();
```

Query 2: List all the tables which matches the given expression, "yelp.*"

```
admin.listTables(``yelp.*");
```

Query 3: To check whether the table, yelpkeys, is disabled, and if not, then it is disabled

```
admin.isTableDisabled(TableName.valueOf(``yelpkeys"));

admin.disableTable(TableName.valueOf(``yelpkeys"));
```

Query 4: To check whether the table, yelpkeys, is enabled and if not, then it is enabled

```
admin.isTableEnabled(TableName.valueOf(``yelpkeys"));

admin.enableTable(TableName.valueOf(``yelpkeys"));
```

Query 5: To disable table(s) which match the given expression, "yel.*"

```
admin.disableTables(‘‘yel.*");
```

Query 6: To enable table(s) which match the given expression, “yel.*”
```
admin.enableTables(‘‘yel.*");
```

Query 7: To scan the whole table in the data store
```
table.getScanner(scan);
```

Query 8: To get column value of a column family for a given row-key
```
a.  table.get(new Get(rowKey.getBytes()));

b.  Bytes.toString((table.get(g)).getValue(columnFamily.getBytes(),
qualifier.getBytes()));
```

Query 9: To delete a particular row
```
table.delete(new Delete(rowKey.getBytes()));
```

Query 10: To update the value of a column
```
table.put(new Put(rowKey.getBytes()).addColumn(colFamily.getBytes(),
qualifier.getBytes(), newVal.getBytes()));
```

Appendix E

Java programming to work with Yelp Academic Dataset

For the complete coding of the project refer `https://github.com/Guryash/Nosql-Project`.