UNIVERSITÀ DEGLI STUDI
DI GENOVA

Department of Computer Science
ADM - Advanced Data Management A.Y. 2018/2019

Easy Beer

Riccardo Bianchini (S 4231932)
Andrea Canepa (S 4185248)

Beer is proof that God loves us and wants us to be happy
— Benjamin Franklin

16 September 2019

Contents

1 Introduction

1.1 Application

To understand the workload we would like to present in the rest of our work it is necessary to spend some words on the *application* we imagine rely on our system. When we spend a night out with our friends at a pub, we would like to enjoy the entire experience, and *what is better than a good* ***beer*** *with a good company* ? But, as we know, everyone is different in taste, so the aim of our application is to help people in choosing the beer that they like the most. The application could be used at different levels: since we consider very specific attributes of beers, not only the newbies could learn something new about their favourite drink, but also expert could satisfy their peculiar tastes.
Moreover we would like to offer a tool to do some analysis and comparisons between beers, styles and whatever comes up in mind to the user.

1.2 Data

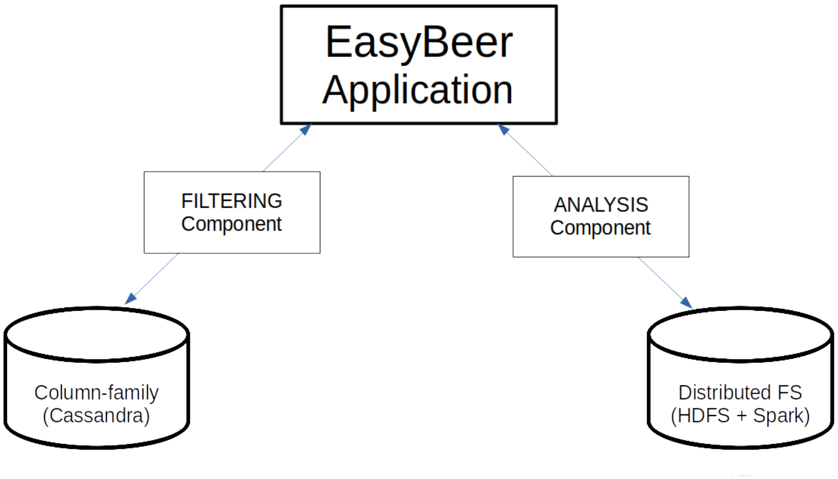
Our application is based on a dataset found on Kaggle at the URL

`https://www.kaggle.com/jtrofe/beer-recipes`

We used only the file `recipe_data.csv` because the file `styleData.csv` was useless for our purposes. In the dataset are reported many technical features useful to *brew*, such as the *boilgravity* or the *mash thickness* of a beer, because a beer expert can use these information to choose the right one. Our application use these information to give the user the best experience possible.

1.3 Structure of the System

First of all, we observe that our application has different requirements in terms of operations. We propose a **polyglot system** in order to satisfy those needs.



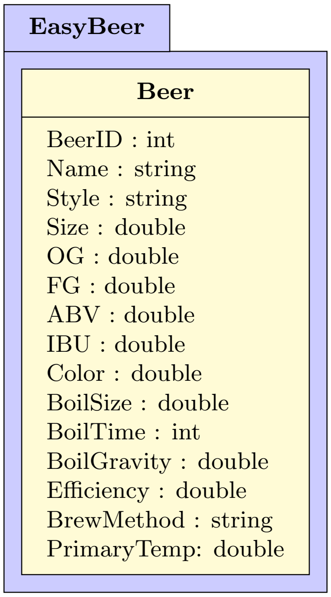
As we can see in the photo we implemented two different components:

- **Filtering component**: in this case we adopted `Cassandra` to cope the workload which is *read-intensive* and it is not necessary to have *consistency* all the time because writes operations are not frequent and they do not affect too much the results of our queries. Furthermore `Cassandra` *parallelize* well filter queries.

- **Analysis component**: for this purpose we chose `HDFS` because we needed to perform some complex and heavy operations on all dataset, often iteratively to build models or to clusterize our data (typical *batch operations*), so we do not claim to grant *availability* in each moment since those results are seldom checked.

1.4 Conceptual schema

This is the conceptual schema of the domain, it represent the domain of our application. Here we can appreciate all the technical features of the beers upon which we made our decision in terms of aggregates and workloads. Our ideal application tries to exploit all the properties shown below in a smart way.



2 Dataset

2.1 Preprocessing

The dataset we chose wasn't suitable for our application, so a preprocessing phase was required. We had to discard many rows and some columns. At the end, we used the following features:

1. BeerID
2. Name
3. Style
4. Size
5. OG
6. FG
7. ABV
8. IBU
9. Color
10. BoilSize
11. BoilTime
12. BoilGravity
13. Efficiency
14. BrewMethod
15. PrimaryTemp

The columns *URL*, *StyleID* and *UserId* were useless for our application, so we discard them. For what concerns the columns *MashTickness*, *PitchRate*, *PrimingMethod*, *PrimingAmount*, those were not specified in too many rows ("N/A"), so we couldn't use them. The *SugarScale* column was useless, since we converted all rows to *Plato degrees*, a well-known unit measure among the brewers. In order to avoid problems with non standard symbols, we kept only the names and the styles longer than 3 characters, that begin with letters and are formed only with letters, numbers, spaces and dashes.

We can see in the directory ***dataset*** how we choose to split the entire dataset in order to examine it without considering unsuitable data for each specific task. Each file was generated starting from *recipe_data.csv* with the help of the scripts that we are going to describe in the next sections. Those files are:

- `recipe_data.csv`: the original dataset downloaded from *Kaggle*;
- `beers.csv`: dataset without noise, generated with the bash script;
- `beers_hdfs.csv`: same as before but without the first line, to do the queries with *Spark*;
- `clustering.csv`: contains the data for the clustering experiments, all numeric features;
- `ml_data.csv`: dataset for the machine learning task in which the style was encoded in a naive way;
- `colors.csv`: contains the color in form of string, wrt Standard Reference Method (SRM) for Beer Color Evaluation;
- `style.csv`: contains the association between the color encoded as a number and the real style name as a string.

2.2 `preprocessing.sh`

This script essentially does 4 different operations on the starting dataset:

1. select only a few features, with the help of the bash command `cut` that splits a line given a delimiter and choose the fields specified as argument;
2. delete all lines that contains non-printable characters in the *name* or *style* columns, since those are not standard and not always are recognized by the system;
3. converts the *boil gravity* values express in *Specific Gravity* to *Plato Degrees*, since the latter is the standard unit of measure for this property and it is useful to catalog the beers;
4. delete all rows that contains "N/A" values because are unsuitable for analytical purpose.

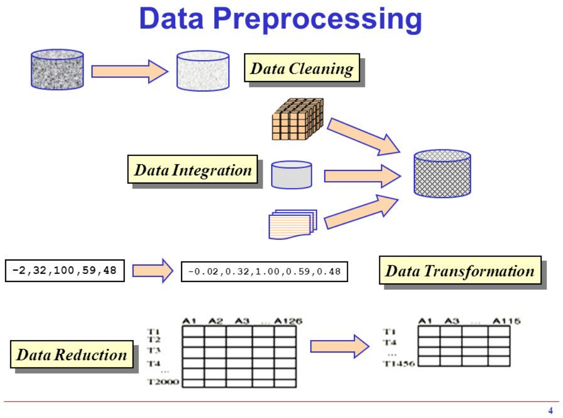
By the end of the execution of this script our dataset is reduced by approximately 30.5%, leaving us with ~51000 beers with respect to the initials ~73000.
Regarding the third point, it is interesting notice the conversion factor between the two measure units. The equivalence is expressed through the following equation:

$$\boxed{Plato = 135.997sg^3 - 630.272sg^2 + 1111.14sg - 616.868}$$

where **sg** stands for *Specific Gravity*

2.3 `preprocess_ml.sh`

This script is easier than the previous one and we used only once in order to create a suitable dataset for machine learning task. The aim was to select only the numerical features and to encode the *style* of a beer as a number in a trivial way. We used the data contained in the files `beers.csv`(in which we erased the first line) and `style.csv` to create `ml_data.csv`.



3 Cassandra

3.1 Workload

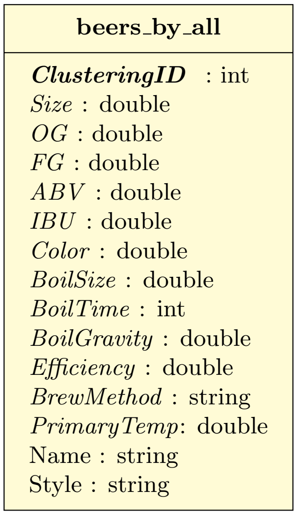
One of the aims of our application is to apply filters to the dataset, in order to select the desired subset of beers, so the workload consist in a list of very simple filter requests. This is the workload:

1. Retrieve all beers whose **name** starts with a given string
2. Retrieve all beers whose **style** contains a given string
3. Remove all beers with **alcohol by volume** less than 2
4. Retrieve all beers with **alcohol by volume** in a given range
5. Get the maximum, minimum and average value of **alcohol by volume** among all beers
6. Delete beers with **boil gravity** out of range
7. Show all the **analcoholic** beers
8. Show all the **light** beers
9. Show all the **normal** beers
10. Show all the **special** beers
11. Show all the **double-malt** beers
12. Remove all beers with **color** value less than 2
13. Retrieve all beers with **color** in a given range
14. Retrieve beers with a named **color**
15. Get the maximum, minimum and average **color** among all beers
16. Remove all beers with **international bitterness unit** smaller than 1
17. Remove all beers with **international bitterness unit** grater than 150
18. Retrieve all beers with **international bitterness unit** in a given range
19. Get the maximum, minimum and average **international bitterness unit** among all beers
20. Given a **style** retrieve the average **alcohol by volume**
21. Given a **style** show the lightest and the darkest **color**
22. Show in alphabetical order the beers of a given **style**
23. Count how many beers share a given **style**
24. Show all **styles** present in the dataset

3.2 Aggregates

This workload is implemented with 5 different aggregates: *beers_by_all*, *beers_by_idabv*, *beers_by_idboilgravity*, *beers_by_idcolor*, *beers_by_idibu*, *beers_by_style*. The implemented aggregates have an attribute added by us, ClusteringId, that is useful to better *parallelize* the execution on the cluster. We added a number from 1 to 3 to each row, and then we used this attribute as partitioning key. This choice was mandatory in the majority of the aggregates, because otherwise the load would be unbalanced in the cluster. In the following *UML* class diagrams attributes are represented in order as in the create table, the partitioning key is in bold text, the primary key is in italic font. The queries are filters, so the aggregates are chosen in order to have in first positions the attributes on which the filters are applied.

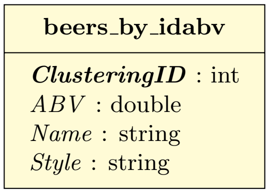
The filters 1 and 2 are implemented with the aggregate *beers_by_all*, here described:



beers_by_all

ClusteringID : int
Size : double
OG : double
FG : double
ABV : double
IBU : double
Color : double
BoilSize : double
BoilTime : int
BoilGravity : double
Efficiency : double
BrewMethod : string
PrimaryTemp: double
Name : string
Style : string

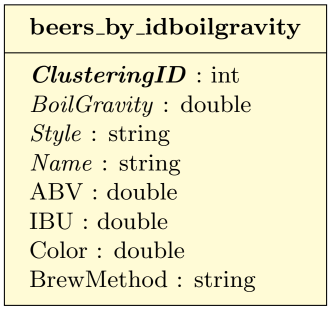
The filters 3 to 5 are implemented with the aggregate *beers_by_idabv*, here described:



beers_by_idabv

ClusteringID : int
ABV : double
Name : string
Style : string

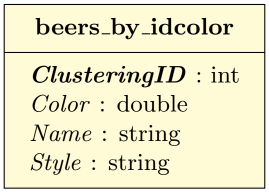
The filters 6 to 11 are implemented with the aggregate *beers_by_idboilgravity*, here described:



beers_by_idboilgravity

ClusteringID : int
BoilGravity : double
Style : string
Name : string
ABV : double
IBU : double
Color : double
BrewMethod : string

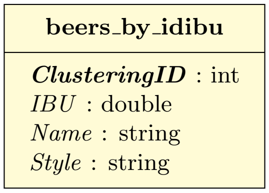
The filters 12 to 15 are implemented with the aggregate *beers_by_idcolor*, here described:



beers_by_idcolor

ClusteringID : int
Color : double
Name : string
Style : string

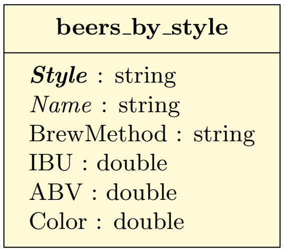
The filters 16 to 19 are implemented with the aggregate *beers_by_idibu*, here described:



beers_by_idibu

ClusteringID : int
IBU : double
Name : string
Style : string

The filters 20 to 24 are implemented with the aggregate *beers_by_style*, here described:



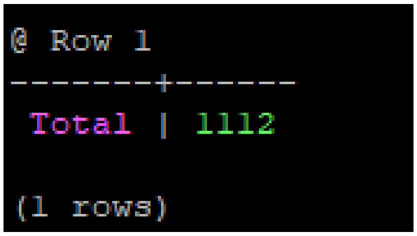
beers_by_style

Style : string
Name : string
BrewMethod : string
IBU : double
ABV : double
Color : double

3.3 Example

This is the implementation of the query number 23, with its result:

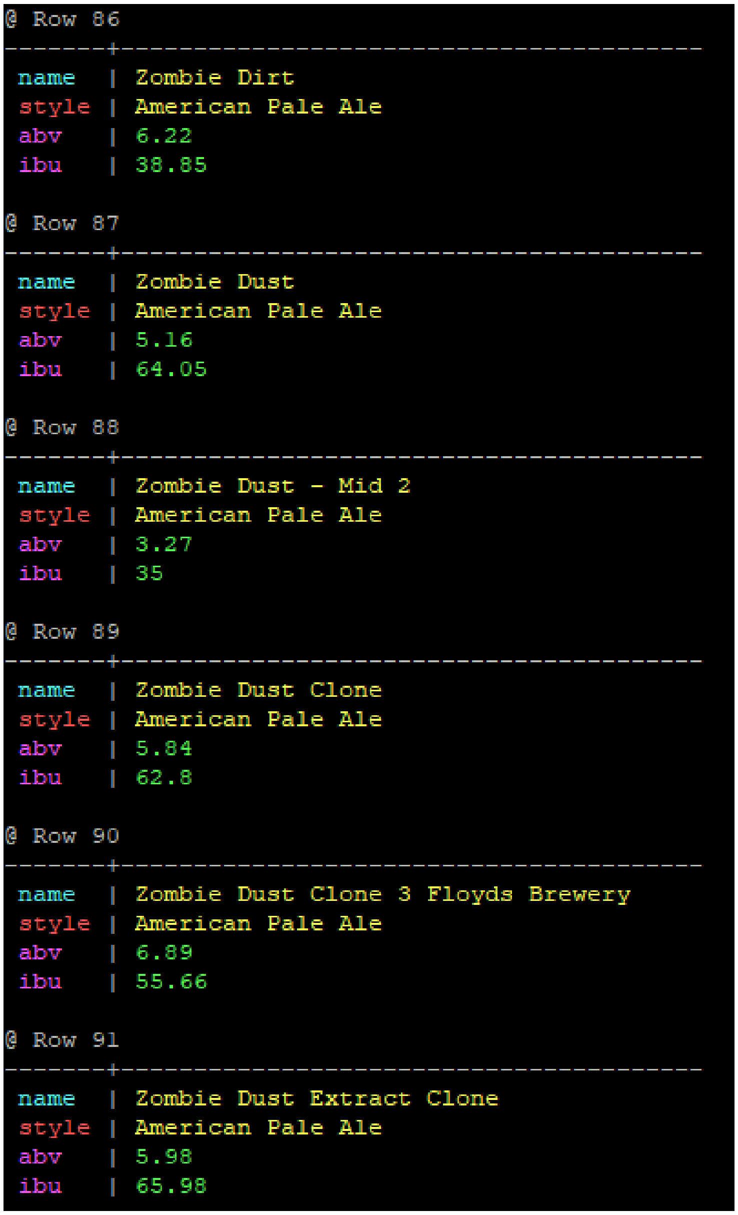
```sql
-- Count how many beers share a given style
SELECT COUNT(*) AS "Total"
FROM beers_by_style
WHERE style = 'Blonde Ale';
```



```
@ Row 1
-------+------
 Total | 1112

(1 rows)
```

This is the implementation of the query number 22:

```sql
-- Show in alphabetical order the beers of a given style
SELECT name, style, abv, ibu
FROM beers_by_style
WHERE style = 'American Pale Ale'
ORDER BY name;
```



```
@ Row 86
-------+-----------------------------------------
 name  | Zombie Dirt
 style | American Pale Ale
 abv   | 6.22
 ibu   | 38.85

@ Row 87
-------+-----------------------------------------
 name  | Zombie Dust
 style | American Pale Ale
 abv   | 5.16
 ibu   | 64.05

@ Row 88
-------+-----------------------------------------
 name  | Zombie Dust - Mid 2
 style | American Pale Ale
 abv   | 3.27
 ibu   | 35

@ Row 89
-------+-----------------------------------------
 name  | Zombie Dust Clone
 style | American Pale Ale
 abv   | 5.84
 ibu   | 62.8

@ Row 90
-------+-----------------------------------------
 name  | Zombie Dust Clone 3 Floyds Brewery
 style | American Pale Ale
 abv   | 6.89
 ibu   | 55.66

@ Row 91
-------+-----------------------------------------
 name  | Zombie Dust Extract Clone
 style | American Pale Ale
 abv   | 5.98
 ibu   | 65.98
```

4 Spark

4.1 General considerations

Part of our work consists of analytical queries. We think that the best way to implement this kind of workload is using *Spark* because, thanks to its own data structure, it makes the work of the programmer easier and produce more readable code. We choose to adopt *Java* as main language to write the *driver* programs.
We would like to spend some words on the internal *data structure*:

- `Tuple2<?,?>`: simply model a tuple with two elements;
- `JavaRDD<?>` and `JavaPairRDD<?,?>`: collections of elements or pair of elements generated from the datasets distributed in *HDFS*;
- `SparkSession`: the entry point of the program.

In general we would like to compute some values aggregating beers by some value and comparing them in order to understand some sort of trends that we could only understand if we take in account all the dataset. We tried to implement some interesting queries that we were not able to do in *Cassandra*. In fact, as we know, the iterative tasks are the workhorse of *Spark* and we have exploited this fact in order to present some heavy machine learning and data mining jobs (see sections 4.4 and 4.5).
The only shrewdness we have to take is about the datasets: in order to work with our data we have to erase the first line, the one with the columns' titles, otherwise we got some parse exception thrown by the driver.

4.2 Workload

Thinking about a possible application that relies on our storage system we detect 6 possible analytical queries, which are:

1. For each **style**, select the greatest **ABV**;
2. Select the most frequent **color** and the beers having that color;
3. Given a **style**, compute the average and the standard deviation of the **color**;
4. Among all beers that share a **brew method**, compute the average of **apparent attenuation** (in %) and the average **real attenuation** (in %);
5. Retrieve the most popular **brew method** among double malt beers[1] and how many beers share it;
6. Show the pair of beer's **style** with most similar **ABV** (on average).

As a memorandum for the 4^{th} query, we notice that the *apparent attenuation* and the *real attenuation* are derived dimension which formula are:

$$\boxed{AA\% = \frac{OG - FG}{OG} \cdot 100} \quad \boxed{RA\% = \frac{AA\%}{1.22}}$$

where AA and RA are, respectively, the apparent and the real attenuation, OG and FG are, respectively, the original and the final gravity. This relation, in percentage, gives us the opportunity to understand how much beer have been *"tone down"*, i.e. how much sugar percentage have been turned into alcohol during the fermentation.
In addition to those queries we got also two tasks that require a special attention: we tried to divide the beers in clusters exploiting two well known ***data-mining*** algorithms and we built a model to try to understand the style of a beer given all the others attributes with the help of a famous ***machine learning*** technique.

4.3 `compile.sh`

This script was realized to simplify the deployment of our code. In fact, in this short bash program we gave the directives to the *Java* compiler to produce the *.class* file starting from our source code, then it makes a *jar* archive compressing the directory *spark* and, eventually, submit the result to *Spark* in order to execute it. The user has only to invoke `compile.sh` from the directory in which is located and giving as its only argument the name of the file containing the driver program that have to be executed.

```
$ ./compile.sh <filename>
```

[1] *double malt beers* are those whose **ABV** is greater than 3.5° and **boil gravity** is greater than 14.5°Plato

4.4 Machine Learning

One of the most controversial question that the brewers ask themselves when they create a new beer is: to which *style* it belongs? Only then he can prove itself that he has done a great job. We tried to build a model to identify the style of beer giving some information a-priori. Keep in mind that our dataset is filled up with *homemade beers*, so not always people are aware of their products.

Obviously this is quite an expensive iterative task, so *Spark* is the best choice to satisfy this kind of request, also because its data structure are built to support parallel operations to reduce the time of execution. The algorithm we adopt is ***decision tree***. At each step we search for the feature and the point at which the data are better split in order to minimize the error function at that point (the algorithm is, in fact, greedy), in our case we tried two different functions. f_i is the frequency of label i at a node and C is the number of unique labels:

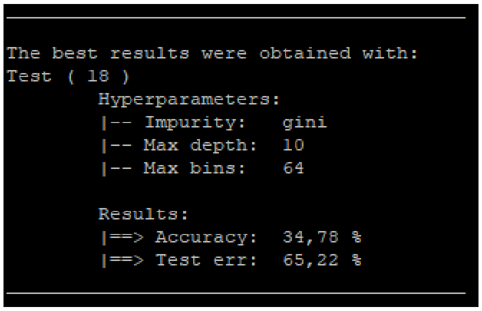
- the *Gini* coefficient:

$$\boxed{\sum_{i=1}^{C} f_i(1-f_i)}$$

- *entropy*:

$$\boxed{\sum_{i=1}^{C} -f_i log(f_i)}$$

We tried different *depths* and different *number of elements per split* in order to find the best model. We slice our dataset in training set and test set, respectively 65% and 35% of all data, to train the model. Our experiment, unfortunately, was not really successful, in fact the best result was an *accuracy* of 34.78%.



```
The best results were obtained with:
Test ( 18 )
        Hyperparameters:
        |-- Impurity:   gini
        |-- Max depth:  10
        |-- Max bins:   64

        Results:
        |==> Accuracy:  34,78 %
        |==> Test err:  65,22 %
```

4.5 Clustering

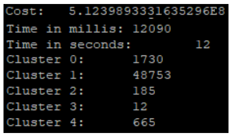
Another possible way to explore our data is **clustering**. Could be an interesting point if we find similar "*families*" of beers, since this functionality could be exploit to suggest the final user what sort of beer he or she could appreciate. We would like to implement this operation using a mathematical approach. In order to see if the beers are divided in similar groups(*clusters*), we used 2 variants of the same algorithm that we have seen in the Machine Learning course:

- **K-Means**
- **Bisecting K-Means**

Both contain an implementation of *K-Means++*, that is a technique to initialize the centers of the clusters, which is optimized to run in parallel, in fact it is called *K-Means*||. The main difference between the two algorithms is how the cluster are created: both of them are *hierarchical*, but the former is *agglomerative* while the latter is *divisive*. Obviously, the results should be different, and, usually, also the elapsed time. We chose to print the **cost** of the algorithm which is a metric to understand how much the result is reliable, it is computed using the **WSSSE** (*Within Set Sum of Squared Errors*), i.e. the sum of squared distances of points to their nearest center:

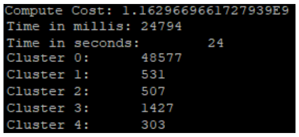
$$\boxed{J \;=\; \sum_{n=1}^{N}\sum_{k=1}^{K} r_{nk}\|x_n - \mu_k\|}$$

where N is the number of points, K is the number of clusters, r_{nk} is 1 if the point n belong to cluster k, 0 otherwise, and μ_k is the nearest center of a cluster. We did some experiments but we observe that the cost was always very high, so we could deduce that the clustering was not so good in our dataset. Below we can see the results of a test with 5 clusters.



```
Cost:   5.1239893331635296E8
Time in millis: 12090
Time in seconds:        12
Cluster 0:      1730
Cluster 1:      48753
Cluster 2:      185
Cluster 3:      12
Cluster 4:      665
```

Figure 1: Result of K-Means algorithm



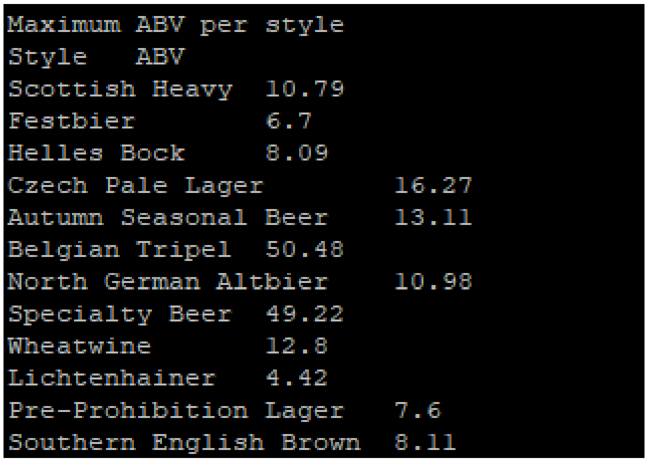
```
Compute Cost: 1.1629669661727939E9
Time in millis: 24794
Time in seconds:        24
Cluster 0:      48577
Cluster 1:      531
Cluster 2:      507
Cluster 3:      1427
Cluster 4:      303
```

Figure 2: Result of Bisecting K-Means algorithm

4.6 Example

As an example we would like to show the first query of our workload, here you can see the *Java* code written in file *Query1.java* and its correspondent result.

```java
package spark;

import scala.Tuple2;

import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.sql.SparkSession;

import java.util.Arrays;
import java.util.List;
import java.util.regex.Pattern;

public final class Query1 {
    public static final Pattern COMMA = Pattern.compile(",");
    public static final String DATASET =
        "hdfs://master:9000/user/user10/input/beers.csv";

    //FOR EACH STYLE SELECT THE GREATEST ABV
    public static void main(String[] args) throws Exception {
        SparkSession ss = SparkSession
                .builder()
                .appName("Q1")
                .getOrCreate();

        JavaRDD<String> lines = ss.read().textFile(DATASET).javaRDD();
        JavaPairRDD<String, Double> abvs_per_beer = lines
                .mapToPair(b ->
                    new Tuple2<>(COMMA.split(b)[2],
                    Double.valueOf(COMMA.split(b)[6])));
        JavaPairRDD<String, Double> max_per_style = abvs_per_beer
                .reduceByKey((abv1, abv2) -> abv1 >= abv2 ? abv1 : abv2);

        List<Tuple2<String, Double>> result = max_per_style.collect();
        System.out.println("Maximum ABV per style");
        System.out.println("Style\tABV");
        for (Tuple2<?,?> tuple : result)
            System.out.println(tuple._1() + "\t" + tuple._2());

        ss.stop();
    }
}
```



```
Maximum ABV per style
Style   ABV
Scottish Heavy  10.79
Festbier        6.7
Helles Bock     8.09
Czech Pale Lager        16.27
Autumn Seasonal Beer    13.11
Belgian Tripel  50.48
North German Altbier    10.98
Specialty Beer  49.22
Wheatwine       12.8
Lichtenhainer   4.42
Pre-Prohibition Lager   7.6
Southern English Brown  8.11
```