Research Project Report: Spark, BlinkDB and Sampling

Qiming Shao
qiming_shao@brown.edu
Department of Computer Science, Brown University

May 15, 2016

ABSTRACT

During the 2015-2016 academic year, I conducted research about Spark, BlinkDB and various sampling techniques. This research helped the team have a better understanding of the capabilities and properties of Spark, BlinkDB system and different sampling technologies. Additionally, I benchmarked and implemented various Machine learning and sampling methods on the top of both Spark and BlinkDB. There are two parts from my work: Part I (Fall 2015) Research on Spark and Part II (Spring 2016) Probability and Sampling Techniques and Systems

CONTENTS

1

RESEARCH ON SPARK

For Part I, I designed and prototyped Spark backend for the Vizdom interface to help our tool to run on Spark. During this semester, I mainly explored Spark in order to help us to get a better understanding on Spark and tried different methods, structures and libraries to reach our goal on running Spark. I explored the traditional RDD method, data frame method, and compared the performance of both strategies. I also examined various related libraries to try to find some useful methods for our project. I will talk about methods that I explored and problems that I encountered in the report. Hopefully, this will give us a better sense on Spark.

1.1 SAMPLE USE CASE

The sample case that I will use in this report is a case to analyze mimic2 data set. We will use this data set and predict the metabolic attribute which is a value which can be only 0 or 1 by using age, heart_rate, height, weight attributes. Also, we should filter patient information to get all data with age greater or equal to 20.

1.2 RDDS METHOD AND SPARK MLLIB (SPARK.MLLIB PACKAGE)

At the beginning, I tried the general way to do data processing and machine learning algorithm using RDDs as the data structure and spark.mllib as the Machine Learning libraries to analyze data. RDDs (Resilient Distributed Datasets) are a fault-tolerant collection of elements that can be operated on in parallel. Spark MLlib is Sparks scalable machine learning library consisting of common learning algorithms and utilities. Spark Machine Learning Library(MLlib) consists two packages: 1. Spark.mllib contains the API built on top of RDDs 2. Spark.ml provides higher-level API built on top of DataFrame for constructing ML pipelines. In the general way, I mainly use RDD and Spark.mllib which is built on top of RDDs to analyze our data.

In order to predict metabolic attribute, I wrote code to build a machine learning model on couple of selected attributes in the data set. First of all, I loaded data from csv file to a RDD. Then I removed the head line of the data (the head line of data is column name line). After that, I parsed the data by only selecting needed columns and

built an array to store selected attributes. Then I used a mapper to convert every data array to a LabelPoint with their label. Labeled point is a local vector associated with a label/response and is used as the input for supervised learning algorithms. Then by importing LogisticRegressionWithSGD library, we can feed our processed data into this model to get predictions. In the end, I used the same data set as test set to test the training error rate. The code [11] is a simple code sample for this general method using RDD and Spark.mllib.

1.3 SPARK DATAFRAME AND SPARK ML (SPARK.ML PACKAGE)

In spark, a DataFrame is a distributed collection of data organized into named columns. The reasons to explore on DataFrame are [2]:

- DataFrame provides a domain-specific language for distributed data manipulation which can make the code simpler and more specific. DataFrames provide some built-in method for data manipulations such as filter, select, groupBy and so on
- DataFrames can incorporate SQL using Spark SQL
- DataFrames also can be constructed from a wide array of sources
- DataFrame(and Spark SQL) has some built in query optimization (optimized using the catalyst engine) which means using DataFrames to process data will be faster than using RDDs and I will talk about that in next section (Comparison between DataFrames and RDDs)

There are two parts of code that I wrote for the sample case. One is to process data using DataFrame and another one is to use Spark.ml machine learning package to do the training and testing. First, I loaded the data from the input file to a data frame using sqlContext. Since we have column names in the data file, I used a package called "com.databricks:spark-csv_2.10:1.2.0" which can recognize column names in data files and assign column names to appropriate data column in DataFrame automatically. Through using built-in function select, I stored data with needed columns in the DataFrame. Since during loading step, all data loaded have String type, I processed data by assigning appropriate type to each column after selection. Then by using built-in filter function, I filtered with all people has an age greater than or equal to 20 for our data set. Then I continued to the next step, building up machine learning model to train and test data. At here, since DataFrame ca not work directly with Spark MLlib(Spark.mllib), I used Spark.ml library which is on top of DataFrame to do the machine learning part. For working on the sample case mentioned on section 1, I used the LogisticRegression in Spark.ml package. However, the LogisticRegression model in

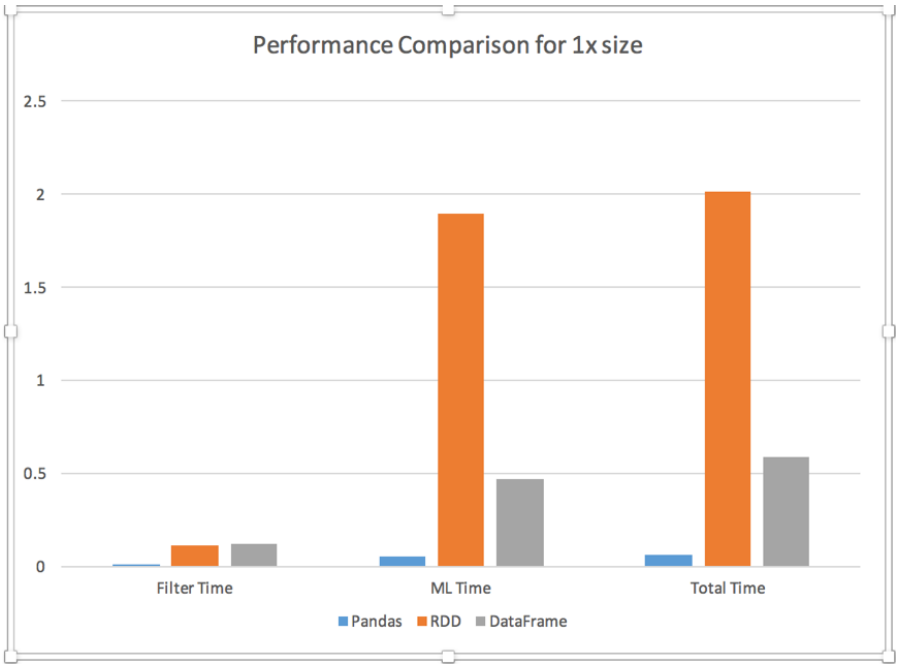
Spark.ml can not take the DataFrame we have now as input. Since DataFrame we have now is a table with multiple columns, but LogisticRegression model in Spark.ml needs a table with only two columns as input. One column is the label and another column is an array to store all needed attributes for prediction. Therefore, we need to find a way to modify DataFrame to get the format that satisfies the Spark.ml input format. Fortunately, in Pyspark DataFrame, there is a method called VectorAssembler which can combine multiple columns in DataFrame to a single vector column. This method can be used to combine columns to generate an aggregated features column for Spark.ml package. Also, I used a StringIndexer to map labels into an indexed column of labels for input DataFrame. After that, I just feed data to the model and get prediction result from Spark.ml LogisticRegression model. The code of this section is located at [12].

1.4 COMPARISON BETWEEN RDDS, DATAFRAMES, AND PANDAS

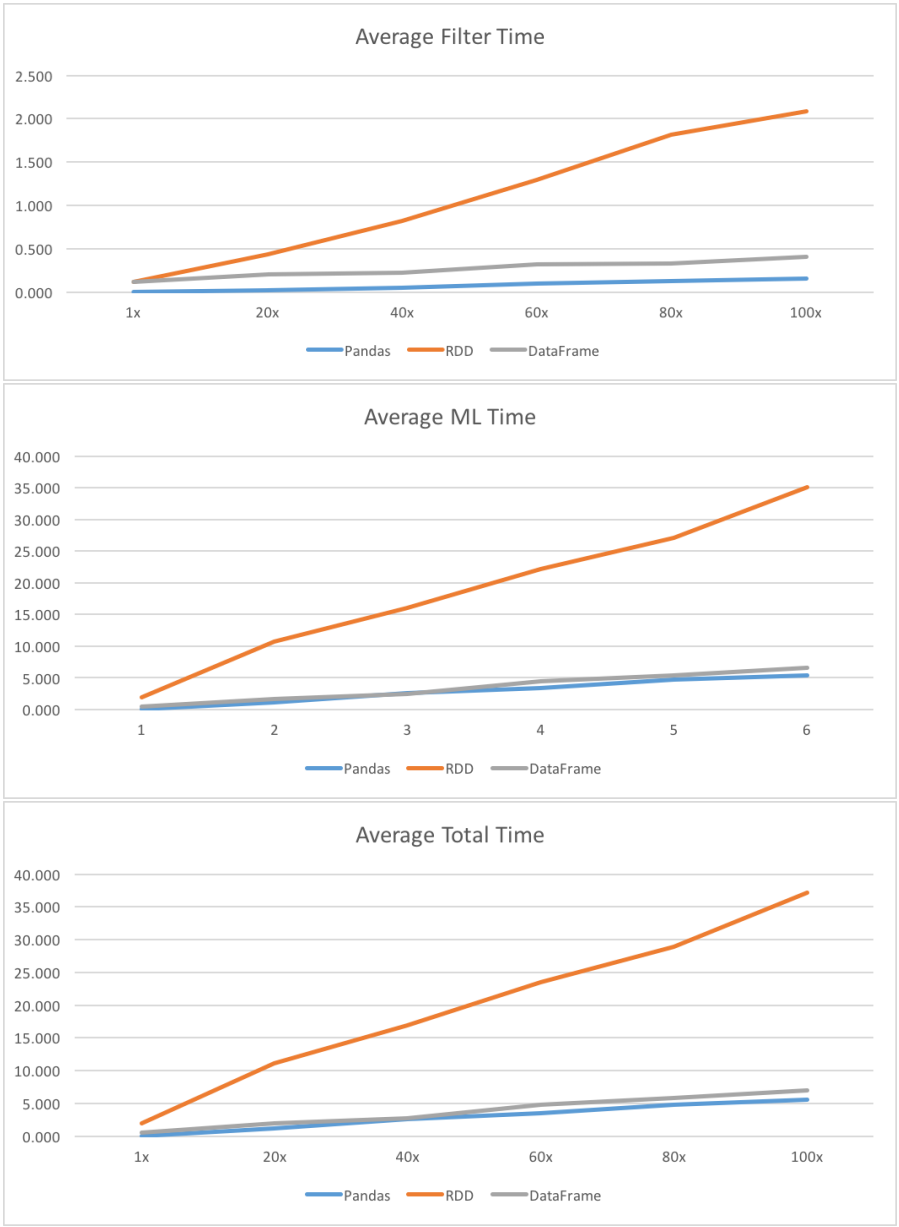
As we can see, compared to the RDD method, DataFrame is much easier to get what we want when coding. However, we still need to determine whether the performance of DataFrame method is better than RDDs method.

In order to measure the performance of both methods, I added timing code in both of methods code to count time for some specific operations. In order to get a good comparison, I used the exact same data set (mimic2) for each methods and setup all parameters and iterations for Machine learning algorithm to the same for methods. I measured both filter operations as well as machine learning operations. Also, I would count the total time to perform the task for both methods. Additionally, I benchmarked Python Pandas and compared them to the RDD and DataFrame methods. Since python uses lazy computing, it wont load the data and run the code unless there are some output operation, such as print. Therefore, in order to eliminate the time influence on loading data and running unrelated code, I set up a warm up step for time counting. In the warm up step, I repeated the code for processing data and print output. After the warm up step, I would rerun the code and use timer to count each steps time. In this way, since warm up step has output part, the code will be executed and the data will be loaded before the real time count code part starts. Also, I added persist method (in RDD and DataFrame methods) to the data after loading to keep the data in memory when running the code. I added timer before and after operations (filter operation and machine learning prediction operation) to measure performance.

Below are some results for the performance of all methods based on the mimic2 data set (we define the size of mimic2 data set as 1x size which is about 9.8MB) and the same filter criteria to the data.



From the chart above, we can see how time distributes in the entire process. Machine learning time composes almost all of time in the entire process for all three methods. And apparently, compared to Spark MLlib library with RDD, using Spark ML library with DataFrame can reduce the time significantly. The filter time is just a very small fraction in the entire process. For filter time count, we can see that DataFrame takes a little bit more than filter in RDD, but compare to the time reducing in Machine learning part, DataFrame is still better than RDD in total time spending for filtering and machine learning. However, The Python pandas method uses less time than DataFrames for all sections. In order to research on how those methods perform for a big data set, I expand the input data set from 1x to 100x (which is about 900 MB) to see the time count changing in those three methods. The following three graphs are the comparison between Python pandas method, RDD method and DataFrame method in average Filter time count, average Machine Learning time count and average Total time count with different input size. In the follow graph, I used different input data size (1x means the size of mimic2 data set and 20x means 20 times the size of mimic2). For the following three graphs, the x-axis is the size of the input data set and y-axis is the time in seconds.



From graphs, we can see all RDDs performance time is more than DataFrames and Python Pandas after 1x size. And Python Pandas have better performance than DataFrame, but with the increasing of input size, the time that Python pandas spends increases faster than the time of DataFrame.

1.5 PROBLEMS

1.5.1 *Machine Learning Algorithm in DataFrame*

According to our time counting and materials on the internet, DataFrames are better than RDDs, so we want to use spark.ml library with DataFrame as input for our Machine Learning algorithms. However, I found Spark.mllib package can work on most of machine learning algo-

rithms, but Spark.ml cant. For example, I wanted to implement a frequent itemsets algorithm by using DataFrame and Spark.ml. But Spark.ml could not support that. After researching online, I found there might be couple of reasons[3].

- DataFrames do not fit for most machine learning algorithms. Most of Machine Learning algorithm require efficient linear algebra library not a tabular processing like DataFrames
- DataFrames gives very little control on the data. We cant use partitioner and there is no control on partitioning for users. So there might be some problems when iterating in the Spark without control over partitions
- Catalyst(Spark.ml used for optimization) applies local optimizations which is good for SQL query or DSL expression. However, typical scalable algorithms usually require iterative processing. So DataFrames may not be faster than RDDs for some operations

1.5.2 *Saving a Spark DataFrame*

I also tried to help figure out a way to save a Spark Dataframe into a thread-safe dictionary. But I didnt find a good way to save it. It looks like that there is no way to serialize a DataFrame in Spark. However, serializing custom data type and RDD works.

1.6 CONCLUSION

From my work, both DataFrame and RDD can implement some algorithms to analyze our data. With some optimizations, DataFrame is more efficient than RDD. However, compare to RDD, the Machine learning algorithm that can use DataFrames are limited.

2

PROBABILITY AND SAMPLING TECHNIQUES AND SYSTEMS

For Part II (Spring 2016), I worked on implementing different kind of probability sampling techniques on very large data systems. I experimented and researched on two systems, Spark and BlinkDB, to check whether they are a good fit and how they perform for probability sampling on very large data set. I will talk about the theory of different kind of probability sampling techniques first and then talking about the implement of sampling methods on Spark and BlinkDB with their performance.

2.1 THEORY

A sample is a subset of a population. In this section, I will introduce the most popular sampling techniques and mainly focus on the introduction of stratified sampling techniques. There are two main sampling methods: **probability sampling** and **nonprobability sampling**. I will mainly introduce probability sampling in this report.

Probability sampling is a sampling method that collects sample in a process that gives each member of the population the same chance to be selected.[6] There are many types of probability sampling, listed below.

2.1.1 *Simple Random Sampling*

Simple random sampling randomly selects a given number of objects from the entire population. Simple random sampling needs to guarantee each member of the population have to have the equal chance to be selected.

2.1.2 *Stratified Random Sampling*

Stratified sampling is a probability sampling method that divides the population into homogeneous subgroups, called strata and select the sample from each strata separately by applying simple random sampling or systematic sampling. There should be no overlapping data items between any two strata.

Compared to simple random sampling, Stratified sampling has some advantages. Stratified sampling makes sure every strata will be represented in the sample of the entire population. Stratified sampling method is possible to increase the precision with the same sample size or reduce the sample size with a given precision. Stratified sampling method works well for population with variety attributes. There are two different kind of stratified sampling techniques: Proportionate Stratified Sampling and **Disproportionate Stratified Sampling**.

2.1.2.1 *Proportionate Stratified Sampling*

In proportionate stratified sampling, the population of sampling units are divided into subgroups(strata) and the sample is selected separately in each strata. For the sampling to be proportionate, the sampling fraction must be identical for each stratum. There are two methods of proportionate stratified sampling: **Explicit Stratification** and **Implicit Stratification**.

2.1.2.1.1 Explicit Stratification[14]

Explicit stratification separates the population into strata and then selects an independent sample from each strata. It also weights the sample size in order to make the sample proportionate. Using this approach, the sample size in each stratum exactly matches the weight of the stratum in the entire population because it is proportionate to the actual population.[4]

Strata sample sizes are determined by the following equation:

$$n_h = (N_h / N) * n$$

where is the sample size for strata h, is the population size for strata h, N is the total population size, and n is the total sample size.

Based on the equation above, we can know for a strata h, the weight of h in sample is same to the weight of h in the entire population.

Here is an example and sample size is 1000.

Population (N)	Weight (N_h/N)	sample size $(n * Weight)$
1000	0.1	100
2000	0.2	200
7000	0.7	700

Based on weight of the region in the entire population, we can calculate the sample size needed from each region. After getting the sample size for each region, we can perform simple random sampling in each region to get samples.

2.1.2.1.2 Implicit Stratification

Implicit stratified sampling procedure uses a sorted list based on target attribute and then a systematic sample from the sorted list using a fixed sampling interval and a random start.[14] Here is an example:

Region	Size	Cumulative Size	Sample Selection
A	30	30	20
B	58	88	49,78
C	65	153	107,136
D	78	231	165,194,223
E	26	257	252
F	33	290	281

We want 10 samples from the population. The sampling interval = total population / number of samples which is 290 / 10 = 29. And we have a random start at 20. So we can select the sample from the sorted list with starting point 20 and interval 29.

2.1.2.1.3 Explicit Stratification vs. Implicit Stratification[14]

Explicit stratification will miss some strata if the size of those strata are too small in the population (e.g. stratas weight is too small to get at least one element from this strata), but implicit stratification can avoid the problem of small (or fractional is less than 1) sample sizes. Both explicit stratification and implicit stratification yield the same results if sample size weighting is used in explicit stratification. Also, implicit stratification is quicker because we can work from one list and dont have to weigh the sample size. Whats more, implicit stratification is good for continuous variables as well as those with a large number of strata.

2.1.2.2 *Disproportionate Stratified Sampling*

In a disproportionate stratified sampling, the sampling fraction is not the same within all strata. Some strata are over-sampled relative to others. The precision of the disproportionate stratified sampling may be very good or very poor, depending on how sample points are allocated to strata[4]. And this design also considers a lot about the cost of sample selection/survey which can minimize the cost for some given precision. There are some different kind of disproportionate stratified sampling methods and I will introduce them at the following section.

2.1.2.2.1 Uniform Allocation

Uniform Allocation is to select the same number of samples from each stratum, which is an ideal approach if there is no information available about variability of units within the strata, the cost of sampling is similar for all strata, and strata are of similar size.

2.1.2.2.2 Optimum Allocation

Optimum allocation is a method of stratum allocation of a stratified random sample in which strata sampling rates are directly proportional to standard deviation of the distribution of the variable and inversely proportional to square root of the average unit cost of data collection in the strata.[4]

Based on optimal allocation, the best sample size for stratum h would be[4]:

$$n_h = n * [(N_h * \sigma_h)/sqrt(c_h)]/[\sum(N_i * \sigma_i)/sqrt(c_i)]$$

where n_h is the sample size for stratum h, n is total sample size, N_h is the population size for stratum h, h is the standard deviation of stratum h, and ch is the direct cost to sample an individual element from stratum h.

2.1.2.2.3 Neyman Allocation

Neyman allocation combines simple random sampling and proportionate allocation sampling.

There are two steps: 1) Before a random selection is conducted, the population frame are separated into different strata. 2) Use random sampling to sample from each stratum. Obtain the confidence limits. The sample size is weighted by both the population proportions and standard deviation for the stratum.

Based on Neyman allocation, the best sample size for stratum h would be:

$$n_h = n * (N_h * S_h)/[\sum(N_i * S_i)]$$

where n_h is the sample size for stratum h, n is total sample size, N_h is the population size for stratum h, and S_h is the standard deviation of stratum h.

Neyman allocation is to maximize survey precision with a given fixed sample size. However, there are still limitation for Neyman allocation. First, Neyman allocation needs to work on very large percentage differences between strata. Second, the population proportion and variability will tend to vary from variable to variable. The allocation for one variable may not be application for another variable. Third, Allocation done based on standard deviation for that stratum may inflate the standard errors. Forth, Nonresponse, coverage and measurement errors are not taken into account in this allocation approach.

2.1.2.3 *Comparison Between Proportionate Stratification and Disproportionate Stratification*

Compare to proportionate stratification, optimum allocation can be a better choice (less cost, more precision) if sample elements are assigned correctly to strata.[15] Disproportionate stratification can be used to save money or to make reliable estimates for each and every stratum. If the variances are very similar, proportional allocation is a good choice, but for optimal allocation, more sampling effort is allocated to larger and more variable strata, and less to strata that are more cost to sample.[16] Optimal allocation provides the most precision for the least cost, so optimal allocation can max precision by given a fixed budget. Neyman allocation can max the precision by given a fixed sample size.

2.1.3 *Systematic Random Sampling*

Systematic sampling is a probability sampling method that create an ordered sampling list. With a random starting point and a fixed, periodic interval, this method can select the sample randomly. The interval is calculated by dividing the entire population by the desired sample size. The systematic sampling method is quicker than random sampling, but if the list of the population comes to a cyclical pattern that matches the sampling interval, the selected sample may be biased.[5]

2.1.4 *Cluster Random Sampling[6]*

Cluster random sampling is done when simple random sampling is almost impossible because of the size of the population. In cluster sampling, we need to identify boundaries and then select a number

of identified areas randomly. In cluster sampling, all areas within the population have to have equal chances of being selected. Once getting the areas, we can either include all the individuals within the selected areas or random selection from the identified areas.

2.1.5 *Mixed/Multi-Stage Random Sampling*

This sampling method is a combination of two or more sampling methods mentioned above.

2.2 SYSTEM

2.2.1 *Spark(PySpark)*

2.2.1.1 *Simple Random Sampling*

2.2.1.1.1 "sample" method

Spark has a sample() method that samples a fraction of RDD data with or without a replacement and using a given random number generator seed.

sample(withReplacement, fraction, seed=None)

Return a sampled subset of this RDD[10]

Example[10]:

$>>> rdd = sc.parallelize(range(100), 4)$

$>>> 6 <= rdd.sample(False, 0.1, 81).count() \leq 14$

2.2.1.1.2 "takeSample" method

takeSample() method generates an array with a random sample of a given number elements of the dataset, with or without replacement and a random number generator seed.

takeSample(withReplacement, num, seed=None)

Return a fixed-size sampled subset of this RDD[10]

Example[10]:

$>>> rdd = sc.parallelize(range(0, 10))$

$>>> len(rdd.takeSample(False, 5, 2))$

2.2.1.2 *Stratified Sampling*

Spark(pyspark) provides a method called sampleByKey to implement stratified sampling. sampleByKey() can be performed on RDDs of key-value pairs. For the stratified sampling, sampleByKey() can pply to an RDD of key-value pairs with key indicating the strata, and take a map from users that specifies the sampling probability for each strata.

sampleByKey(withReplacement, fractions, seed=None)
Return a subset of this RDD sampled by key. Create a sample of this RDD using variable sampling rates for different keys as specified by fractions, a key to sampling rate map.[10]

Example[10]:

$>>> fractions = a : 0.2, b : 0.1$

$>>> rdd = sc.parallelize(fractions.key()).cartesian(sc.parallelize(range(0, 1000)))$

$>>> sample = dict(rdd.sampleByKey(False, fractions, 2).groupByKey().collect())$

$>>> 100 < len(sample[a]) < 300 and 50 < len(sample[b]) < 150$

$>>> True$

sampleByKey() applies Bernoulli sampling or Poisson sampling to each item(key or stratum) independently, but doesn't guarantee the exact sample size for each stratum. sampleByKey() allows users to sample approximately $[f_k * n_k]$ items, where f_k is the desired fraction for key k, n_k is the number of key-value pairs for key k.

Since sampleByKey() can be used to select samples from each strata(key of RDD) with a given fraction from users, we can implement a stratified sampling by deciding the sample size or fraction for stratas. If we can get the fraction map for stratas and input it to sampleByKey(), Spark will select a stratified sample from the RDD(the entire population). Here are some detail implements (deciding the sample fraction) for stratified sampling techniques.

Proportionate Stratified Sampling: if we want a sample from the entire population with a fraction f, a proportionate stratified sampling can be applied by giving every strata(key) the same fraction f in the fractions map.

Uniform Allocation: if we want a sample size n from the entire population N and we have m stratas, for ith strata, the fraction will be , where is the population of ith strata.

Optimum Allocation or Neyman Allocation: based on the formula for each allocation on Theory section, we can calculated the fraction of the strata in fractions map.

2.2.2 *BlinkDB*

BlinkDB is a large-scale data warehouse system built on Shark and Spark and is designed to be compatible with Apache Hive[8]. Current

version of BlinkDB supports creating/deleting samples on any input table/materialized view and executing approximate HiveQL queries with some aggregates that have statistical closed forms (e.g. AVG, SUM, COUNT, VAR and STDEV).[8] In the current BlinkDB paper, authors talked about the stratified sampling using behind queries. However, after doing some experiments on the current version of BlinkDB, I do not think they are applying stratified sampling on BlinkDB. I will talk about that in later section.

2.2.2.1 *BlinkDB Query Language [8]*

BlinkDB implements a SAMPLEWITH operator that get a sampling ratio from user as a parameter and return a random sample from the given table with the given sampling ratio. The number of sample that we got is an approximate number close to the size for ratio. Also, there is a CREATE TABLE AS SELECT operator that can record the random sample as a new table. Here is an example:

If we want to get a 10% random sample from a table called logs, we can use this query:

$ CREATE TABLE logs_sample AS SELECT * FROM logs SAMPLEWITH 0.1; Also, we can add filtering or grouping for the sample data. Once the sample is created, users need to specific the number of row in the sample that is just created and the number of row in the original table by using blinkdb.sample.size and blinkdb.dataset.size. Here is an example:

$ set blinkdb.sample.size=1000;

$ set blinkdb.dataset.size=10000;

After giving BlinkDB the size of sample and original table, users can run approximate aggregation function (such as APPROX_AVG, APPROX_COUNT, APPROX_SUM) on the sample. Then BlinkDB will return an approximate result of the original table with error bars at 99% confidence. Here are some examples:

$ SELECT APPROX_COUNT(1) FROM sample WHERE col1=male;

$ SELECT col1, APPROX_AVG(col2) FROM sample GROUP BY col1;

$ SELECT col1, APPROX_SUM(col2) FROM sample GROUP BY col1;

Also, BlinkDB provides a BlinkDB Command Line Client(CLI) to execute Hive queries. Here are some examples[8]:

$ ./bin/blinkdb # Start CLI for the interactive session

$ ./bin/blinkdb e SELECT * FROM sample # Run a query and exit

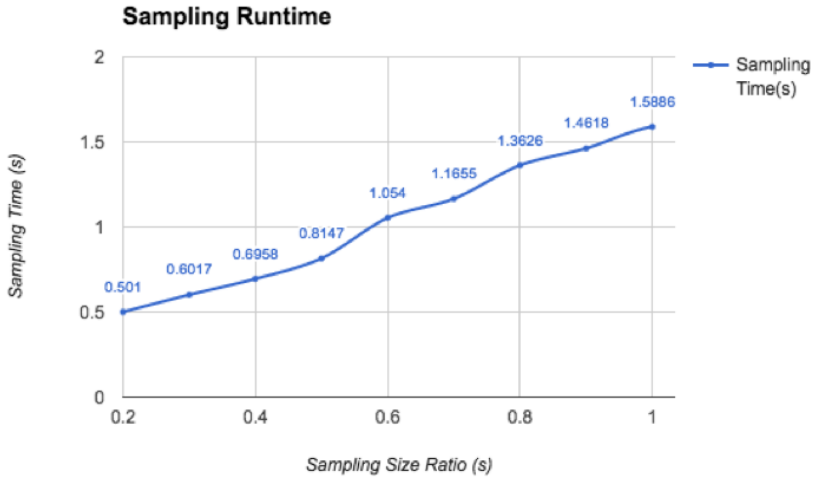
$ ./bin/blinkdb i queries.hql # Run queries from files

$ ./bin/blinkdb H # Start CLI and print help

2.2.2.2 *BlinkDB Query Performance*

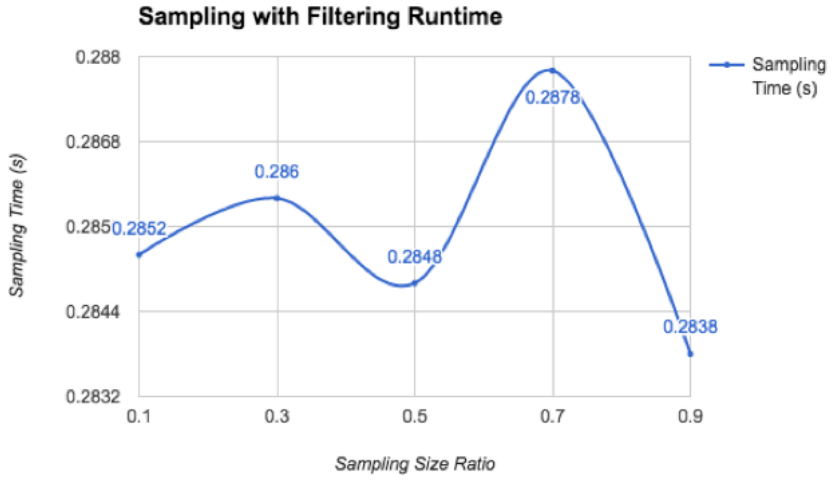
In order to test the runtime of BlinkDB, I used the census dataset. The sample that I used had 32562 rows.

First, I tested the performance for running only samplewith method and used a query "SELECT * FROM census SAMPLEWITH 0.1;". The results are shown below.



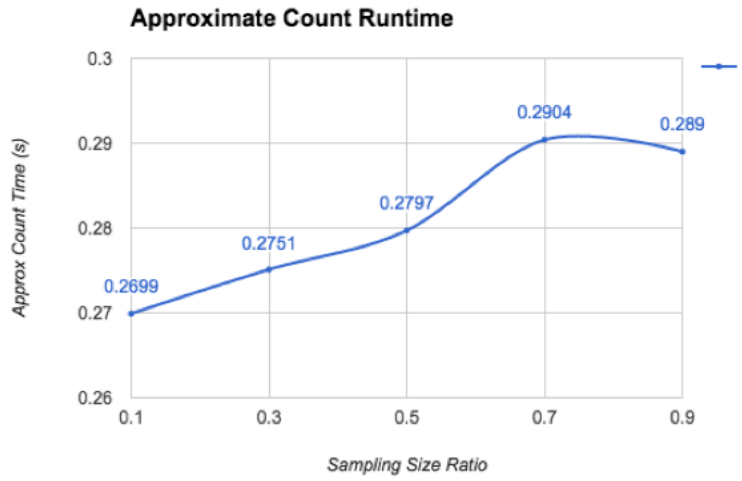
From the graph, we can see the sampling time increase slowly with the increasing of the sampling size ratio.

Second, I tested sampling with with filtering by using the query "SELECT * FROM census WHERE $education\_num \geq 9$ SAMPLEWITH 0.1;". Below are the runtimes.

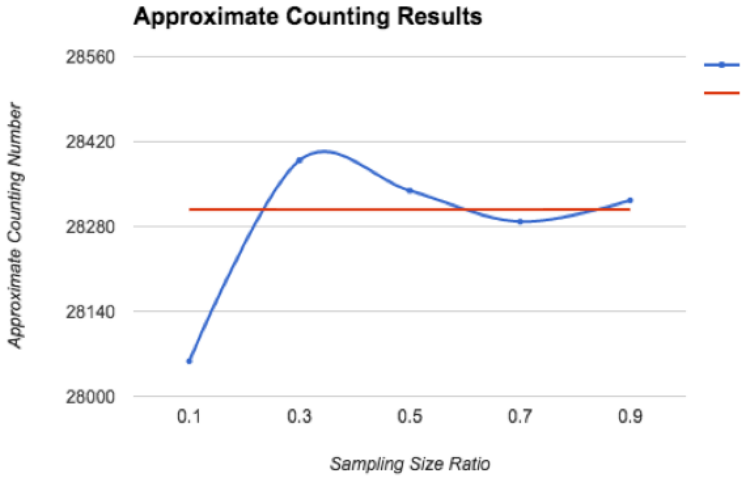


From the graph, with the increase of the ratio, we can see the runtime is within a small range. Then, I set the size of sample and database and tried to test the approximate aggregation queries. I used a filter in all approximate aggregation queries.

Third, I tested the approximate count query by using query "SELECT $APPROX\_COUNT(1)$ FROM sample WHERE $education\_num \geq 9$;"

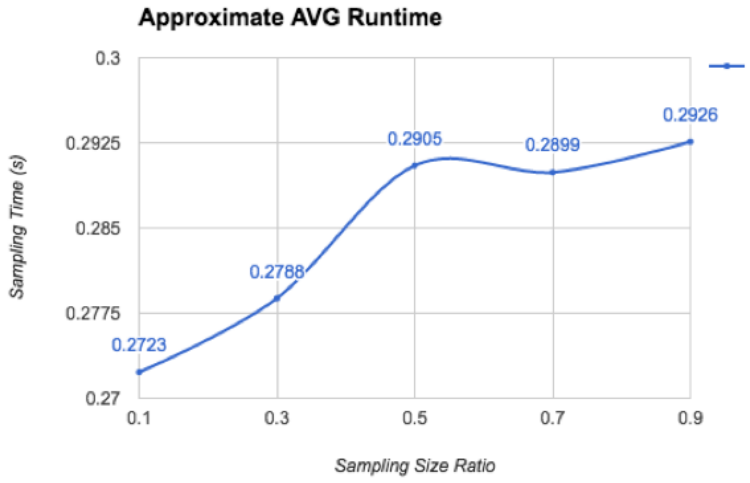


From the graph, we can see with the increasing of sample size ratio, the run time of approximate counting will increase.

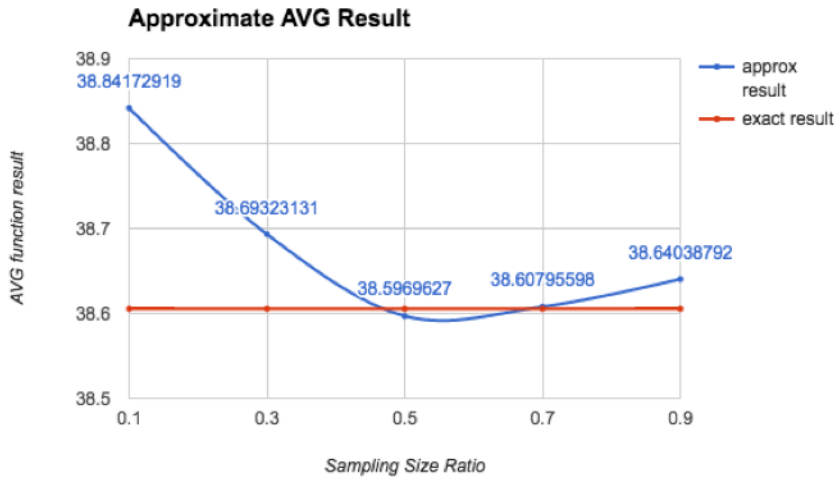


The graph above is the result of approximate counting compared to the current answer. We can see the bigger the sampling size, the better the result that we can get.

Forth, I tested the approximate average aggregation function by using the query "SELECT $APPROX\_AVG(age)$ FROM sample WHERE $education\_number \geq 9$".

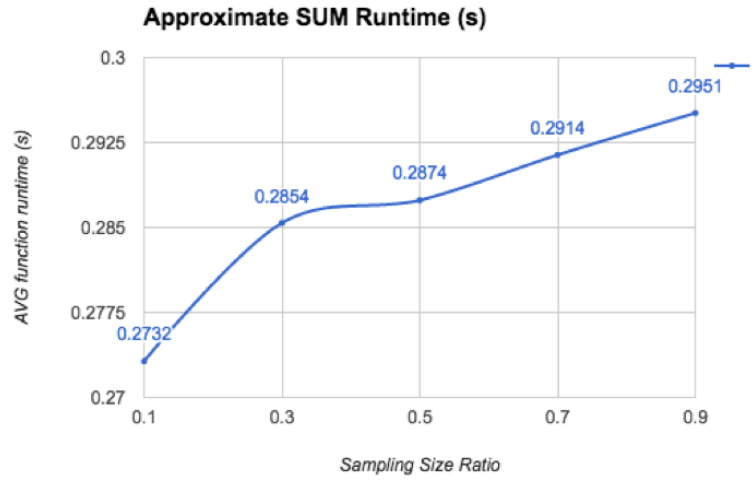


From the graph above, we can see the runtime is increasing when the sample size increases. And the graph below is between approximate result and exact result about the AVG function.



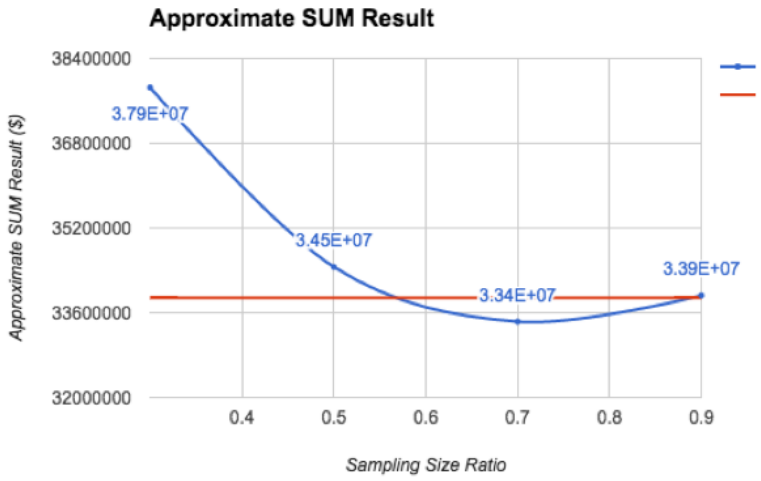
From the graph above, we can see with the increasing of the sampling size, the approximate result is closer to the exact result.

Fifth, I tested the approximate sum function by using the query "SELECT $APPROX\_SUM(capital\_gain)$ FROM sample WHERE $education\_num \geq 9$;"



From graph above, we can see sum runtime has the same pattern as other two. With the increasing of sampling size, the runtime will increase.

And the graph below is for the approximate result of the approximate sum function. We can see with the increasing of the sampling size , the approximate result is approaching to the exact result.



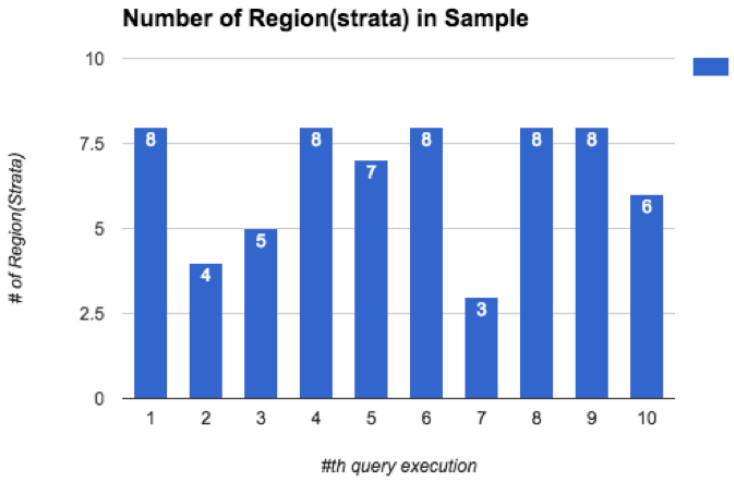
2.2.2.3 *Stratified Sampling in BlinkDB*

The paper talked about BlinkDB mentioned that there are stratified sampling techniques in their designed database system. However, after experimenting some designed dataset on BlinkDB, I do not think they support stratified sampling on current version of BlinkDB.

Here is my designed experiment dataset:

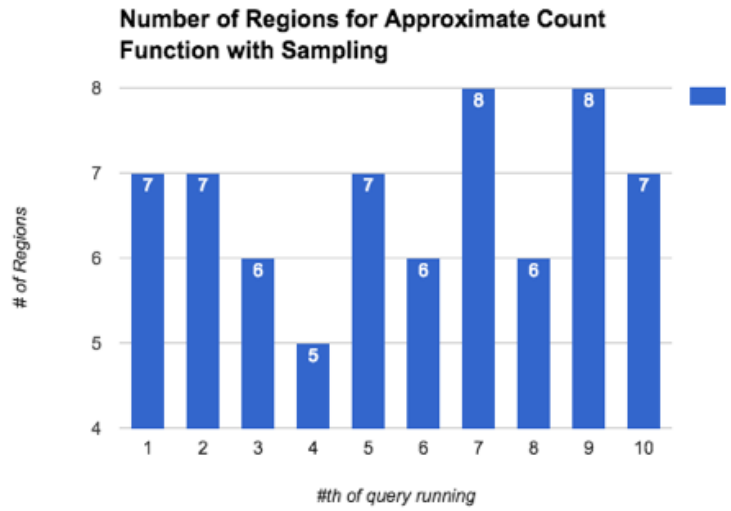
I created a data set with only two columns: Region and Number. Region is designed as the strata for the stratified sampling and the number is just a random number. The total number of regions is

11 and the range of Region is from a to k. In this dataset, every region has 10 rows and the total number of row in the dataset is 110. Since regions or stratas are equal size, if stratified sampling applies, our selected sample should include all or most of stratas. First, I experiment on samplewith method. I executed a query "SELECT * FROM regions WHERE *region* > *a* SAMPLEWITH 0.1;". By running the query, I should get about 10 sample element and about 10 regions (b-k). The graph below is the number of region that I got for 10 times query running.



From the histogram above, we can see all those 10 times query execution got the result match the stratified sampling, because stratified sampling should represent every strata(region) in the sample and every result of query running here has some region missing. Therefore, the samplewith method does not include a stratified sampling techniques and it only applies a simple random sampling.

Then, I experimented on approximate aggregation function for BlinkDB by using the same designed dataset. I choose $APPROX\_COUNT$ to test and the result should be similar with other aggregation functions. The query I used to test is "SELECT region, $APPROX\_COUNT(1)$ FROM sample GROUP BY region;" and for every run, I will created a new 10% sample from regions. The graph below is the number of region we got for each run.

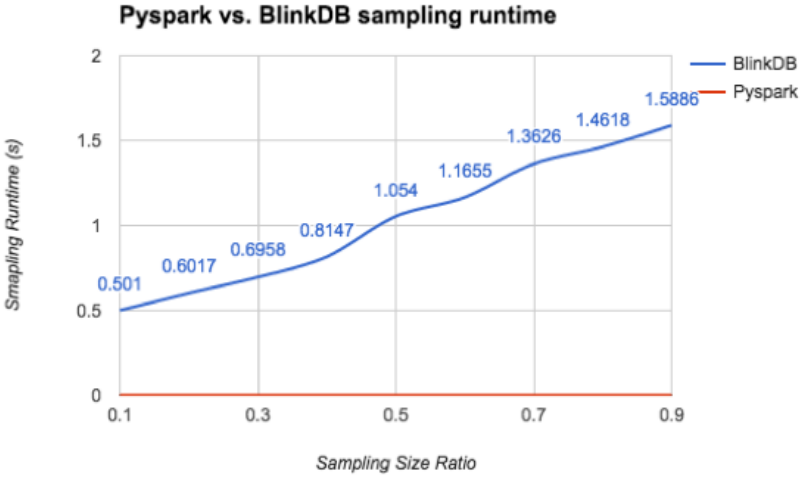


From this graph, we can see the most number of region that we can get is 8, however, all of cases are lower than our expectation, 10 regions. Therefore, with the experiment, the current version of BlinkDB only implements on simple random sampling.

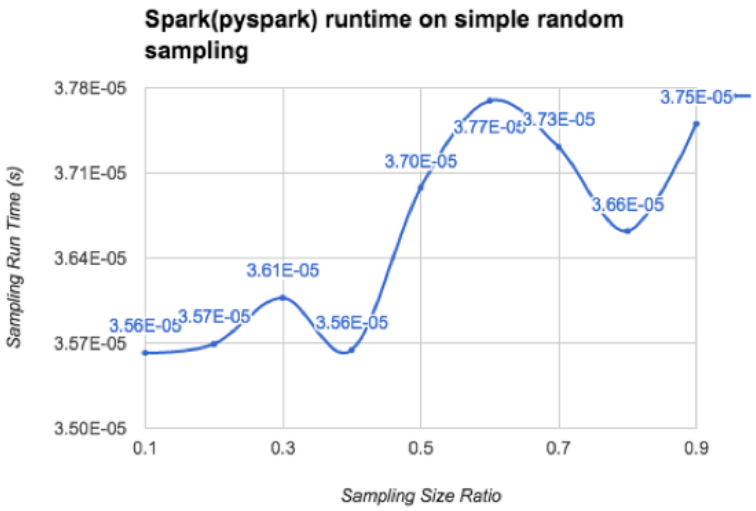
2.2.3 *Comparison between Spark and BlinkDB*

2.2.3.1 *Simple Random Sampling*

In order to compare between Spark and BlinkDB, I used the census dataset to run simple random sampling on both systems. The graph between is the runtime for both:



From the graph, we can see Spark spent much less time for sampling data with all sampling size ratio. In order to see what is the runtime for PySpark, the graph below shows:



2.2.3.2 *Stratified Sampling*

Since current version of BlinkdDB does not support stratified sampling, Spark will be a good choice for implementing the stratified sampling.

2.3 FUTURE WORKS

Although current version of BlinkDB does not support stratified sampling, authors of BlinkDB are currently working on a new version of BlinkDB which uses newest version of Hadoop and Spark. We should keep eyes on this new version and it will have some new features available for BlinkDB.

REFERENCES

[1] http://spark.apache.org/docs/latest/mllib-guide.html
[2] https://databricks.com/blog/2015/02/17/introducing-dataframes-in-spark-for-large-scale-data-science.html
[3] http://stackoverflow.com/questions/33237096/why-spark-ml-dont-implement-any-of-spark-mllib-algorithms
[4] http://stattrek.com/survey-research/stratified-sampling.aspx
[5] http://www.investopedia.com/terms/s/systematic-sampling.asp
[6] https://explorable.com/probability-sampling
[7] http://spark.apache.org/docs/latest/api/python/
[8] https://github.com/sameeragarwal/blinkdb/wiki
[9] http://www.cs.berkeley.edu/ sameerag/blinkdb_eurosys13.pdf
[10] http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD
[11]https://www.dropbox.com/s/icz5qvblnbqguhz/append1.py?dl=0
[12]https://www.dropbox.com/s/7txxjnl8u8vpnzt/append2.py?dl=0
[13] http://www.restore.ac.uk/PEAS/sratheory.php#whatitis2
[14] https://sites.google.com/site/drhuiliew/sampling-and-weighting/implicit-or-explicit-stratification
[15] http://stattrek.com/sample-size/stratified-sample.aspx
[16] http://ag.arizona.edu/classes/rnr321/Ch4.pdf

ACKNOWLEDGEMENTS

Special thank you for the advising from Dr. Tim Kraska, Alex Galakatos and Andrew Crotty