Approximate Query Processing in Spark

Accuracy tradeoffs and performance gains.

Francisco Javier Salido Magos
MS in Data Science
University of Washington
Seattle WA, USA
javiers@uw.edu

ABSTRACT

In this report we analyze the use of Approximate Query Processing (AQP) on GROUP BY queries that compute aggregates on dataset attributes. We employ two Core Spark API sampling functions, `rdd.sample()` and `rdd.sampleByKey()`, and two equivalent Spark SQL functions `df.sample()` and `df.sampleBy()`, to extract smaller, representative samples of various sizes from a dataset of commercial flights in the USA. We run a set of preselected queries on the full dataset and on the samples and compare the accuracy of the results along with the time it takes to process the queries.

We conclude that, for this type of query, performance gains can be significant, up to two orders of magnitude when running on RDDs, and one order of magnitude for DataFrames and Spark SQL, with acceptable levels of accuracy. We further conclude that DataFrames and Spark SQL are actually an order of magnitude faster than the Core Spark API in query execution, albeit with a possibly larger error in some cases. We explain some of the particulars of implementing this approach on Amazon Web Services (AWS), provide and discuss our results. While part of the original scope of this project, we were not able to run the queries on Azure Databricks, having been unable to sort out its security features on time.

KEYWORDS

Approximate Query Processing, AQP, OLAP, Spark, uniform sampling, stratified sampling

1 Introduction

The past two decades have seen an explosion in the size of datasets employed by organizations to meet all sorts of business, government and research challenges. While technology has played a significant role as enabler of this growth, we are just beginning to understand that, for some scenarios, big data datasets may indeed constitute too much of a good thing.

One of these scenarios surfaces in the use of Online Analytics Processing (OLAP), where users may need quick and approximate answers to specific queries that can help guide decision making or market research, and don't have the hours or days it would take to get 100% accurate answers. OLAP professionals may be able to leverage Approximate Query Processing (AQP) in this type of scenario, by randomly selecting representative samples of very large datasets and making them available for analysts to query.

For our analysis we employ a dataset from the Bureau of Transportation Statistics, an office of the U.S. Department of Transportation, in which each record contains the details of one of approximately 61.5 million commercial flights scheduled in the U.S. between 2009 and 2018 [1].

The overall approach is quite simple; we compute the four selected queries on the entire dataset, all 61.5 million flights, employing Resilient Distributed Datasets (RDD) and the Core Spark API, and establish the correct responses and runtime on the full dataset. We then proceed to collect sets of rows selected uniformly at random and without replacement, to create samples of various sizes. We collect a second set of random samples of the same sizes, but this time employ uniform distributions, also without replacement, over the flights flown by each individual carrier. Next, we run the queries on each of the random samples. We compare the resulting runtimes and results, using a metric proposed in [3], to those from the full dataset. We repeat the above flow by running two of the four queries on DataFrames and using Spark SQL functions. All tests were carried out on Spark/Hadoop clusters with 2, 4 and 8 instances, running on the AWS cloud.

Section 2 of this report will discuss our process in detail, Section 3 will focus on our results and in Section 4 we present conclusions and directions for future work.

2 Evaluated System

Hadoop is a data storage and processing platform with the key distinction that it applies the data locality principle. That is, the computation must take place where the data is located, as opposed to having the data travel to where processing will take place. There are two main components to Hadoop, the Hadoop Distributed File

DATA 516, December 2019, Seattle, WA USA
 978-1-4503-0000-0/18/06...$15.00
https://doi.org/10.1145/1234567890

System (HDFS), and a resource management and scheduling system called Yet Another Resource Negotiator (YARN).

Hadoop was built originally with a MapReduce programming model that seemed unsuited for real-time or low-latency applications. As a result, a project led by Matei Zaharia at U.C. Berkeley developed Spark, a cluster computing framework that incorporates data parallelism and fault tolerance in its design. [4]

Foundational components to Spark were Resilient Distributed Datasets (RDDs), read-only data structures that are distributed throughout the instances of a cluster and operate in fault-tolerant fashion using Hadoop's HDFS and YARN. There are two types of operations that can be executed on RDDs; transformations and actions. Transformations create new RDDs from previously existing ones, and actions are operations meant to generate an output. Execution of transformations is lazy, which in practical terms means that these are operations that will not actually be materialized until an operation of the action type is requested, at which point the query plan will be drawn up and executed by Spark.

Over time, a number of abstractions, like DataFrames and datasets, and APIs like the Spark Core API, PySpark, Spark SQL, Spark Streaming, the MLlib and SparkR were developed for use with Spark. Our analysis employs Spark Core API and Spark SQL.

Figure 1 shows the components and data flow of a Spark standalone application.

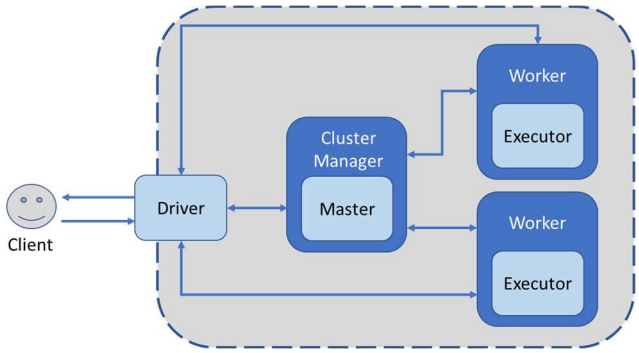

Figure 1. Components of a Spark standalone application. A client submits the application to the Spark cluster through the Driver. The Driver interacts with the Cluster Master, an instance running in the Cluster Manager, and the Executors that run in the Worker instances, of which there can be two or more. Executors run the application and communicate results through the Master, back to the client. [5]

The Driver creates the SparkSession/SparkContext, and plans the execution of the application by creating a Directed Acyclic Graph (DAG), a graphical representation of the transformations and actions that are to be executed as part of the application, and the order in which they are to be executed. The Driver also keeps track of available resources and returns results to the client.

Spark Executors will run the tasks mapped in the DAG and are terminated once the application completes execution. Finally, the Master is the process that requests resources available within the cluster and makes them available to the Driver.

3 Problem Statement and Method

We wish to establish whether the use of randomly chosen samples of a large dataset can be used to obtain reasonably accurate answers to GROUP BY queries, with aggregations, in significantly shorter periods of time. We would like to know how big those samples need to be in order to provide an adequate level of accuracy, and how much faster we can expect those answers to arrive. We also wish to establish if there is a significant difference between using the Core Spark API or Spark SQL functions in this context.

As indicated before, we employed a dataset where each of the 61.5 million rows represents a single flight by a major U.S. carrier between 2009 and 2018. Most flights took place as planned, with some been delayed or even cancelled. Schema for the dataset is:

```
rdd((year, int), (month, int), (day, int),
(carrier, str), (flightNum, int),
(origin, str), (dest, str),
(depTime, float), (arrTime, float),
(cancelled, bool), (diverted, bool),
(actualTime, float), (airTime, float),
(distance, float), (delay, float))
```

The full dataset was uploaded using GZIP compression to a S3 bucket in AWS, taking advantage of the fact that Spark has native support for various lossless compression formats, by decompressing the data when uploading to memory.

3.1 Core Spark API and RDDs

The first step we took was to identify the sampling functions that are available through the different Spark programming tools. There are Core Spark API functions for sampling RDDs, Spark SQL functions for sampling DataFrames and also functions in SparkR. The Core Spark API functions `rdd.sample()` for uniform random sampling, and `rdd.sampleByKey()` for stratified sampling were chosen for the first set of tests. For Spark SQL, running on DataFrames, we chose the equivalent functions `df.sample()` and `df.sampleBy()`. SparkR was left for future work.

A general overview of the state of AQP was provided by [2], after which we settled on four queries to be computed on the dataset, all of them GROUP BY queries that compute aggregate metrics on some attribute. Two of the queries group flights by carrier, the attribute used for sampling, and the other two group by the attributes year and month. The RDD code for all four is shown below, where the full dataset RDD is designated "`rdd`."

```
rdd1 = rdd.map(lambda x:(str(x[0]), 1))\
.reduceByKey(lambda x, y:(x + y), 100)
```

Query 1: Compute the total number of flights for each of the 23 carriers, over the whole period from 2009 to 2018.

```
rdd1 = rdd.map(lambda x:(str(x[0]),\
float(x[4]))).mapValues(lambda x:(x, 1))\
.reduceByKey(lambda x, y:(x[0]+y[0],\
x[1]+y[1]),\ 100).map(lambda x:(x[0],\
x[1][0]/x[1][1]))
```

Query 2: Compute the average flight delay for each of the 23 carriers, over the full period of time.

```
rdd1 = rdd.map(lambda x: (int(x[1]), 0 if\
x[3] == 'False' else 1))\
.reduceByKey(lambda x, y: (x + y), 100)
```

Query 3: Compute the total number of flight cancellations for each year over the full period of time.

```
rdd1 = rdd.map(lambda x: (int(x[2]), 0 if\
x[3] == 'False' else 1))\
.reduceByKey(lambda x, y: (x + y), 100)
```

Query 4: Compute the total number of flight cancellations for each month over the full period of time.

Here we note the use of the `rdd.reduceByKey()` function to group by carrier because, unlike the `rdd.groupByKey()` Core Spark API function, the former combines records locally in each worker node before shuffling, which should improve query performance.

To extract the desired uniform and stratified random samples we used the following functions:

```
rddU = rdd.sample(False, 0.1, seed=None)
```

Uniform random sample of 10% the size of the full dataset, sampled with no replacement.

```
fx = 0.1
fractions = {'F9': fx, '9E': fx, 'CO': fx,\
'OH': fx, 'EV': fx, 'B6': fx, 'XE': fx,\
'UA': fx, 'NK': fx, 'WN': fx, 'YV': fx,\
'DL': fx, 'VX':  fx, 'NW': fx, 'AS': fx,\
'MQ': fx, 'FL': fx, 'US': fx, 'HA': fx,\
'AA': fx, 'YX': fx, 'G4': fx, 'OO': fx}

rddS = rdd.map(lambda x: (str(x[0]),\
(int(x[1]),\ int(x[2]), str(x[3]),\
float(x[4]))))

rddS = rddS.sampleByKey(False, fractions,\
seed = 3)
```

Stratified random sample function in the Core Spark API. Rows in each bucket/carrier are selected through Bernoulli trials with success probability 0.1. We use the two-character IATA code for each carrier. The `map()` function maps tuples to the format required by `rdd.sampleByKey()`, which is (key, [list of attributes]).

Random samples were of sizes 0.1, 0.01, 0.001 and 0.00001 of the full dataset.

Spark transformations are not executed until an action is requested. Thus, left on its own, queries reported execution times that included the time it took for Spark to read and decompress data from S3, or the time it took to extract the relevant sample. We fixed this by executing the first query after reading the full dataset, or after taking a sample, twice. The resulting RDD was cached with the first execution of the first query and would remain in memory for subsequent query execution.

After running the queries, we compared runtimes and results using the error metric originally proposed by Acharya et al. in [3]:

$$\epsilon = \frac{1}{n}\sum_{i=1}^{n} \epsilon_i \qquad (1)$$

Where n denotes the number of groups in the query, 23 carriers in this case, ϵ denotes the total error in the query result from the sample, as compared to that from the same query on the full dataset, and ϵ_i represents the error in the computation for group i, which is in turn computed by:

$$\epsilon_i = \left(\frac{|c_i - c_i'|}{c_i}\right) 100 \qquad (2)$$

Here, c_i is the actual result from computing the query on group i, and c_i' is the result of computing the query on the sample group i.

This process was executed on AWS Spark/Hadoop clusters with 2, 4 and 8 instances.

3.2 Spark SQL and DataFrames

For this portion of the project we re-wrote the program using Spark SQL functions. The general flow is similar, and we used the same measure of error as we did in the previous section. In this case, we limited our tests to taking samples of 0.01 of the original dataset and ran only the first two queries. The full dataset is represented below by the DataFrame "`df`."

```
total = df.groupBy("carrier").count()
```

Query 1: Compute the total number of flights for each of the 23 carriers, over the whole period from 2009 to 2018.

```
mean = df.groupBy("carrier").mean("delay")
```

Query 2: Compute the average flight delay for each of the 23 carriers, over the full period of time.

The sampling functions are:

```
dfU = df.sample(False, 0.01, seed=None)
```

Uniform random sample of 1% the size of the full dataset and sampled with no replacement.

```
fx = 0.01
fractions = {'F9': fx, '9E': fx, 'CO': fx,\
'OH': fx, 'EV': fx, 'B6': fx, 'XE': fx,\
'UA': fx, 'NK': fx, 'WN': fx, 'YV': fx,\
'DL': fx, 'VX':  fx, 'NW': fx, 'AS': fx,\
'MQ': fx, 'FL': fx, 'US': fx, 'HA': fx,\
'AA': fx, 'YX': fx, 'G4': fx, 'OO': fx}

dfS = df.sampleBy("carrier", fractions,\
seed=None)
```

Stratified random sample in Spark SQL. Elements in each bucket are selected through Bernoulli trials with success probability 0.01. Unlike its Core Spark API equivalent, this function did not require remapping attributes.

We dealt with Spark SQL's lazy execution of transformations in the same way we did for RDDs.

4 Results

4.1 RDD and Core Spark API Results

We first look at the time it took for each of the four queries to execute on the full dataset, and on the uniform and stratified samples, using a sample size of 10% the full dataset, as shown in Table 1.

There is a drop of one to two orders of magnitude depending on the query, between the time it takes to obtain answers on the full dataset, and the time on either type of sample. Nevertheless, we see no real performance improvement when increasing the number of instances in the cluster to 4 or 8 workers.

Cluster Instances	Sample Type	Num Flights by Airline	Avg Delay by Airline	Cancelled Flights/ Year	Cancelled Flights/ Month
2 instances	Full Dataset	76	121	82	77
	Uniform	8.89	13.4	9.47	10.1
	Stratified	8.45	13.7	9.86	9.32
4 instances	Full Dataset	78	124	84	80
	Uniform	9.13	13.7	9.79	9.34
	Stratified	8.38	13.7	9.83	9.37
8 instances	Full Dataset	76	120	82	78
	Uniform	9.02	13.4	9.58	9.14
	Stratified	8.26	13.5	9.69	9.17

Table 1: Time is measured in seconds. We can see a significant reduction from the time it takes to execute any of the queries on the full dataset, to the time it takes to execute on any either type of sample of size 0.1.

Table 2 shows running times for each of the four queries, using different sample sizes on both uniform and stratified random samples. Running time decreases along with the sample size, but the magnitude of the gains in processing time stabilizes quickly reaching what, looking at execution time alone, seems an ideal sample size of 1% that of the full dataset.

Sample Size	Sample Type	Num Flights by Airline	Avg Delay by Airline	Distrib Cancelled Flights Year	Distrib Cancelled Flights Month
Full Dataset		76	121	82	77
0.1	Uniform	8.89	13.4	9.47	10.1
	Stratified	8.45	13.7	9.86	9.32
0.01	Uniform	2.15	2.63	2.25	2.21
	Stratified	2.03	2.62	2.23	2.24
0.001	Uniform	1.46	1.43	1.47	1.56
	Stratified	1.42	1.5	1.57	1.56
0.00001	Uniform	1.38	1.54	1.57	1.53
	Stratified	1.47	1.43	1.47	1.51

Table 2: Times above are measured in seconds. This is a comparison of running times for each query on the full dataset, and on samples of various sizes for both uniform and stratified samples.

We then turn our attention to the quality of the results we got from the queries. Table 3 shows the measure of error, computed using expressions (1) and (2), in the estimation of the number of flights for each carrier, at the different sample sizes, for both uniform and stratified samples.

We can clearly appreciate how total error decreases as the size of the sample grows, with relatively little difference between the results obtained from the uniform and stratified samples.

Carrier	Uniform Samples				Stratified Samples			
	0.00001	0.001	0.01	0.1	0.00001	0.001	0.01	0.1
OH	10.7884	2.4597	1.7778	0.1900	45.6292	2.1071	1.2203	0.8500
XE	15.6228	2.7893	0.8004	0.4188	33.5070	1.2807	0.3491	0.1458
MQ	30.6243	2.7386	0.8199	0.0562	27.3776	0.6813	0.8034	0.0334
VX	64.3232	3.8877	2.2075	0.5606	19.0001	0.1958	1.3656	0.7019
G4	89.8355	0.4303	1.3670	0.5056	100.0000	3.6387	1.4391	0.1208
NK	22.4135	3.7710	5.0232	0.0198	42.7804	3.7863	0.5866	0.7421
US	17.5121	1.6390	0.1466	0.2180	26.5002	1.0762	0.9574	0.3135
WN	1.0929	0.8587	0.1412	0.1412	3.3393	0.3041	0.5739	0.0247
AA	41.2282	1.0809	0.1343	0.1509	11.6146	0.8707	0.2381	0.0119
UA	13.5330	0.1368	0.5753	0.1275	17.2979	0.7900	0.0371	0.0686
YX	42.2122	2.1239	2.8888	0.0317	66.7671	2.8244	3.3766	0.2706
CO	48.0044	2.0203	0.1862	0.2577	56.0445	5.9596	0.3358	0.2060
NW	6.2953	10.2430	0.5443	0.5655	43.7018	7.4045	1.3384	0.2452
YV	48.4763	6.3150	0.3632	0.1217	18.5222	6.3020	0.2682	0.0558
AS	0.9987	1.3291	1.0772	0.2414	17.7618	2.0748	0.6569	0.0883
DL	5.9836	1.5995	0.2111	0.1352	4.8919	0.5012	0.2312	0.0279
EV	0.4815	2.5543	0.2904	0.0434	15.5444	0.3221	0.2569	0.1685
9E	2.7872	2.5645	1.2997	0.1437	11.3329	1.9193	1.1875	0.3717
F9	8.9946	1.3217	0.0686	0.1895	4.6718	8.4936	1.2374	0.0925
B6	0.7261	0.4409	0.9487	0.1211	11.1919	1.0225	0.1699	0.0173
OO	6.4521	1.3492	0.7685	0.1519	4.3977	0.0322	0.3794	0.0670
FL	64.9127	1.2978	2.3782	0.0663	20.7040	0.2513	0.1624	0.3259
HA	1.9255	0.7357	0.8174	0.1811	85.8997	0.1014	1.5987	0.2208
Error	23.7054	2.3342	1.0798	0.2017	29.9338	2.2583	0.8161	0.2248

Table 3: Numbers show the individual group (carrier) error for each type/size of sample, when computing the total number of flights in the dataset for each carrier. The bottom row shows total error for each type/size sample, as computed using equations (1) and (2). Column headers represent the size of each sample as a fraction of the total dataset.

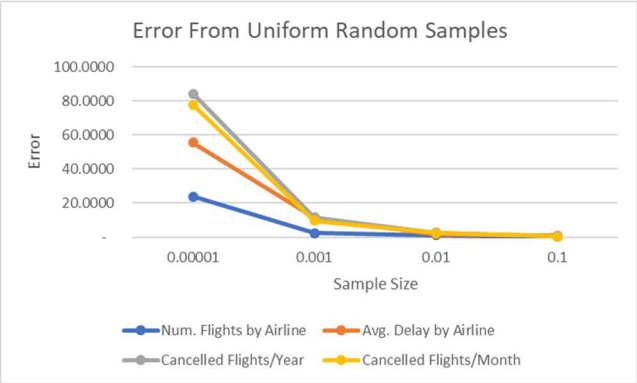

Figure 1: Total error observed for each query by uniform sample size. Sample sizes are fractions of the full data set.

Figures 1 and 2 illustrate how total error decreases as the sample size increases, for all four queries and with both types of samples. Deciding what an acceptable level of accuracy or, in other terms, what an acceptable magnitude for total error should be, is a scenario-dependent decision. Certain error magnitudes may be acceptable in some instances and not in others, or the criteria may be set based not on total error but on a maximum acceptable error for each group/carrier.

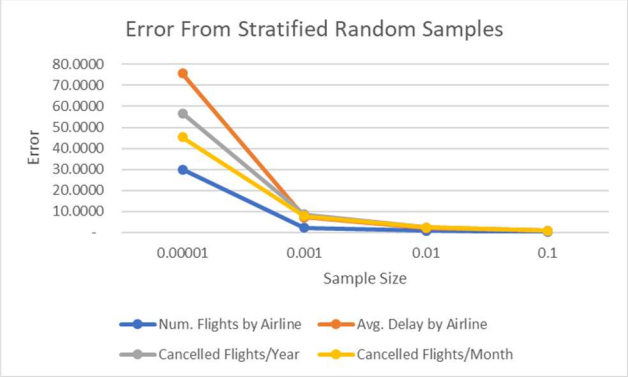

Figure 2: Total error observed for each query by stratified sample size. Sample sizes are fractions of the full data set.

4.2 Spark SQL Results

Given that a sample size of 0.01 of the total dataset seemed to be the sweet spot in our Core Spark API tests, yielding the shortest runtimes and low error measures, we extracted a sample of this size for our Spark SQL tests, and ran queries 1 and 2 on clusters with 2, 4 and 8 workers.

The results surprised us, as runtimes were an order of magnitude less, and the error measurements roughly similar to those obtained from the Core Spark API, for uniformly chosen samples. Runtimes were similar and error larger, but still acceptable, for stratified samples. Once again, there were no relevant differences in runtimes when we increased the number of workers in the setup. The results are summarized and compared to those obtained with the Core Spark API, in table 4 below.

2 instances, 1% sample size		Num. Flights by Airline		Average Delay by Airline	
		Core Spark API	Spark SQL	Core Spark API	Spark SQL
Error	Uniform	1.0798	0.8175	1.5530	3.2217
	Stratified	0.8161	0.9455	2.0941	3.6834
Time	Full dataset	76 s	4.4 s	121 s	5.72 s
	Uniform	8.89 s	702 ms	13.4 s	812 ms
	Stratified	8.45 s	649 ms	13.7 s	666 ms

Table 4: Comparison in runtimes and error measurements for Core Spark API and Spark SQL. Error measurements are close in the first query, but double for the second query. More testing will be required to establish if this is significant. For runtimes, DataFrames and Spark SQL are the faster option.

5 Conclusions

Our results show the significant advantage of using Spark SQL over RDDs and the Core Spark API for this type of task. Spark SQL is a higher-level language, as a declarative language it abstracts much of the complexity of dealing with RDDs and the Spark/Hadoop architecture, and it is clear that the use of DataFrames and query optimization in Spark SQL yield a significant advantage. We have to further analyze and understand Spark SQL's seemingly higher error when running the Average Delay by Airline query.

An interesting takeaway is the fact that runtime improvements and optimal use of resources in Spark require application of the principle of "data minimization." That is, avoid using data that is not essential for query execution and results. Application of this principle is clear in Core Spark API, where we only use attributes that are relevant to the computation. It is reasonable to assume that the query optimizer for Spark SQL takes similar measures.

Dealing with Spark's lazy execution of transformations in order to obtain accurate measurements of query runtimes was another important takeaway.

With respect to the fact that increasing the number of worker nodes had no effect on query runtimes, we concluded that the size of the dataset is probably the reason. Each record from the full dataset is approximately 64 bytes long, and in the Core Spark API version, the RDD rows are further reduced to 20 bytes. Thus, the full dataset is no bigger than 4 GB, which fits in the memory of the worker nodes with plenty of space left available. The basic configuration with two workers is more than enough to cache the full dataset and the samples, and to execute the requested queries. Adding workers will increase communication costs, and likely eliminate any performance gains that might be achieved with an additional 2 or more workers.

Our conclusion is that using AQP is a feasible option to reduce runtimes and maintain an acceptable level of accuracy for decision making, when running GROUP BY queries with aggregation. Given the size of the datasets traditionally used in Spark applications, both methods of sampling, uniform and stratified seem to yield similarly adequate results, unless group size for the smaller groups is a fraction of a percent the size of larger groups. Samples of 1% the size of the dataset seemed to be enough in this case, though it may be possible to use smaller sizes or necessary to use larger sizes, depending on the distribution of the different attributes in the dataset. Using 1% samples, improved response time by as much as an order of magnitude and using DataFrames and Spark SQL would definitely be the right choice.

ACKNOWLEDGMENTS

Thank you very much Dan, Brandon and Deepanshu. It's been great to have the opportunity to learn about big data algorithms and systems from you.

REFERENCES

[1] Reporting on carrier on-time performance. *Bureau of Transportation Statistics*, Department of Transportation. Washington DC, USA. https://www.transtats.bts.gov/DL_SelectFields.asp?Table_ID=236&DB_Short_Name=On-Time

[2] Surajit Chaudhuri, Bolin Ding, Srikanth Kandula, 2017. Approximate Query Processing: No Silver Bullet. *SIGMOD'17 May, 2017*, Chicago, IL, USA. DOI: https://doi.org/10.1145/3035918.3056097.

[3] Swarup Acharya, Phillip B. Gibbons, Viswanath Poosala. 2000. Congressional Samples for Approximate Query Processing of Group-By Queries. *ACM SIGMOD 2000, Dallas, TX, USA.*

[4] Matei Zaharia et al, 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. *USENIX NSDI'12, San Jose, CA, USA.*

[5] Jeffrey Aven, 2018. Data Analytics with Spark Using Python. *Addison Wesley Data & Analytics Series, USA.*