Competitive Assessment of Scala vs. PySpark

Max Moro (mmoro@smu.edu), Nikhil Gupta (guptan@smu.edu)

***Abstract* — In this paper, we present a detailed comparative analysis of Scala and PySpark for distributed computing. While substantial work has been done in this area (with the conclusion that Scala is faster than PySpark), our study aimed at performing a more detailed analysis with blocking for many confounding factors such as data size, setup, and type of query. Ultimately, we wanted to answer two questions – (1) Which language is faster from an end user's perspective (i.e. in a real-world setup) after accounting for these confounding factors and (2) For a data science team that has its applications in Python, would it be a good investment to switch to Scala. Through this analysis, we concluded that Scala was indeed faster than PySpark for several common queries, although the number of machine learning algorithm libraries available in Scala were not as pervasive as those found in PySpark. Hence, we recommend using Scala when the algorithms are available natively and using PySpark if the algorithms are not available in Scala.**

***Index Terms* – Spark, Scala, PySpark, Python**

I. Introduction

As data gets bigger, it can no longer be processed in dedicated machines through vertical scaling, but rather needs to be distributed to more systems (horizontal scaling) using programming languages such as Spark. Spark can be programmed in its native language, Scala, or using Application Programming Interfaces (API's) provided in Python called PySpark [1]. Both methods have their advantages and disadvantages. Scala is claimed to be 10x faster than PySpark [2]. This is due to the fact that Scala is the native language in which Spark is written and hence Spark queries written in Scala are optimized while in order to perform Spark queries in Python, one needs to go through APIs. On the other hand, Python is a more pervasive language compared to Scala. Many more machine learning algorithms are available in Python (compared to Scala) and companies have already built a machine learning infrastructure around the Python programming language.

Through this paper, we wished to perform a detailed comparative analysis between Scala and PySpark to answer two questions – (1) How does the performance of Scala compare with PySpark from an end user perspective after accounting for various confounding factors such as setup, data size and query type, and (2) For a company looking to invest in a big data infrastructure, should they continue using a platform in which they have already made considerable investments (Python), or should they invest in a new platform (Scala) due to its speed and performance advantage.

To answer these questions, we conducted a statistical experiment and took several confounding variables into account using a technique called blocking. 33 different queries were developed across various categories – Row, Column, Aggregate and Mixed operations. All these queries were run across both Scala and PySpark, on 3 different setups (Single Node Local Virtual Machine (VM), Single Node cloud machine, and multi-node cloud cluster). The queries were run from 2 different machines on 5 datasets of sizes varying from 10MB to 500MB. Other confounding variables were accounted for by developing a random data generator to produce the datasets, running the queries multiple times under the same conditions and randomizing the order of the queries in each run.

The results of this study showed that Scala was indeed faster than PySpark under real world setups and had less memory requirements. The barrier to entry for Scala is relatively small since the syntax differences between Scala and PySpark are minimal and many queries could be run in both languages using the exact same syntax. Hence, Scala should be recommended for use in a data science setting provided the machine learning algorithm is available in the Scala framework. In cases where the machine learning algorithm is not available in Scala, PySpark can be used as an alternate solution for data processing and analysis and Python can be used for running the machine learning algorithm.

II. Literature Review

A. Spark

Apache Spark is a unified engine for distributed data processing as defined by the developer group at the University of California, Berkley [3]. Spark works by using a master (Driver) node which interfaces with slave (Worker) nodes. Worker nodes run as 'Executors' of the code received from the Driver node [4].

The most interesting element of Spark is its ability to manage large data by using Resilient Distributed Datasets (RDD). This is a collection of datasets which can be loaded from any source and managed by a rich set of languages by using the Spark API. RDDs are lazily evaluated, meaning the operations are performed only when they are needed and not when they are declared [4]. Consequently, a benchmark test of Spark should consider the full consumption of the query results, and not just the creation and submission to the Driver.

Once an expression is submitted, the Driver node splits and distributes the relative RDD and code to the Workers and collects back the partial results. These are then summarized to obtain the final output. This process is completed by using the 'Shuffle' mechanism, where portions of data are copied between memory processes, or between machines. The Driver

node defines how and when Shuffle is needed, based on the type of operation requested. Shuffling is a costly process, and requires high-performance machine and fast memory, disk, and network I/O. To improve the efficiency of the Shuffling process, data and partial results are kept in memory until the memory is full [4]. Consequently, even relatively small datasets require a large amount of memory in the system to run fast and efficiently. Since memory is a critical element of the benchmarking process for Spark, the test dataset and queries should be complex enough to engage the Shuffle mechanism.

B. Scala

Spark is implemented using a language called Scala, which is a statistical language based on Java Virtual Machine and is used to interact directly with Spark [5]. This close relationship with Spark, allows Scala to be more efficient in the memory and network I/O, hence providing a faster response time compared to other APIs [6].

On the other hand, these benefits of Scala are counterweighed by the lack of good visualization tools and a reduced set of data science and machine learning libraries [7] compared to other languages like Python.

C. Python and PySpark

Python is a scripting language that has a rich set of libraries and modules dedicated on numerical computation and is considered as a top ranked programming language for data science [8].

PySpark is the official Spark API for Python. It allows for the use of all the advanced features of Python and its extended set of computational libraries, with the power of the Spark engine. However, using the Spark API requires that all the Spark operations on the RDD datasets must be serialized and then deserialized between Spark and Python. This process is not only costly in terms of memory, it is also costly in terms of time as the copy process of a large dataset is CPU intensive [6].

III. Solution Methodology

A. Data

To replicate real world settings, the benchmarking was performed on datasets of different sizes (10MB, 100MB, 200MB, 300MB and 500MB), different number of rows (10K, 100K, 200K, 300K and 500K), and different number of columns in each operation. A random data generator was written in Python to create the datasets with (1) 20 columns of random integer numbers, (2) 20 columns of random decimal numbers, (3) 10 columns of sentences containing 5 to 10 random words per sentence, selected randomly from a list of 466,000 English words [9], and (4) 10 columns of identifiers, where unique values in each column were repeated 10 times in that column. These columns were used predominantly to perform the grouping and joining operations.

B. Queries

Benchmarking was performed over 33 different operations that are outlined below. The query order was random and different in each benchmarking run. Each query represented atomic operations (from an end users' perspective). Complex queries such as subqueries were excluded since this would be a combination of queries ran for benchmarking and would not add any additional information to the results.

1) Row Operations

The objective of the row operations was to measure the performance of the system during operations involving one or more rows. In all, 8 queries were benchmarked for row operations. A set of queries were designed to test filtering operation and included (1) filtering rows based on a numeric value, (2) filtering rows based on matching set of characters, and (3) filtering rows based on wildcards.

Another set of queries was focused on performing more computationally intensive row operations where the data had to be shifted forward from one row to the next. This included (1) running sum calculations, and (2) shift (lag) operation

We also tested writing new rows to the dataset. This included (1) adding 100 new rows to the dataset, (2) adding 1,000 new rows to the dataset, and (3) adding 10,000 new rows to the dataset.

2) Column Operations:

The objective of the column operation was to measure the performance of the system during operations involving one or more columns. In all, 18 queries were benchmarked for column operations. Sorting operation was benchmarked using 6 queries - (1) ascending sort by 1 column, (2) ascending sort by 5 columns, (3) ascending sort by 10 columns, (4) descending sort by 1 column, (5) descending sort by 5 columns, and (6) descending sort by 10 columns.

We performed benchmarking of splitting and merging operations which included (1) splitting 1 column into 5 columns, (2) splitting 1 column into 10 columns, (3) merging 2 columns into 1 column, (4) merging 5 columns into 1 column, and (5) merging 10 columns into 1 column.

We also looked at mathematical operations applied to columns. This included a single query that applied the following 5 functions to different numeric columns in the dataset - sum, average, count, minimum, and maximum.

Finally, we benchmarked one of the most computationally intensive operations in this category – the join operation. A secondary dataset was created for this purpose containing half the number of rows as the main dataset, to allow enough records with and without a corresponding key in the secondary table. The join queries that were benchmarked included (1) full outer join using 5 columns, (2) full outer join using 10 columns, (3) inner join using 5 columns, (4) inner join using 10 columns, (5) left outer join using 5 columns, and (6) left outer join using 10 columns.

3) Aggregate Operations

The objective of the aggregation operations was to measure the performance of the languages when queries were performed on groups of data. In all, 4 queries were performed in this category which included grouping the rows based on the values in one of the identifier columns and ranking the

rows in each group based on the values in a numeric column. This category also included calculating count, sum, average, minimum, and maximum value within groups when grouping was done based on 1 column, 5 columns, and 10 identifier columns.

4) Mixed Operations

The objective of the mixed operation was to measure the performance of aggregate queries that impacted both rows and columns. This was done by performing a 'pivot' operation where a new table is calculated based on the existing dataset. A total of 3 queries were executed in this category. The unique row values in a column of the original dataset were converted into individual columns in the new table. The rows identifiers of this new dataset consisted of the discrete value from (1) 1 column, (2) 5 columns, and (3) 10 columns of the original dataset. Finally, the values in the rows and column intersections of this new dataset were the sum of a floating-point column from the original dataset.

C. Hardware Setups & Libraries Used

We performed the study using three different setups as described below. All these systems were accessed from two different computers (except the 4-node cluster which was accessed from only 1 machine due to cost constraints) and all datasets were run in both Scala and PySpark. This blocked out the effect of computer to computer and setup variation on the benchmarking (a confounding factor).

1) Virtual Machine (VM) running on a local machine.

This environment was managed by Oracle VM Virtual Box software, version 5.2.26 r128414 (Qt5.6.2). The VM was configured with Ubuntu Operating System (version 18.04.2 release 18.04), with 4GB of RAM, 20GB of SSD disk space, 4 CPUs, 80MB of Video Memory with 3D acceleration. The experiment was run on Jupyter Notebook version 5.2.2 with Python for Linux version 3.6.7 and PySpark version 2.4.0. We installed Spark version 2.4.0, Hadoop version 2.7, and Java version 10.0.2 on these systems.

While the Scala runs completed with 2GB of swap memory, the PySpark runs needed 4GB of swap memory to complete for the larger datasets. Hence this change was made for the PySpark runs to allow for data collection.

2) Single Node Amazon Web Services (AWS) Machine using Databricks

Databricks is a cloud platform that provides a unified framework for data processing. The main advantage of using Databricks is the immediate availability of Scala, Python, SQL, and R environments. The environment leverages a standard and optimized configuration that can be customized based on user needs. While Databricks stores the code and the data, it uses Amazon or Microsoft Azure machines to perform the data operations [10].

We used a configuration with a single AWS node, with 6.0GB of memory, 0.88 Cores (standard configuration in Databricks). Databricks runtime 5.2 was used, which included Scala version 2.11, Spark version 2.4.0, Python version 3.5, PySpark version 2.4.0 and Java version 1.8.0_191.

3) 4-node Cluster on AWS using Databricks

For this setup, we used a cluster of on-demand machines with 1 master node and 2-3 slave nodes. All machines used the "m5.xlarge" configuration with 16GB memory and 4 cores. Databricks runtime 5.2 was used, which included Scala version 2.11, Spark version 2.4.0, Python version 3.5, PySpark version 2.4.0 and Java version 1.8.0_191.

D. Metrics

The metric used for benchmarking was the *throughput* in MB of data processed per seconds, computed by dividing the *absolute time taken* to perform the queries with the dataset size. In order to avoid human error and ensure consistency across setups, custom code was written in the native languages (Scala and Python) to compute the metric and automatic logging capability of the metric was also developed.

To account for the statistical variations between runs, we ran each query in PySpark and Scala a total of *5 times x 2 machines = 10 runs* (for each of the 5 datasets). The statistical analysis was based on the median of the results using Wilcoxon Rank Sum Test, due to the right-skewness in the metric.

We flushed the memory with an initial run of the queries and discarded its results for the benchmark. This was done to account for warm-cache results only [11].

Table I
DESIGN OF EXPERIMENT SUMMARY

Randomization	Data Generation
	Execution order of queries for each run
Blocking for Confounding Factors	Machine: 2 different machines used to run all queries in both languages
	Setup – 3 setups used (Local Virtual Machine, Single Node AWS machine through Databricks), and 4 node cluster on AWS using Databricks
	Data Size: 10MB, 100MB, 200MB, 300MB, 500MB
	Query: 33 different queries benchmarked
	Memory Flush (Warn Cache results only)
Metrics	Throughput: Data processed per second (MB/sec)

IV. RESULTS

The results are shown by the blocked variable "Setup" and "Dataset Size" since the performance varied considerably based on these variables. Results from machine 1 and machine 2 were combined since they did not show any statistically significant differences expect for PySpark queries on Databricks (Single Node). Even there, the results were not practically significant (~5MB/sec difference in throughput between the machines) and hence the combination of the results from the machine 1 and machine 2 was justified. While this paper contains all relevant conclusions from this study, the full source code is publicly available on GitHub [12] for further exploration.

A. Row Operations

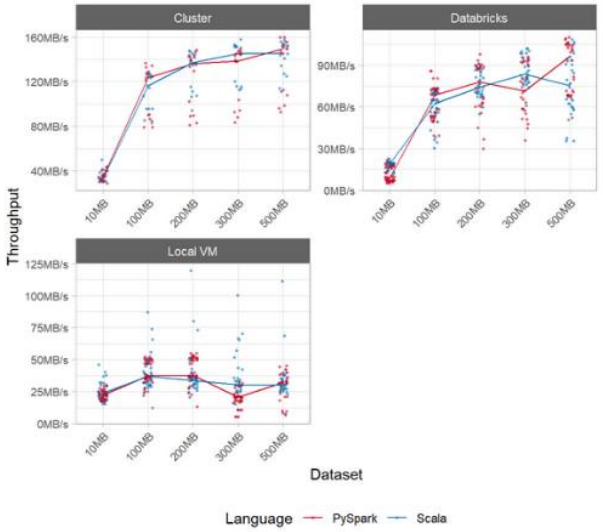

Fig. 1. Comparing filtering operation by language.

The row operations show a consistent pattern across all setups (Local VM, Databricks – Single Node, Databricks – Cluster). Except for filter operations (Fig. 1) where the performance of both Scala and PySpark is comparable, row operations consistently show Scala outperforming PySpark.

Fig. 2 shows that Scala is 4.5x – 13.5x faster than PySpark for Running Sum and Shift (lag) operations and is 1.2x – 2.4x faster than PySpark for writing new rows. Both these results were statistically significant. Also, the performance difference on the cluster setup was significantly higher than that on the single node configurations (Databricks and Local VM).

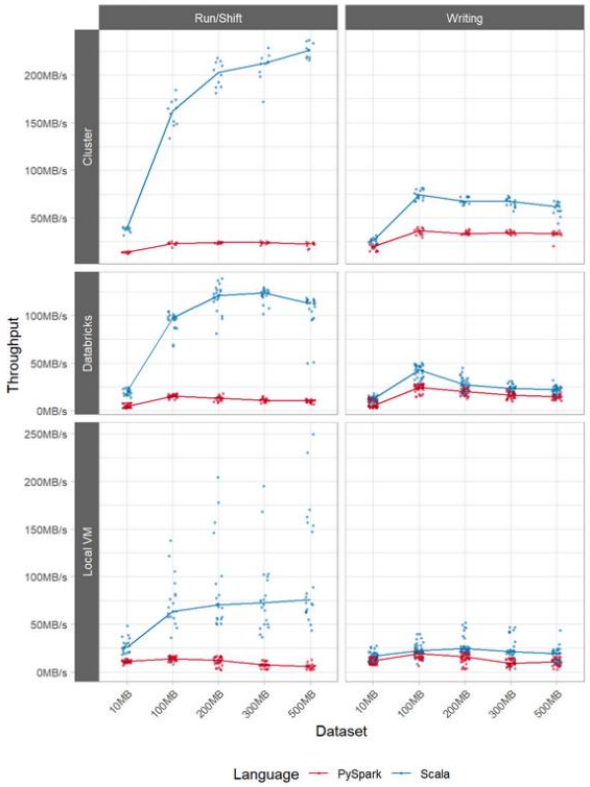

Fig. 2. Comparing performance of Running Sum/Shift and writing new rows by language.

B. Column Operations

Fig. 3 shows that Scala was 1.2x – 2.6x faster than PySpark for merging columns and 1.2x – 3.8x faster than PySpark for splitting columns. All these differences were statistically significant, and the performance difference was consistent across dataset size. Like some row operations, this figure also shows that the performance gap increases as we go from a single node setup to a cluster of nodes.

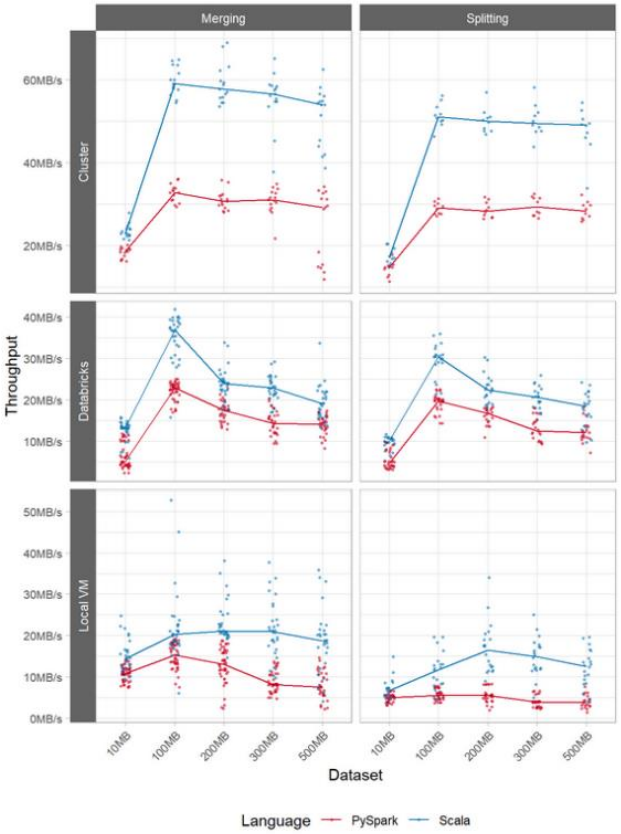

Fig. 3. Comparing column operation – Merging/ Splitting by language

Sorting operation did not show significant difference between Scala and PySpark on a single machine setup (Databricks and Local VM), especially for smaller datasets (<= 200MB). For larger datasets, Scala started to outperform PySpark (Fig. 4). More significantly, as we moved to a distributed cluster setup, the difference in performance between Scala and PySpark was more apparent (Scala's throughput was ~ 1.4x that of PySpark).

The join operation showed that the performance between Scala and PySpark was virtually identical on the VM, but differences were seen in single node Databricks and the Cluster setup (Fig. 4). These differences however were not as large as the differences for the merge/split operation.

For mathematical operations, the performance of Scala and PySpark were essentially identical on the cluster setup, with PySpark even outperforming Scala in on the single node setups (Databricks and Local VM) for some datasets (Fig. 5).

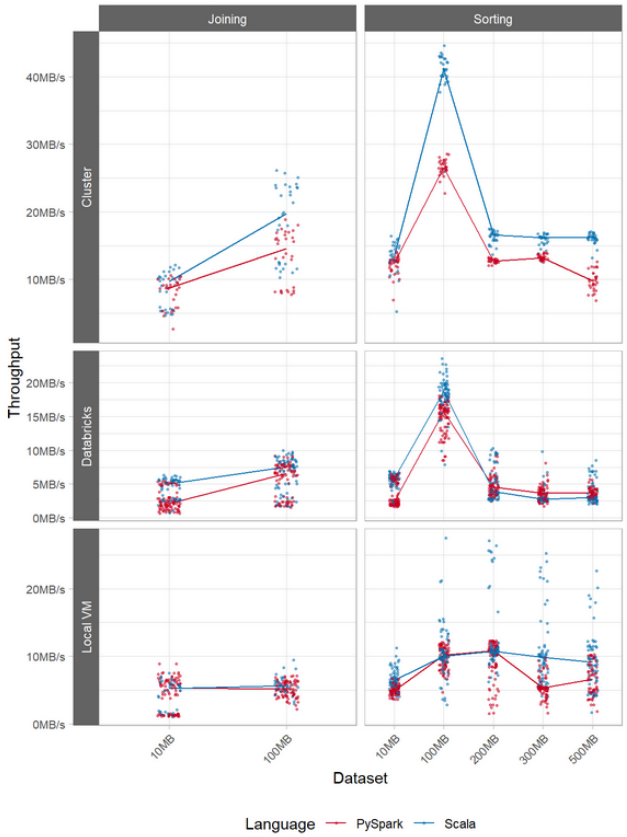

Fig. 4. Comparing column operation – Joining and Sorting by language

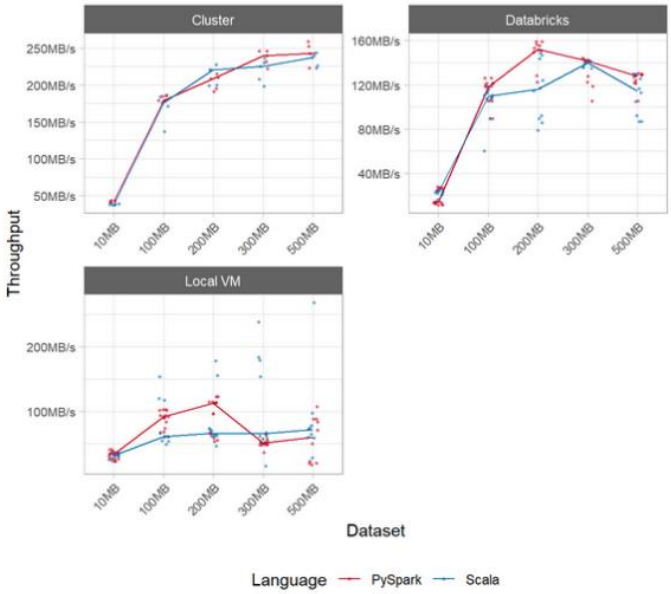

Fig. 5. Comparing column operation – mathematical operations by language

C. *Aggregate Operations*

For the ranking operation, the Local VM and Single Node Databricks setup had significantly higher throughput (1.1x – 2.7x) for Scala compared to PySpark (Fig. 6). This was equivalent to a difference in throughput of 5MB/s to 10MB/s. Similar to some row and column operation, the cluster environment shows a larger gap in the performance between Scala and PySpark compared to a single node setup. Starting from the 100MB dataset, Scala's throughput was 15MB/sec – 20MB/sec faster than that of PySpark.

Grouping operations for the single node Databricks and Cluster environments were not very different between PySpark and Scala (Excluding the 10MB dataset, Scala to PySpark throughput ratio varied from 0.77 – 1.23). However, on the local VM, Scala consistently outperformed Python by more than 1.5x.

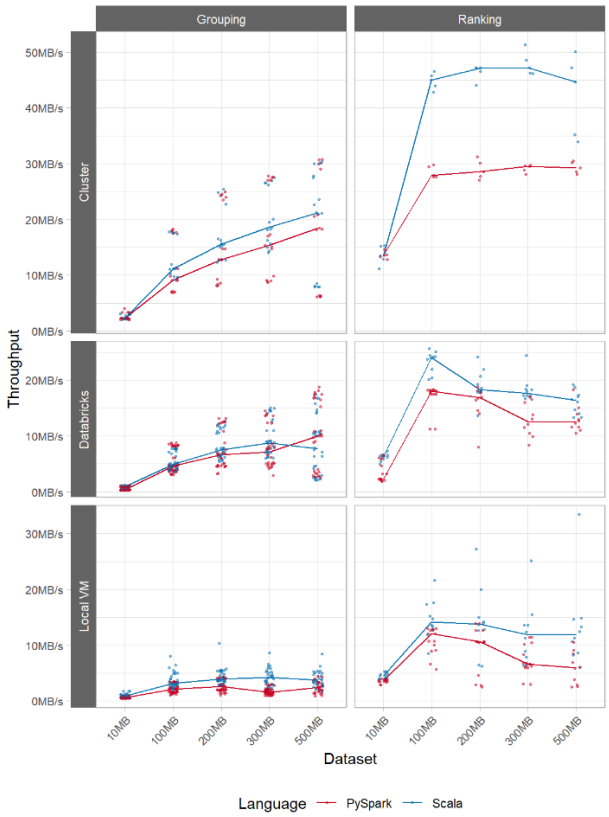

Fig. 6. Comparing ranking operation between environments.

D. *Mixed Operations*

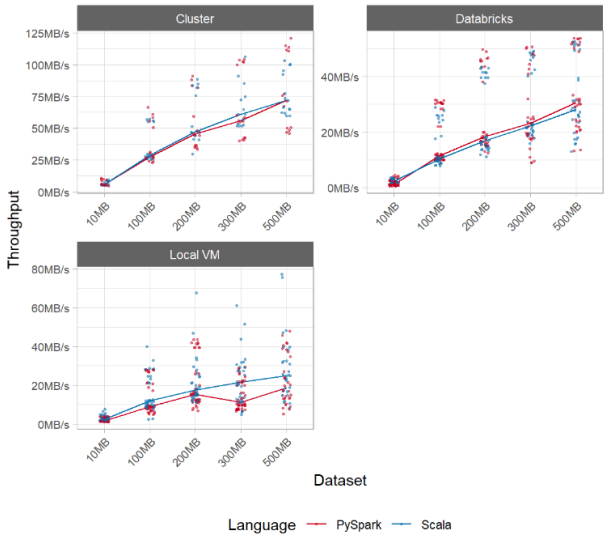

Fig. 7. Comparing Mixed operations across languages and Environments

Pivot operations had similar performance for PySpark and Scala on single node Databricks and 4 node cluster, with an overall increasing throughput with higher datasets (Fig. 7). However, the Local VM results showed that Scala consistently outperformed PySpark, especially for datasets larger than 200MB.

TABLE II
RATIO BETWEEN SCALA AND PYSPARK MEDIAN THROUGHPUTS FOR LOCAL VM ENVIRONMENT

Query / Datasets	10MB	100MB	200MB	300MB	500MB
Rows Operations					
Filtering	1.08	1.00	0.89	1.49	0.94
Running/Shift	2.33	4.51	5.80	9.73	13.52
Writing	1.42	1.18	1.53	2.36	1.83
Column Operations					
Joining	1.68	1.10			
Mathematics	0.95	0.67	0.59	1.29	1.20
Merging	1.33	1.32	1.61	2.56	2.49
Sorting	1.31	0.99	0.99	1.86	1.38
Splitting	1.35	2.15	3.00	3.80	3.30
Aggregate Operations					
Grouping	1.51	1.46	1.48	2.75	1.58
Ranking in Groups	1.23	1.18	1.30	1.79	2.00
Mixed Operations					
Pivots	1.63	1.33	1.16	1.94	1.35

TABLE III
RATIO BETWEEN SCALA AND PYSPARK MEDIAN THROUGHPUTS FOR THE SINGLE NODE DATABRICKS ENVIRONMENT

Query / Datasets	10MB	100MB	200MB	300MB	500MB
Rows Operations					
Filtering	2.16	0.91	0.95	1.17	0.78
Running/Shift	4.61	6.31	9.37	11.19	10.71
Writing	2.43	1.75	1.37	1.42	1.49
Column Operations					
Joining	2.41	1.15			
Mathematics	1.65	0.93	0.76	0.99	0.90
Merging	2.56	1.61	1.37	1.59	1.35
Sorting	2.36	1.19	0.85	0.76	0.83
Splitting	2.11	1.54	1.35	1.66	1.53
Aggregate Operations					
Grouping	2.41	1.06	1.13	1.23	0.77
Ranking in Groups	2.70	1.33	1.09	1.41	1.31
Mixed Operations					
Pivots	2.10	0.90	0.92	0.96	0.92

TABLE IV
RATIO BETWEEN SCALA AND PYSPARK MEDIAN THROUGHPUTS FOR THE CLUSTER ENVIRONMENT

Query / Datasets	10MB	100MB	200MB	300MB	500MB
Rows Operations					
Filtering	0.96	0.94	1.01	1.05	0.97
Running/Shift	2.81	7.03	8.47	8.91	9.99
Writing	1.30	2.04	2.02	1.96	1.86
Column Operations					
Joining	1.39	1.36			
Mathematics	0.91	0.98	1.06	0.94	0.98
Merging	1.23	1.81	1.88	1.83	1.85
Sorting	1.09	1.55	1.30	1.23	1.65
Splitting	1.16	1.75	1.77	1.69	1.74
Aggregate Operations					
Grouping	0.99	1.21	1.22	1.22	1.15
Ranking in Groups	0.99	1.61	1.65	1.60	1.53
Mixed Operations					
Pivots	0.96	1.05	1.04	1.09	1.00

V. ANALYSIS

- *Cluster vs. Single Machine:* In several results ranging across operation types (example, running sum/shift, merging, splitting, sorting, and ranking within groups), the performance difference between Scala and PySpark increased as we moved from a single machine setup (Local VM and Databricks) to a Cluster setup. This was evidence that Scala better utilized the benefits of a distributed framework compared to PySpark.
- *Memory Limitations:* Both PySpark and Scala had memory issues while running the join operations on the larger datasets (≥ 200MB). Also, while running many PySpark queries on the local VM, we noticed that the queries would not run with the same swap memory settings as Scala (even for datasets ≤ 100MB). Hence, we had to increase the swap memory settings for PySpark to 4GB (vs. 2GB for the Scala). In addition, on the VM where memory was limited, the performance of the PySpark queries degraded as the dataset size increased (for example in Aggregate and Mixed operations). While Scala's performance also degraded, it did not degrade as much as that of PySpark. (Table II). Both these findings point to two important facts – (1) Both Scala and PySpark are memory intensive, and (2) PySpark is more memory hungry than Scala.
- *Unaccounted Confounding Variables*: While most results show that Scala outperformed PySpark, for many of the column operations PySpark's performance was comparable to that of Scala. The increase in swap memory for PySpark to 4GB might be a confounding factor that affected this result. For example, a quick look at the setup wise data comparison for joins (Fig. 4) shows that the performance for Scala is much higher than PySpark on Databricks single machine and cluster where the setup was identical between Scala and PySpark, but the performance was comparable on the VM where PySpark runs had higher swap memory.
- *Warm-cache findings:* In our literature survey, we had noted that for warm-cache runs, there is potential for the first run to take longer than the subsequent runs. In our analysis, we did not find any substantial difference in the run times between runs (Fig. 8). Even then, to avoid any confounding effects, we discarded the first from our analysis to stay consistent with the literature. We don't expect the inclusion of RunID 1 in our analysis to impact the conclusions since the results of this run were consistent with the other runs.

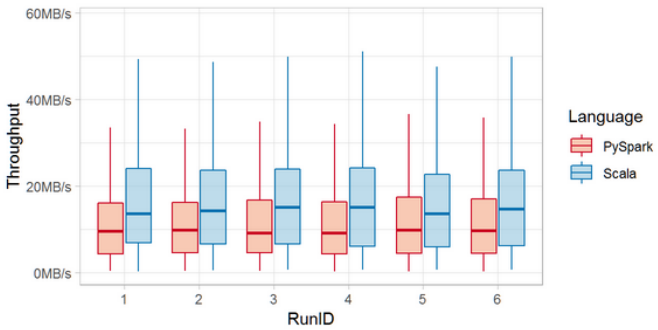

Fig. 8. Throughput comparison between runs

- *Generalization:* This study can be classified as a randomized experiment. Many confounding factors were accounted for through randomization of variables (data and order of queries). In addition, many other confounding factors (dataset size, setup and machine used) were blocked in the experiment and a wide variety of data queries were used for benchmarking. Hence, the results of this study can be generalized to real work use cases. The only caveat is that the tests were performed on datasets up to 500MB, so these results only apply to datasets up to this limit. To draw conclusions to bigger datasets, more analysis would be needed.

VI. Conclusion

We set out to answer two questions with this study – (1) How does the performance of Scala compare with PySpark from an end user perspective after accounting for confounding factors, and (2) For a company looking to invest in a big data infrastructure, should they continue using a platform in which they have already made considerable investments (Python), or should they invest in a new platform (Scala) due to its speed and performance advantage.

Based on this analysis, the answer to the first question is that Scala is indeed faster than PySpark for several common queries with the added benefit of Scala requiring less memory to operate compared to PySpark.

As for the second question, the answer is not so straightforward. The Python framework does have its advantages in that it has a mature ecosystem of machine learning libraries available at its disposal. Although some of these algorithms are available in the Scala environment as well, the machine learning environment in Scala is not as mature as that in Python. Our recommendation to data scientists working with large datasets would be to evaluate the algorithms they plan to use. If all the algorithms are available in Scala, our recommendation would be to use Scala due to its speed and memory advantages over PySpark. If only a subset of the algorithms is available in Scala, our recommendation would be to use Python and the PySpark libraries instead. To make this more efficient, we recommend (1) using an optimized and scalable cloud computing platform like AWS/Databricks, (2) leverage PySpark's machine learning libraries when the algorithms are available in Spark, and (3) revert to using native Python algorithms when they are unavailable in PySpark.

VII. References

[1] Spark, "Spark API Documentation," [Online]. Available: https://spark.apache.org/docs/2.3.1/api.html.

[2] P. Gandhi, "Apache Spark : Python vs. Scala," KDnuggets, May 2018. [Online]. Available: https://www.kdnuggets.com/2018/05/apache-spark-python-scala.html.

[3] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker and I. Stoica, "Apache Spark: A Unified Engine For Big Data Processing," *Communications of the ACM,* vol. 59, no. 11, pp. 56-65, November 2016.

[4] N. Gureev, "Hive, Spark, Presto for Interactive Queries on Big Data," KTH Royal Institute of Technology, Stockholm, 2018.

[5] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker and I. Stoica, "Spark: Cluster Computing with Working Sets," University of California, Berkeley.

[6] S. Akhadov, "PySpark at Bare-Metal Speed," ETH, Zurich, 2017.

[7] Dezyre, "Scala vs. Python for Apache Spark," Iconiq Inc, March 2019. [Online]. Available: https://www.dezyre.com/article/scala-vs-python-for-apache-spark/213. [Accessed 13 April 2019].

[8] S. Cass, "The 2018 Top Programming Languages," 31 Jul 2018. [Online]. Available: https://spectrum.ieee.org/at-work/innovation/the-2018-top-programming-languages. [Accessed 10 Apr 2019].

[9] DWYL, [Online]. Available: https://github.com/dwyl/english-words. [Accessed 10 03 2019].

[10] Databricks, "Databricks FAQ," Databricks, 2018. [Online]. Available: https://databricks.com/product/faq. [Accessed 15 03 2019].

[11] A. Traeger , E. Zadok, N. Joukov and C. P. Wright, "A Nine Year Study of File System and Storage Benchmarking," 2007.

[12] M. Moro and N. Gupta, "https://github.com/maxmoro/SMU-DB7330-Spark_python," 15 Apr 2019. [Online].