Analysis and Comparison of Dockerized and Standalone Apache Spark Configurations for Efficient Distributed Data Processing.

Alain P. Ndigande
Department of Computer Science
Ozyegin University
Istanbul, Turkiye
alain.ndigande@ozu.edu.tr

Ismail Ari
Department of Computer Science
Ozyegin University
Istanbul, Turkiye
ismail.ari@ozyegin.edu.tr

Sedat Ozer
Department of Electrical and Computer Engineering
California State Polytechnic University, Pomona
Pomona, USA
sedatist@gmail.com

***Abstract*—Apache Spark, a powerful distributed computing framework, has become a key to handling large-scale data processing tasks in many applications, including signal processing. However, there are different considerations for performance, cost, or ease of use during the development and deployment stages. This paper benchmarks Apache Spark on different local setups in terms of performance and elaborates on alternative cloud computing costs. Performances of different Spark master-slave configurations with Docker and no Docker are evaluated over different numbers of worker nodes, cores, memory, and executors. Spark WordCount benchmark is tested on Wikipedia datasets of sizes ranging from 1GB to 25GB. The results reveal that in local setups Docker creates additional parameter complexity as well as performance overheads, therefore a "no Docker" setup is a better choice. We also observe the dominance of I/O bottlenecks in local setups. These results can help practitioners choose optimal setups for different Development and Operations (DevOps) and big data processing scenarios.**

***Index Terms*—Apache Spark, Docker, WordCount, Executor, RDD, HDFS, Cloud, Distributed System.**

I. Introduction

In the dynamic landscape of distributed data processing and intelligent systems, Apache Spark stands as a versatile framework, which is widely known for its speed, scalability, and user-friendly design [1]. Apache Spark addresses the challenges posed by big data, characterized in terms of volume, variety, velocity, and veracity (4Vs) [2] for those performing analytical tasks with the demand for efficient, fast, and scalable solutions. It has become instrumental in parallel data processing across computer clusters [3], [4] for many signal processing applications [5], [6] and big data processing tasks such as association rule mining (ARM) [7].

Intelligent systems, which leverage artificial intelligence to process and analyze data, are increasingly being integrated with sensors networks to monitor environments, detect anomalies, and make data-driven decisions in real time [8], [9]. These sensor networks generate vast amounts of data continuously, which poses significant challenges in terms of storage, processing, and real-time analytics (ETL/ELT). Apache Spark, with its in-memory computing capabilities and powerful distributed processing framework, helps overcome these challenges. Spark can process large volumes of streaming data from sensors quickly and in parallel, facilitating real-time insights and actionable intelligence. Using Spark's machine learning libraries, intelligent systems can also perform advanced analytics and predictive modeling on sensor data, enhancing their ability to make accurate and timely decisions [10]–[12].

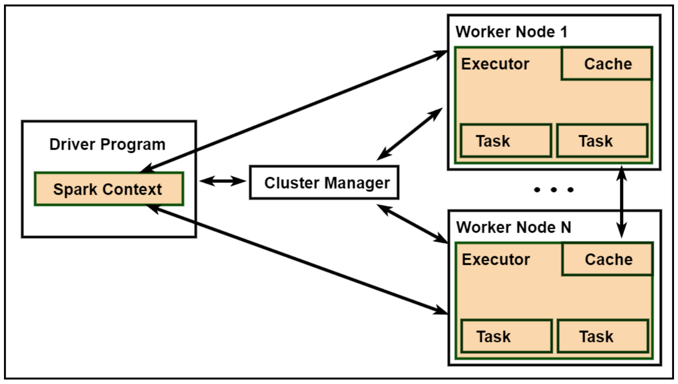

Fig. 1: Apache Spark overview: Distributed applications composed of tasks, are coordinated by the SparkContext object in the driver program. SparkContext connects to cluster managers which allocate resources across worker nodes and executors.

This study aims to benchmark Apache Spark with PySpark in a local environment with and without Docker. We are also motivated by the increasing financial considerations associated with available public cloud services. Although platforms such as Amazon Elastic MapReduce (EMR) [13] are suggested for production setups, as they offer platforms as a service (PaaS) for processing of large-scale workloads. However, if not strictly-controlled, associated costs can pose a challenge for non-production (development) scenarios. Increasing number of nodes, virtual compute instances, memory and storage, multiplied by execution time, contribute to fast growth in operational expenses. Therefore, local cluster setups emerge as an economical alternative, offering the flexibility to experiment with emulated cluster configurations, model development, and

979-8-3503-7943-3/24/$31.00 ©2024 IEEE

dataset preparation at a lower price. Furthermore, this approach provides an opportunity to explore diverse parameters in a controlled environment, enriching the understanding of Spark's behavior under varying load conditions.

II. Background and Related Work

Apache Spark is a distributed data processing engine designed for large-scale data analysis and machine learning [3], [14], [15]. It provides a high-level API in Python, Scala, Java, and R, enabling developers to process and analyze massive datasets. Spark's architecture shown in Figure 1, allows it to distribute data and computations across multiple machines, significantly improving performance compared to traditional single-machine processing. Spark applications run as independent sets of processes coordinated by the SparkContext object in the main (*i.e.* driver) program. SparkContext can connect to several types of cluster manager, including the standalone administrator, Apache YARN [16] or Apache Mesos [17]. These manage resources and scheduling across the nodes in the cluster. This paper uses the standard standalone cluster configuration.

A key feature that enables Spark is the Resilient Distributed Dataset (RDD), which is a fault-tolerant and immutable core data structure that represents distributed data collections across the cluster. Spark optimizes RDD transformations (*map, filter, join, etc.*) and actions (*count, collect, etc.*) by rearranging them and choosing the most efficient physical execution plan [18]–[20]. Spark also improves performance by utilizing in-memory caching and minimizing data shuffling across the cluster.

Docker Containers are lightweight, standalone, executable packages of software that include the code, runtime, system tools, and libraries needed to run an application [21]. Containers provide isolation and portability, enabling applications to run across different environments without conflicts or dependencies on the underlying system. Docker Compose is a tool for defining and running multi-container applications, enabling developers to launch complex applications easily. In our system, Docker Compose is used for local cluster setups as well as limiting resource allocations to instances.

The marriage of Apache Spark and Docker has caught the attention of researchers exploring containerization benefits. Previous research has looked at benchmarking Spark [4], [18], [22] and containerized environment performance tests [23], [24]. These studies have explored various aspects including performance comparisons [25], scalability assessments [26], and resource utilization analysis [23]. Although these works contribute valuable insights, our approach distinguishes itself by focusing on localized experimentation using Docker, bringing a practical perspective to configuration optimization. In their study, Duarte, *et al.* [27] compare Spark performance in big data, focusing on batch and query processing tasks. Spark demonstrates superiority in handling workloads with inter-record dependencies, such as sorting and joining, leveraging the distributed storage and resource management abstractions provided by the Hadoop stack [28]. Hriday, *et al.* [25] give a comprehensive comparison of Apache Spark, comparing it against Hadoop MapReduce, Apache Flink, Apache Storm, and Apache Samza. Although their work sheds light on the relative strengths of these frameworks, our study differs by emphasizing localized experimentation using Docker for practical insights into everyday usage. In exploring the impact of memory size on big data processing, Han, *et al.* [26] compared the performance of Hadoop and Spark clusters using the K-means algorithm from the HiBench benchmark. Their findings reveal that Spark exhibits superior performance when memory size is adequate for the data, but Hadoop surpasses Spark as data size increases due to performance degradation in Spark when replacing disk data with memory-cached data beyond the memory cache limit.

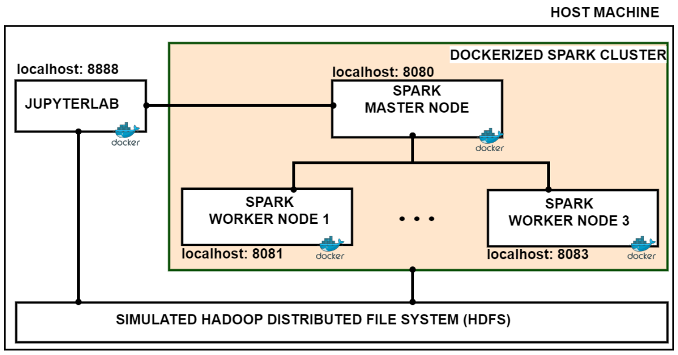

Fig. 2: Overview of our experiment system. A standalone Spark Docker cluster can be established with one master and multiple worker nodes, which are all containers. Jupyter notebook allows interactive scripting making it easy to debug and track the jobs. HDFS is simulated through shared volumes specified in the DockerFile.

Although existing work offers valuable insights into Spark benchmarking, dockerized environments, cost considerations, and challenges, our study distinguishes itself by combining these aspects in a unique context, local experimentation using Docker and PySpark. Furthermore, our work sheds light on custom setups in cloud or on premises as an alternative to using PaaS solutions that may incur additional charges. The following sections will detail specifics of our experimental setups and results, highlighting contributions from this work.

III. Methodology

Our experimentation begins with the establishment of a local Apache Spark cluster using Docker Compose. Figure 2 shows our system, which consists of a master node and a configurable number of worker nodes. Containerization ensures the encapsulation of each Spark component, allowing for easy deployment and scalability. This localized environment aims to emulate real-world scenarios with the flexibility to dynamically adjust configurations. We specifically use PySpark for seamless interaction with the Spark engine using Python. Additionally, a container with JupyterLab offers an interactive and user-friendly environment for script execution and result visualization. For real-time insights and cluster health, exposed

TABLE I: Configuration details for setups 1-2-3.

Configuration	Executors	Cores	Memory (GB)
E1C4M4	1	4	4
E1C1M4	1	1	4
E1C2M2	1	2	2
E12C1M1	12	1	1
E6C2M2	6	2	2
E3C3M4	3	3	4

ports facilitate monitoring, and history server access enhances visibility into job execution history. For performance evaluation, we record the execution times for each task. We leverage Spark's built-in monitoring capabilities along with additional logging to extract detailed insights.

Building on the Github[1] codebase, we explore a spectrum of configurations to understand their impact on Spark's performance. The key parameters are the number of worker nodes, executors, the number of cores per executor, and the executor memory size. By varying these factors, we explore the relationships between configuration settings and overall system efficiency.

IV. Experiments

Configuration: By using Docker vs. no Docker and modifying the number of workers, we create three different setups on top of which we vary the amount of executors to create alternative configurations:

- *Setup 1*: A local PySpark installation without Docker.
- *Setup 2*: A standalone cluster on Docker with 1 master node and 1 worker node.
- *Setup 3*: A standalone cluster on Docker with 1 master node and 3 worker nodes.

As shown in Table I, we create experimental configurations by varying number of cores and memory associated with different number of executors. Configuration names are encoded with initials and amount of resource counts. We obtain 6 different configurations for each setup. We examine the time it takes to complete the given job.

Dataset: To measure the performance of our Spark setup, we compile datasets of varying sizes from English Wikipedia articles[2]. Ranging from 1GB to 25GB, these datasets constitute diverse data loads that can be stored in simulated HDFS [29] shared volumes. The task at hand for benchmarking involves executing the WordCount application on these datasets, allowing us to measure execution times and resource utilization across different configurations. The master and worker nodes expose ports that are used as proxy to the master, driver and history server that can be accessed from a web interface allowing us to visually observe the cluster.

The experiments were run on Dell G15 Intel(R) Core(TM) i7-12700H with 14 cores (20 logical processors), SSD (Samsung 1024GB NVMe), and 16 GB DRAM with Ubuntu 22.04 as the host operating system. Docker 24.0.5-ubuntu1, Docker-compose 1.29 were used for containerization. All experiments were run with SPARK-VERSION=3.5 and Python 3.10. By limiting the cores and memory of the worker containers, we controlled the capacity that each worker node can have.

[1]https://github.com/mrn-aglic/spark-standalone-cluster

[2]https://www.kaggle.com/datasets/yauangon/enwiki20201020-energy

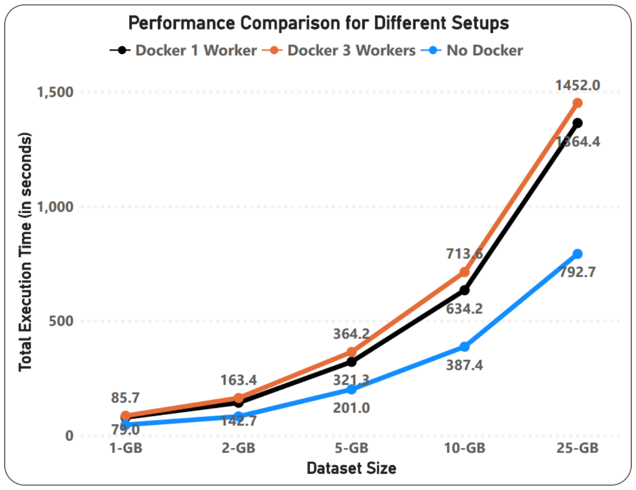

Fig. 3: Comparison of computation time with and without Docker. This plot shows no Docker setup performs better and indicates a small degradation for more workers.

V. Results

Fig. 3 shows that the setup (1) with no Docker has lower processing times compared to setups with Docker (2-3). The performance difference increases ∼2x as the size of the dataset increases. We also observe that increasing worker counts from 1 to 3 increases processing time, causing ∼ 10% slowdown. This can be attributed to the additional communication overhead among the 3 worker nodes and the master node.

Table II shows the processing times (in seconds) on Wikipedia datasets for different setups (no Docker vs. Docker), configurations, and data set sizes. The tables are sorted from left to right, generally in decreasing order of performance (increasing times). Finally, a potential *SpeedUp=Max(time)/Min(time)* value is given for different configurations. As a starting reference in Setup 3, we see that having more executors (E12C1M1) benefits from the parallelism and available resources achieving better performance, whereas lack of parallelism in executors (E1C1M4) leaves the nodes under-utilized leading to the worst performance. In setup 2, the same configuration (E1C1M4) is again the worst performer. However, E12C1M1 is not the best performer in Setup 2, because there is only 1 worker and executors can not benefit from parallelism. In setups 2-3 with Docker, there are speed-ups ranging from ∼2.40x to ∼3.07x among different configurations. In setup 1 (no Docker), we did not observe a significant performance difference among different configurations (*SpeedUp*∼1x). Spark standalone resource manager dynamically allocates all available resources to the selected configurations for best utilization. Our performance

TABLE II: Spark WordCount benchmark execution times (in seconds) for different setups and configurations.

Benchmark	E1C1M4	E1C2M2	E1C4M4	E3C3M4	E12C1M1	E6C2M2	SpeedUp
1-GB	**41.2**	41.5	41.5	43.9	***46.9***	46.3	1.14
2-GB	**78.0**	78.2	78.8	83.0	***83.6***	82.7	1.07
5-GB	195.4	**189.3**	189.5	200.3	198.1	***201.0***	1.06
10-GB	**362.5**	367.4	367.1	373.8	376.8	***387.4***	1.07
25-GB	***799.0***	777.2	**775.3**	796.4	789.7	792.7	1.03

(a) **Setup1**: Standalone No Docker

Benchmark	E1C2M2	E6C2M2	E1C4M4	E12C1M1	E3C3M4	E1C1M4	SpeedUp
1-GB	**78.4**	79.0	80.5	87.1	91.9	***206.6***	2.64
2-GB	**140.6**	142.7	150.6	151.4	170.1	***380.0***	2.70
5-GB	326.5	**321.3**	335.4	330.3	387.3	***985.5***	3.07
10-GB	**625.3**	634.2	652.0	633.3	773.2	***1893.7***	3.03
25-GB	**1319.9**	1364.4	1379.3	1359.3	1758.4	***4007.2***	3.04

(b) **Setup2:** Standalone-Docker 1 Worker

Benchmark	E12C1M1	E1C2M2	E1C4M4	E6C2M2	E3C3M4	E1C1M4	SpeedUp
1-GB	89.8	**83.0**	85.6	85.7	94.7	***199.5***	2.40
2-GB	**151.4**	153.8	154.5	163.4	172.7	***380.2***	2.51
5-GB	**339.5**	364.5	353.6	364.2	414.6	***943.6***	2.78
10-GB	**642.3**	710.6	677.3	713.6	798.8	***1895.9***	2.95
25-GB	**1340.1**	1461.1	1424.8	1452.0	1718.9	***4053.8***	3.02

(c) **Setup3:** Standalone-Docker 3 Workers

may be limited by the I/O bandwidth since there is a linear relationship between the dataset sizes and the processing times (*e.g.* 1GB/41.2sec$\sim$25MB/sec). We also tested a subset of our benchmarks with Gzip data compression. We could not observe significant performance gains, which we attribute to the additional CPU, memory, and I/O costs associated with the compression and decompression operations (results omitted here for brevity).

Localized setups also offer cost advantages. For example, we conducted this research using 3 nodes and executed 50 hours/week for 1 month. If we were to run this cluster with Elastic Compute Cloud (EC2) *m7i.xlarge* instances the prices would be as follows;

$$\begin{aligned}\text{Estimated cost} &= \text{Time} \times ((\text{EC2 instances} \times \text{vCPU-hour EC2}) + \text{vCPU-hour EMR})\\ &= (50 \times 4) \times ((3 \times \$0.2016) + \$0.0504)\\ &= \$131.04\end{aligned}$$

With a monthly $\$131.04$ cost, it starts to justify investing in servers, *i.e.* chosing Capital Expense (CAPEX) over cloud Operational Expenses (OPEX). Yet, if you need the elasticity and scale, you can configure an Amazon EMR 30-node cluster and run it for 1 hour or equivalently configure a 3-node cluster and run it for 10 hours for the same price.

VI. Conclusion

This study offers a valuable resource for (big data processing) practitioners looking to optimize Apache Spark deployments in local clusters before migrating to a production environment, emphasizing the importance of configurations parameters for specific workloads. Docker results in containers fighting for limited resources, causing overhead when run locally. These preliminary findings contribute to the understanding of Spark's behavior in emulated environments. Understanding these limitations is crucial for interpreting our results and considering the broader implications for development and operations.

While our work provides valuable insights into the comparison between dockerized and standalone Spark configurations, there is a limitation of solely relying on the classical WordCount benchmark. We focused on having an easy to spin up infrastructure that can run various data processing workloads and as such can easily be extended. We initiated our experimentations with docker and then Vagrant, and we continue to explore other relevant alternatives including recent trends (Kubernetes Orchestration, Amazon EKS, serverless Spark offerings). For a better representation of different Spark workloads and enhance the generalizability of our findings, we intend to extend our analysis to incorporate various benchmarks including TPC-DS[3] and new 4G-5G mobile telecommunication datasets. Additionally, while our experiments were conducted on a single machine for controlled comparisons, in-depth analysis on more varied or distributed environments may give a deeper explanation on key points such as network and I/O bottlenecks to supplement our findings.

In our future work, we plan to directly evaluate the performance of these setups on a variety of tasks such as evaluating the performance of different ARM methods with respect to different configurations. Furthermore, we plan to explore hybrid approaches that leverage both localized environments for development and cloud services for production, striking a balance between cost-effectiveness and scalability.

References

[1] B. Chambers and M. Zaharia, *Spark: The definitive guide: Big data processing made simple.* " O'Reilly Media, Inc.", 2018.

[2] N. Khan, M. S. Alsaqer, H. Shah, G. Badshah, A. A. Abbasi, and S. Salehian, "The 10Vs, issues and challenges of big data," *Proceedings of the 2018 International Conference on Big Data and Education*, 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:44140950

[3] S. Salloum, R. Dautov, X. Chen, P. X. Peng, and J. Z. Huang, "Big data analytics on Apache Spark," *International Journal of Data Science and Analytics*, vol. 1, no. 3, pp. 145–164, Nov 2016. [Online]. Available: https://doi.org/10.1007/s41060-016-0027-9

[4] M. Guller, *Big Data Analytics with Spark: A Practitioner's Guide to Using Spark for Large Scale Data Analysis.* Apress, 2015. [Online]. Available: https://books.google.de/books?id=bNP8rQEACAAJ

[5] S. Özer, E. Ilhan, M. A. Özkanoglu, and H. A. Cirpan, "Offloading deep learning powered vision tasks from UAV to 5G edge server with denoising," *IEEE Transactions on Vehicular Technology*, 2023.

[6] S. Özer, M. Ege, and M. A. Özkanoglu, "SiameseFuse: A computationally efficient and a not-so-deep network to fuse visible and infrared images," *Pattern Recognition*, vol. 129, p. 108712, 2022.

[7] E. Ölmezogullari and I. Ari, "Online association rule mining over fast data," in *2013 IEEE International Congress on Big Data.* IEEE, 2013, pp. 110–117.

[8] L. Erhan, M. Ndubuaku, M. Di Mauro, W. Song, M. Chen, G. Fortino, O. Bagdasar, and A. Liotta, "Smart anomaly detection in sensor systems: A multi-perspective review," *Information Fusion*, vol. 67, pp. 64–79, 2021.

[9] A. C. Ikegwu, H. F. Nweke, C. V. Anikwe, U. R. Alo, and O. R. Okonkwo, "Big data analytics for data-driven industry: a review of data sources, tools, challenges, solutions, and research directions," *Cluster Computing*, vol. 25, no. 5, pp. 3343–3387, 2022.

[10] M. Assefi, E. Behravesh, G. Liu, and A. P. Tafti, "Big data machine learning using Apache Spark MLlib," in *2017 ieee international conference on big data (big data).* IEEE, 2017, pp. 3492–3498.

[3]https://www.tpc.org/tpcds/

[11] M. A. Khan, M. R. Karim, and Y. Kim, "A two-stage big data analytics framework with real world applications using spark machine learning and long short-term memory network," *Symmetry*, vol. 10, no. 10, p. 485, 2018.

[12] S. Alotaibi, R. Mehmood, I. Katib, O. Rana, and A. Albeshri, "Sehaa: A big data analytics tool for healthcare symptoms and diseases detection using Twitter, Apache Spark, and Machine Learning," *Applied Sciences*, vol. 10, no. 4, p. 1398, 2020.

[13] A. W. Services, "Amazon Elastic MapReduce (EMR)," *Amazon Web Services Documentation*, 2023. [Online]. Available: https://aws.amazon.com/emr/

[14] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark SQL: Relational Data Processing in Spark," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 1383–1394. [Online]. Available: https://doi.org/10.1145/2723372.2742797

[15] X. Meng, J. K. Bradley, B. Yavuz, E. R. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. B. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar, "MLlib: Machine Learning in Apache Spark," *CoRR*, vol. abs/1505.06807, 2015. [Online]. Available: http://arxiv.org/abs/1505.06807

[16] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache Hadoop YARN: yet another resource negotiator," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC '13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2523616.2523633

[17] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'11. USA: USENIX Association, 2011, p. 295–308.

[18] A. J. Awan, M. Brorsson, V. Vlassov, and E. Ayguadé, "How Data Volume Affects Spark Based Data Analytics on a Scale-up Server," *CoRR*, vol. abs/1507.08340, 2015. [Online]. Available: http://arxiv.org/abs/1507.08340

[19] E. R. Sparks, A. Talwalkar, M. J. Franklin, M. I. Jordan, and T. Kraska, "TuPAQ: An Efficient Planner for Large-scale Predictive Analytic Queries," *CoRR*, vol. abs/1502.00068, 2015. [Online]. Available: http://arxiv.org/abs/1502.00068

[20] E. R. Sparks, A. Talwalkar, D. Haas, M. J. Franklin, M. I. Jordan, and T. Kraska, "Automating model search for large scale machine learning," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, ser. SoCC '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 368–380. [Online]. Available: https://doi.org/10.1145/2806777.2806945

[21] K. Eng, A. Hindle, and E. Stroulia, "Patterns in Docker Compose Multi-Container Orchestration," 2023.

[22] S. Ryza, U. Laserson, S. Owen, and J. Wills, *Advanced Analytics with Spark: Patterns for Learning from Data at Scale*. O'Reilly Media, 2015. [Online]. Available: https://books.google.de/books?id=M0_GBwAAQBAJ

[23] N. Muhtaroglu, B. Kolcu, and I. Ari, "Testing performance of application containers in the cloud with HPC loads," in *Proceedings Of The Fifth International Conference On Parallel, Distributed, Grid And Cloud Computing For Engineering*. Civil-Comp, 2017.

[24] B. A. Baransel, A. Peker, H. O. Balkis, and I. Ari, "Towards low cost and smart load testing as a service using containers," in *Intelligent Technologies and Applications: Third International Conference, INTAP 2020, Grimstad, Norway, September 28–30, 2020, Revised Selected Papers 3*. Springer, 2021, pp. 292–302.

[25] H. K. Gupta and D. R. Parveen, "Comparative study of big data frameworks," *2019 International Conference on Issues and Challenges in Intelligent Computing Techniques (ICICT)*, vol. 1, pp. 1–4, 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:211057527

[26] S. Han, W. Choi, R. Muwafiq, and Y. Nah, "Impact of memory size on Bigdata processing based on Hadoop and Spark," *Proceedings of the International Conference on Research in Adaptive and Convergent Systems*, 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:31856220

[27] D. M. Nascimento, M. Ferreira, and M. L. Pardal, "Does Big Data Require Complex Systems? A Performance Comparison Between Spark and Unicage Shell Scripts," *ArXiv*, vol. abs/2212.13647, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:255186153

[28] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Anthony, H. Liu, and R. Murthy, "Hive - a petabyte scale data warehouse using Hadoop," *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, pp. 996–1005, 2010. [Online]. Available: https://api.semanticscholar.org/CorpusID:2972808

[29] K. V. Shvachko, H. Kuang, S. R. Radia, and R. J. Chansler, "The Hadoop Distributed File System," *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–10, 2010. [Online]. Available: https://api.semanticscholar.org/CorpusID:13925042