SparkCAD: Caching Anomalies Detector for Spark Applications

Hani Al-Sayeh
TU Ilmenau, Germany
hani-bassam.al-sayeh@tu-ilmenau.de

Muhammad Attahir Jibril
TU Ilmenau, Germany
muhammad-attahir.jibril@tu-ilmenau.de

Muhammad Waleed Bin Saeed
TU Ilmenau, Germany
muhammad-waleed.bin-saeed@tu-ilmenau.de

Kai-Uwe Sattler
TU Ilmenau, Germany
kus@tu-ilmenau.de

ABSTRACT

Developers of Apache Spark applications can accelerate their workloads by caching suitable intermediate results in memory and reusing them rather than recomputing them all over again every time they are needed. However, as scientific workflows are becoming more complex, application developers are becoming more prone to making wrong caching decisions, which we refer to as *caching anomalies*, that lead to poor performance. We present and give a demonstration of *Spark Caching Anomalies Detector* (*SparkCAD*), a developer decision support tool that visualizes the logical plan of Spark applications and detects caching anomalies.

PVLDB Reference Format:
Hani Al-Sayeh, Muhammad Attahir Jibril, Muhammad Waleed Bin Saeed, and Kai-Uwe Sattler. SparkCAD: Caching Anomalies Detector for Spark Applications. PVLDB, 15(12): 3694 - 3697, 2022.
doi:10.14778/3554821.3554877

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/dbis-ilm/SparkCAD.

1 INTRODUCTION

Apache Spark [20] maximizes performance efficiency for iterative workloads using large amounts of memory to cache frequently-used datasets rather than recomputing them in each iteration [18].

Typically, Spark application developers make *caching decisions* based on their knowledge of the application's data flow dependencies [10, 19]. However, applications are becoming more complex with a massive number of Resilient Distributed Datasets (RDDs §2) and the dependencies between them, resulting in gigantic data flows with plenty of interleaving forks and joins. Additionally, Spark autonomously persists intermediate results at some processing stages, (e.g., shuffled data blocks), which further complicates the caching decisions for the developers. Consequently, they become increasingly more prone to making wrong caching decisions that can lower the performance of their applications to 51.2 % [10].

These wrong caching decisions cause two types of *caching anomalies* in the application data flow. The first anomaly, which we term as *non-reused cached RDD*, occurs when the application developer caches an RDD that is not reused and that occupies space from already limited memory resources. As a result, other reused cached RDDs may be evicted, the free memory for execution will be reduced or, even worse, Out of Memory error might occur [10]. The second anomaly, which we refer to as *recomputed RDD*, takes place when the application developer does not cache an RDD that is reused multiple times, leading to significant recomputation overhead.

To see how frequently these caching anomalies occur, we studied 130 applications from machine learning libraries like Spark MLlib [11], graph analysis libraries like GraphFrames [13] (a library on top of Spark Graphx [16]), advanced Spark analytics [15], and synthetic Spark benchmarks [1, 7]. We realize that only 32 applications are free of caching anomalies. The remaining 98 applications have, in total, 1, 756 and 15, 554 non-reused cached RDD and recomputed RDD anomalies respectively. To show the impact of the caching anomalies on the overall application performance, we select the Principal Component Analysis (PCA) implementation of Spark MLlib to process 16.8 GB input dataset (generated by HiBench [3] with 'bigdata' scale) and run it on our 16-node Spark cluster. Each node is equipped with an Intel Core i5 CPU running at 4x 2.90 GHz, 16 GB RAM, 1 TB disk, and 1 GBit/s LAN and run Hadoop MapReduce 3.2.2, Spark 3.1.2, Java 8u102, Apache YARN, and HDFS. In the recomputed RDD anomaly in PCA, we realize that PCA iterates over an uncached RDD more than 600 times, thus taking 19 minutes to run. We update the source code of Spark MLlib by caching that particular RDD in memory, resulting in 9.8 minutes to process the same input dataset on the same cluster configuration.

Spark's History Server [2] reads execution logs and provides a web user interface (UI) that application developers can use to get more insights into the execution of their applications. This web UI displays the directed acyclic graph (DAG) of RDDs in each job (§2), individually, which gives a partial view on RDDs and their dependencies but does not provide a comprehensive overview on the whole application data flow, i.e., logical plan of RDDs and transformations across all jobs and stages in a single view. Thus, it is not reliable as a caching decision support tool for application developers when their programs become complex.

Some previous studies rank cached RDDs to select those to purge from memory in case of memory limitation by proposing cache eviction policies [14, 18]. Some adjust memory parameters to avoid cache eviction in advance [4, 9, 17]. Even though these solutions are effective, they still work based on the caching decisions of application developers. For example, these solutions and their likes will not improve the performance in the previously illustrated PCA scenario because the developers of Spark MLlib do not cache any of its RDDs. Other studies try to solve the problem by caching RDDs

Proceedings of the VLDB Endowment, Vol. 15, No. 12 ISSN 2150-8097.
doi:10.14778/3554821.3554877

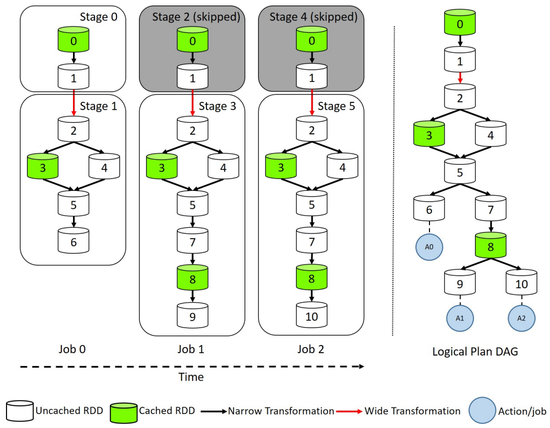

Figure 1: Logical Plan as a Single View of Applications.

on the fly [12] or detecting cache-related bugs [10]. These solutions (1) require instrumenting Spark's code for trace collection, which adds more complexity and performance overhead, (2) are generic for all applications without considering specific characteristics of each application, leading to sub-optimal solutions, and (3) do not give application developers the option to contribute to the caching decision based on their knowledge of the specific performance characteristics of their applications. For example, an application developer might opt for recomputing a reused RDD whose computation time is negligible rather than caching it in memory, especially if it is huge in size. We use the Latent Dirichlet Allocation (LDA) application to validate this. Similar to our experiment on PCA, Spark MLlib developers do not cache any RDD in LDA even though there is an RDD reused 20 times. We run LDA on 4.1 GB input dataset (generated by HiBench [3] with 'bigdata' scale) with the default implementation and after caching the reused RDD, we do not realize any impact of our update on the performance. This is because LDA is a CPU-intensive application and its performance bottleneck is in data processing rather than recomputing the reused RDD.

In this demonstration, we present *SparkCAD*, a tool that supports Spark application developers in writing complex programs, e.g., advanced analytics. Firstly, it visualizes the logical plan of the entire application in a single view with various display options. Secondly, it helps application developers to detect caching anomalies in the logical plan based on their criteria. Thirdly, as an interactive *what-if* analysis tool, it allows application developers to make new caching decisions and see the impact of their caching decisions without carrying out additional experiments. Lastly, it provides the developer with a sequence of recommended cache/unpersist commands, which we term *Recommended Schedule*, to help the developer to know when to cache or unpersist an RDD. In addition, it gives an overview of memory footprint during the application run.

2 EXECUTION MODEL OF SPARK

RDDs are the primary abstraction for distributed data processing in Spark [19]. A class of operations called *transformations* (e.g., map, filter) create new RDDs from existing ones. Another class of operations called *actions* (e.g., collect, count) return a value to a central process driver after running a computation over RDDs.

The *application* level is the highest level of computation in Spark and consists of one or more sequential *jobs*, each of which is triggered by an action. A job comprises a single action and a sequence of the transformations preceding it, represented by a *DAG* of transformations. When a transformation is applied on a (parent) RDD, a new (child) RDD is created. A transformation is either *narrow* or *wide*. Spark *stages* are created by splitting the DAG at shuffle boundaries (wide transformation), whereby the scheduler pipelines each group of narrow transformations into a stage.

Several jobs in the same application may have transformations in common. Figure 1 illustrates the merging of all the DAGs of jobs to have a single logical plan of the entire application. The computation of the RDDs can be traced in a depth-first traversal order, starting from RDD_0. Without caching, the number of times an RDD is computed is determined by the number of its child branches in the complete DAG. However, RDD_1 is computed once because it is followed by a wide transformation and Spark persists its shuffle blocks. $Stage_2$ and $Stage_4$ are therefore *skipped stages*. Even though RDD_8 is used twice, it is computed once because it is cached and since it is the only child of RDD_7, the latter is also computed once. RDD_5 is used to compute each of RDD_6 and RDD_7. RDD_5 is thus computed twice because it is not cached. Even though RDD_3 is cached, RDD_2 is computed every time RDD_5 is computed. This is because computing RDD_5 requires computing RDD_4, which is not cached and, in turn, computing RDD_4 requires computing RDD_2.

3 SPARKCAD

SparkCAD is a Python decision support tool for Spark application developers. As shown in Figure 2, *SparkCAD visualizes* the entire logical plan of an application and *detects* caching anomalies in three steps, namely, *parse*, *analyze* and *visualize*.

3.1 Parse

The log file that Spark's History Server reads to make displays via its web UI contains an ordered list of runtime events stored in JSON format. Even though Spark does not provide the size of each RDD and the execution time of each transformation, *SparkCAD* uses the log file without any additional metadata. *SparkCAD* selects three relevant events: (1) *SparkListenerApplicationStart*, from which it extracts the application name, (2) *SparkListenerJobStart* to get the ID and name of each job and information on each RDD in each job such as the list of its parent RDDs, its *callsite* (i.e., the location in the source code), whether it is cached or not, etc, and (3) *SparkListenerStageSubmitted* to obtain the set of actually executed stages (that are not skipped) like $Stage_0$, $Stage_1$, $Stage_3$, and $Stage_5$ in Figure 1. The extracted information is stored in a data structure we call *FactHub* that serves as the data source for later steps.

3.2 Analyze

Firstly, *SparkCAD* generates the set of transformations between RDDs based on their parent-child dependencies. It considers a dependency as a narrow transformation if the parent and child RDDs are in the same stage, and as a wide transformation otherwise. Secondly, it calculates the number of usage of each RDD by traversing each submitted stage starting from the last RDD therein and going backwards in a recursive fashion towards its root RDDs. Consider

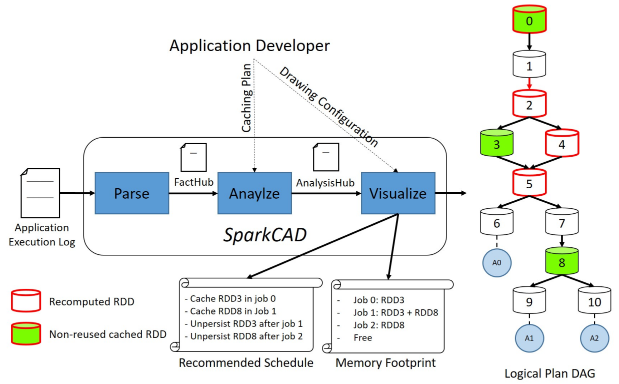

Figure 2: Overview of SparkCAD.

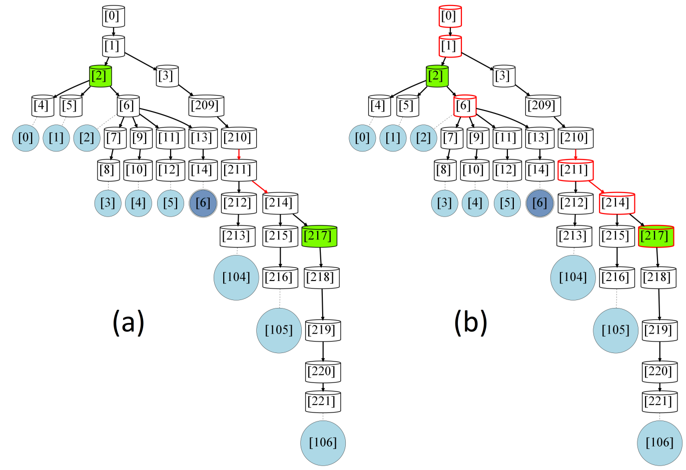

Figure 3: SparkCAD: Logical Plan Visualization and Caching Anomalies Detection of SVM Application in Spark MLlib.

Figure 1. A root RDD in a stage is one that has no parent RDD (RDD_0 in $Stage_0$), or is cached (RDD_8 in $Stage_5$), or whose parent RDD(s) is in another stage (RDD_2 in $Stage_1$). Note that even though RDD_8 is cached, it is not a root RDD in $Stage_3$ because it is computed for the first time in this stage. Thirdly, *SparkCAD* detects caching anomalies by identifying two cases. If an RDD is cached and the number of its usage is less than or equal to the *Computation Tolerance Threshold* (1 by default), then the case is considered as a non-reused cached RDD anomaly. If an RDD is not cached and the number of its usage is more than the threshold, then the case is considered as a recomputed RDD anomaly. By increasing the value of the Computation Tolerance Threshold, the cached RDDs in memory will be less. Users can determine this value based on their knowledge of the available memory. All the results of this step are stored in the *AnalysisHub* to be visualized in the next step. In the interactive *what-if* analysis session, the user can re-trigger the *analyze* step after changing the *caching plan* (set of cached RDDs). *SparkCAD* then recalculates the number of usage of each RDD and detects caching anomalies with regards to the new caching plan. While traversing RDDs, *SparkCAD* keeps track of the job and stage of the last usage of each RDD. This way, *SparkCAD* recommends when to unpersist an RDD, as part of the sequence of recommended cache and unpersist instructions, which we refer to as *Recommended Schedule*. In Figure 2, starting from Job_1, RDD_8 is a child of RDD_3 in all the remaining jobs. Therefore, the recommended schedule specifies unpersisting RDD_3 after caching RDD_8 in Job_1. *SparkCAD* displays the change in memory footprint with regards to each item in the recommended schedule (i.e., cache or unpersist) to let the user see the memory usage during the application run.

3.3 Visualize

SparkCAD uses Graphviz [6] to visualize the logical plan of an application as a DAG of nodes and edges, where the nodes are the RDDs and the edges are the transformations between them. With various drawing configurations in *SparkCAD*, a user can identify whether an RDD is cached or not, whether a transformation is narrow or wide, the occurrence of caching anomalies, etc. It is worth mentioning that even though *SparkCAD* is used for Spark applications, the concept behind it is applicable to any other dataflow processing system (e.g., Flink and Storm) by updating the parse step (3.1).

4 DEMONSTRATION

Figures 3 and 4 are sample screenshots of the visualization by *SparkCAD*. Using Jupyter notebook [8], a user can interactively run *SparkCAD* as demonstrated below:

Step 1: *Show me the logical plan of my application.* The user selects one of the 130 prepared Spark execution logs or uses other logs to see the logical plan of the application.

Step 2: *Show me a different view of my application.* The user changes the drawing parameters to improve the readability of the logical plan. In Figure 3a, instead of displaying the logical plan with hundreds of iterations (i.e., the repetitive lineage of RDDs and transformations), the user reduces the maximum number of drawn iterations to four and, as a result, *SparkCAD* does not display jobs/actions between Job_6 - Job_{104}.

Step 3: *Are there caching anomalies in the logical plan?* In Figure 3b, the user selects the option to highlight both caching anomaly types (i.e., non-reused cached RDD and recomputed RDD).

Step 4: *What happens if I cache/do not cache a certain RDD?* Firstly, the user defines the Computation Tolerance Threshold (three in Figure 4a). This means that *SparkCAD* does not highlight an RDD that is computed twice (e.g., RDD_1) as a recomputed RDD. To resolve caching anomalies, the user adds RDD_6 to the caching plan and removes RDD_{217} from it. As depicted in Figure 4b, RDD_2 becomes a non-reused cached RDD because it is cached and its number of usage is equal to the Computation Tolerance Threshold. As Figure 4c depicts, resolving this anomaly by removing RDD_2 from the caching plan leads to RDD_1 becoming a recomputed RDD because it would be computed four times in Job_0, Job_1 Job_2 and Job_{104}.

Step 5: *Which RDDs should be unpersisted and when? What is the memory footprint?* In Figure 4b, *SparkCAD* recommends unpersisting RDD_2 after Job_2 because in this job, RDD_6 is cached and it is a child of RDD_2 in all remaining jobs. This means that starting from Job_3, RDD_6 is used rather than RDD_2. Note that RDD_2 is not to be unpersisted in Job_2 because, due to the *lazy evaluation* of Spark, it will be recomputed in Job_2. The peak memory pressure could be analyzed and the user decides whether to keep the caching plan in

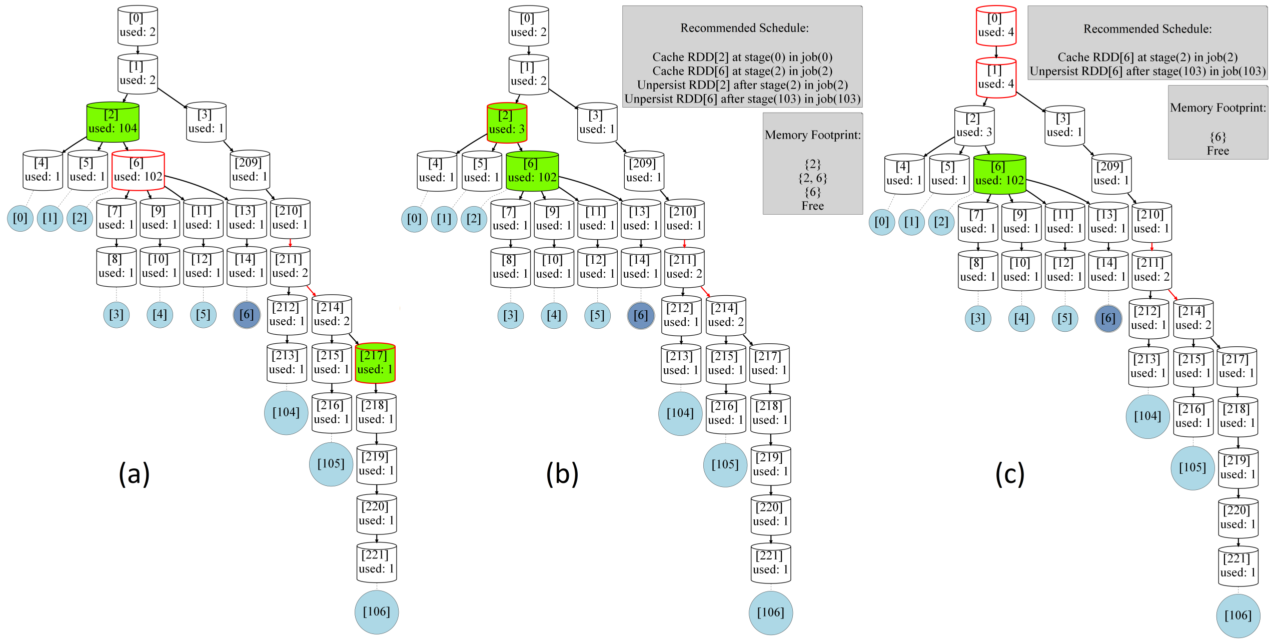

Figure 4: SparkCAD: Interactive What-if Analysis Session with on SVM Application in Spark MLlib.

Figure 4c by caching only RDD_6 or by adding RDD_1 to the caching plan. The users make these decisions based on their knowledge of the size of each RDD, the allocated memory and the computation overhead of transformations.

Step 6: *Show me the impact of my updates to the caching plan on the application performance.* In the end, the resulting recommended schedule could be applied using *Juggler Engine* [5], which is an instrumented version of Spark that accepts the recommended schedule as a configuration and overwrites the default caching plan.

ACKNOWLEDGMENTS

This research was partially funded by the Thuringian Ministry for Economy, Science and Digital Society under the project thurAI and by the Carl-Zeiss-Stiftung under the project MemWerk.

REFERENCES

[1] [n.d.]. Apache Spark Examples. https://github.com/apache/spark/tree/master/examples/src/main/scala/org/apache/spark/examples. Accessed: 2022-07-11.
[2] [n.d.]. Monitoring and Instrumentation Spark's Applications. https://spark.apache.org/docs/3.1.2/monitoring.html. Accessed: 2022-07-11.
[3] [n.d.]. The HiBench Suite. https://github.com/Intel-bigdata/HiBench/tree/v7.1.1. Accessed: 2022-07-11.
[4] Hani Al-Sayeh, Muhammad Attahir Jibril, Bunjamin Memishi, and Kai-Uwe Sattler. 2022. Blink: Lightweight Sample Runs for Cost Optimization of Big Data Applications. In *European Conference on Advances in Databases and Information Systems (ADBIS)*. Springer.
[5] Hani Al-Sayeh, Bunjamin Memishi, Muhammad Attahir Jibril, Marcus Paradies, and Kai-Uwe Sattler. 2022. Juggler: Autonomous Cost Optimization and Performance Prediction of Big Data Applications. In *Proceedings of the 2022 International Conference on Management of Data*. 1840–1854.
[6] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C North, and Gordon Woodhull. 2001. Graphviz—open source graph drawing tools. In *International Symposium on Graph Drawing*. Springer, 483–484.
[7] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. 2010. The HiBench benchmark suite: Characterization of the MapReduce-based data analysis. In *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*. IEEE, 41–51.
[8] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B Hamrick, Jason Grout, Sylvain Corlay, et al. 2016. *Jupyter Notebooks-a publishing format for reproducible computational workflows*. Vol. 2016.
[9] Mayuresh Kunjir and Shivnath Babu. 2020. Black or White? How to develop an autotuner for memory-based analytics. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1667–1683.
[10] Hui Li, Dong Wang, Tianze Huang, Yu Gao, Wensheng Dou, Lijie Xu, Wei Wang, Jun Wei, and Hua Zhong. 2020. Detecting cache-related bugs in Spark applications. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 363–375.
[11] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. 2016. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research* 17, 1 (2016), 1235–1241.
[12] Michael J Mior and Kenneth Salem. 2020. ReSpark: Automatic Caching for Iterative Applications in Apache Spark. In *2020 IEEE International Conference on Big Data (Big Data)*. IEEE, 331–340.
[13] Mark Needham and Amy E Hodler. 2019. *Graph algorithms: practical examples in Apache Spark and Neo4j*. O'Reilly Media.
[14] Tiago BG Perez, Xiaobo Zhou, and Dazhao Cheng. 2018. Reference-distance eviction and prefetching for cache management in spark. In *Proceedings of the 47th International Conference on Parallel Processing*. 1–10.
[15] Sandy Ryza, Uri Laserson, Sean Owen, and Josh Wills. 2017. *Advanced analytics with spark: patterns for learning from data at scale*. " O'Reilly Media, Inc.".
[16] Reynold S Xin, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. 2013. Graphx: A resilient distributed graph system on spark. In *First international workshop on graph data management experiences and systems*. 1–6.
[17] Luna Xu, Min Li, Li Zhang, Ali R Butt, Yandong Wang, and Zane Zhenhua Hu. 2016. Memtune: Dynamic memory management for in-memory data analytic platforms. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 383–392.
[18] Yinghao Yu, Wei Wang, Jun Zhang, and Khaled Ben Letaief. 2017. LRC: Dependency-aware cache management for data analytics clusters. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*. IEEE, 1–9.
[19] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. 15–28.
[20] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10)*.