Importance of Memory Management Layer in Big Data Architecture

Maha Dessokey[1], Sherif M. Saif[2], Hesham Eldeeb[3]
Computer and Systems Department
Electronics Research Institute
Cairo, Egypt

Sameh Salem[4], Elsayed Saad[5]
Computer and Systems Department
Faculty of Engineering, Helwan University
Cairo, Egypt

***Abstract*—The generation of daily massive amounts of heterogeneous data from a variety of sources presents a challenge in terms of storage and analysis capabilities and brings new problems into high-performance computing clusters. To better utilize this huge and heterogeneous data, the continuous development of advanced Big Data platforms and Big Data analytic techniques are required. One of the significant issues with in-memory Big Data processing platforms, such as Apache Spark, is the user's responsibility to decide whether the intermediate data should be cached or not. In addition, the data may be kept in several storage systems and physically scattered over different racks, regions, and clouds. Data need to be close to the computation nodes and hence data locality issue is a challenge. In this paper, using a distinct memory management layer between the data processing layer and the data storage layer, which automatically caches data without the need for any interaction from the applications' developers, is evaluated. K-means, PageRank and WordCount workloads from the HiBench benchmark beside a real case to predict the price of Real Estate that is implemented using Gradient Boosting Regression Tree model, are used to evaluate this framework. Experiments show that the memory management layer outperforms the Apache Spark in reducing the execution time.**

***Keywords*—Apache Spark; Big Data; data analytics algorithms; memory management**

I. Introduction

For both academic, business and engineering communities, Big Data analytics for storing, processing, and analysing large scale heterogeneous datasets has become a must-have tool. New Big Data analysis techniques [1], as well as the constant development of advanced Big Data platforms [2], are required to take full advantage of this massive and heterogeneous data. Fig. 1 shows the Big Data architecture. The architecture consists of four layers.

Data storage layer, which contains multiple storage systems such as distributed file systems Hadoop Distributed File System, GlusterFS, Ceph, etc, or remote access file systems such as Amazon S3, Swift, Google Cloud Storage, etc. and it can contain tools and techniques such as relational databases and NoSQL tools.

Resource management layer controls resource management, scheduling, and security. Examples of resource managers are YARN, Mesos, and Kubernetes K8s.

Data processing layer, which contains one of the Big Data platforms such as the open-source Hadoop Map Reduce [3], the Apache Spark platform [4], The Apache Flink, etc.

Application layer, the top layer, can contain any application type, such as batch, graph analytics, machine learning, streaming, etc.

The default Big Data architecture has many challenges [5][6], such as:

- In some Big Data platforms that support in-memory computing, such as Apache Spark and Apache Flink, developers can cache data that will most likely be reused. As a result, it is entirely up to the developer to decide which data to cache in memory. Determining which data should be cached is a difficult task when dealing with jobs that consist of operations with complex dependencies. When there isn't enough memory, caching all data in memory will result in a significant performance loss. While disc caching saves RAM, it reduces the efficiency of in-memory computing;
- Data locality is another problem for data processing; data are physically scattered over different racks, regions, and clouds. Data must be near to where data computation occurs to be processed;
- The data needed by the application may be stored in different storage systems, the application developer must be aware of all Storage APIs, such as HDFS API, FUSE API, S3 API, REST API.

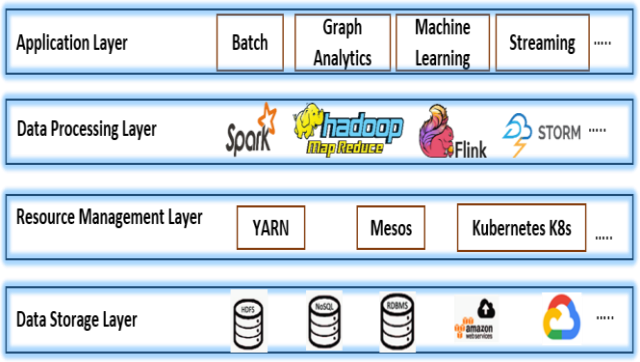

Fig. 1. The Big Data Architecture.

All the previous challenges necessitate the use of a distinct memory management layer between the data processing layer and the data storage layer.

In this paper, to evaluate the distinct memory management layer, the Apache Spark platform is used in the processing layer because Apache Spark boosts Hadoop's performance by up to 100x using in-memory cluster computing [7] and it is widely used in a variety of application domains, including bioinformatics [8], image processing [9], deep learning [10], finance [11], and astronomy [12], etc. The next section gives a background about Apache Spark and memory management layer. Three workloads from HiBench [13] are used for evaluation with randomly generated dataset, then a real case study to predict the price of real estate with real data set is developed to be used in the evaluation.

The rest of the paper is organized as follows. In Section 2 an overview about Apache Spark and memory management layer is given. In Section 3, a review of the related work is presented. In Section 4, the experimental setup and workloads are de-scribed. In Section 5, our experimental results are discussed. Finally, the conclusion of our findings is in Section 6.

II. Background

In this section, an overview is given on Apache Spark which is used in data processing layer, Apache Spark Standalone cluster manager which is used in resource management layer, Hadoop Distributed File System which is used in data storage layer and the memory management layer.

A. Apache Spark

Apache Spark [14] is a computing engine and a suite of libraries for processing data in parallel on computer clusters. Apache Spark is one of the most used open-source engines for Big Data processing. Apache Spark is compatible with several popular programming languages (Python, R, Scala, and Java). It offers libraries for a wide range of operations, including data loading and SQL queries, as well as machine learning and streaming computation. The basic abstraction in Apache Spark is a Resilient Distributed Dataset (RDD). It acts as an immutable, partitioned collection of elements. These elements can be run in parallel.

The Apache Spark cluster can be managed by Apache Spark Standalone cluster manager or by other cluster managers as Mesos or YARN. The Apache Spark Standalone cluster manager is used to test Apache Spark performance in our experimental setting. As shown in Fig. 2, the Apache Spark master receives the application then a driver process is created and connected to the cluster manager through the SparkContext [4]. The cluster manager allocates the Apache Spark workers. Each worker contains an executor processes. The driver process is in charge of running the main () function, keeping track of the Apache Spark application's progress, responding to a user's program input, and analyzing, distributing, and scheduling work throughout the executors. The executors are in charge of completing the job that has been given to them and reporting the state of the computation back to the driver node.

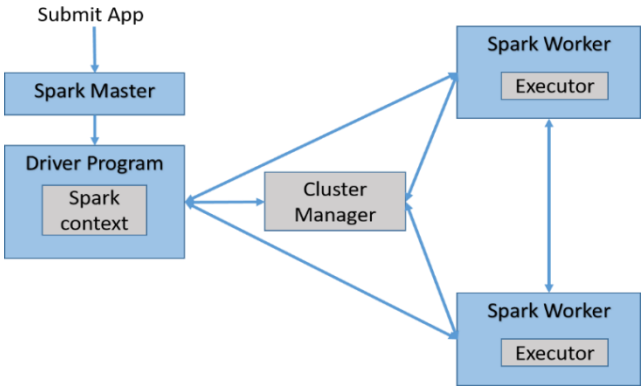

Fig. 2. Apache Spark Cluster Architecture.

Apache Spark was created to read and write data from and to Hadoop Distributed File System (HDFS) [15], allowing it to be used with Hadoop clusters. HDFS as a distributed file system allows users to access application data quickly. It allows enormous amounts of structured and unstructured data to be managed. HDFS is a file system that divides the processing of massive data sets across inexpensive hardware clusters.

HDFS has a primary NameNode, which keeps track of where the file is kept in the cluster and multiple of DataNodes on a commodity hardware cluster. Splitting huge files into little sections known as blocks is one of the major HDFS characteristics. These blocks hold a specific amount of data that can be read and written. The block size is set to 128 MB by default. Hadoop splits up blocks and distributes them across different nodes called DataNodes.

Another main characteristic in HDFS is replication which duplicates data blocks to provide fault tolerance and allows an application to select the number of replicas for a file. The replication factor can be specified when the file is created and can be changed later. If a node fails, you can still access the data on other nodes in the HDFS cluster that have a copy of the same data. By default, HDFS duplicates blocks three times.

B. Memory Management Layer

The challenges in the main Big Data architecture, as demonstrated in the introduction, necessitate the use of a distinct memory management layer between the data processing layer and the data storage layer. Fig. 3 shows the distinct memory management layer's location in the Big Data architecture.

This distinct memory management layer:

- Caches the most frequently used data automatically. A local storage space is set aside in each node of the processing cluster for hot and transient data. This storage could be of any type (memory, SSD or HDD). The size and type of storage are determined by the user. When an application attempts to read data that is only available in shared storage, the data are duplicated in local storage. When the local storage is full, one of the eviction policies [5] can be used to determine which data should be deleted.

- Handles the distributed storage system; because the data are duplicated in local storage, this can address the data locality issue in distributed storage systems.
- Serves as a global namespace, a global namespace is an important feature of a distributed file system that makes it easy to find and access data from multiple storage systems using a single control and administration layer. The global namespace can be considered as a global file directory that allows all data from several storage systems to seem as if they were stored in a single storage system. It automatically converts the standard client-side interface to any storage interface.

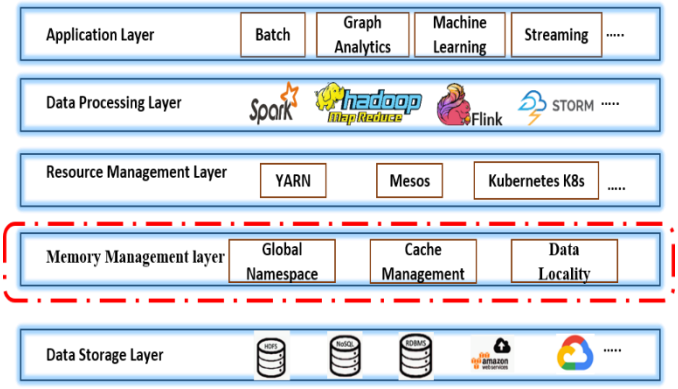

Fig. 3. The Big Data Architecture with a Distinct Memory Management Layer.

III. Related Work

Many optimization strategies have been presented to improve memory management in Big Data frameworks. For Apache Spark, when the memory used to cache data reaches its capacity limit, data must be selected to be deleted to make way for new ones. Apache Spark's cache replacement strategy uses Least Recently Used (LRU) criteria to determine which RDDs should be replaced, different cache replacement techniques were investigated [5] to improve Apache Spark performance. Apache Ignite [16], a high-performance, distributed in-memory computing platform for large-scale data sets has a cache management feature that keeps data in RAM as much as possible, having minimal interaction with the disk, but researchers in [17] observed that Apache Spark outperform Apache Ignite as Apache Ignite does not distribute well data among available nodes and it does not balance well the communication between the nodes. Other researchers are interested in investigating Apache Spark's performance on various disk types. Doppio [18] proposed an I/O Aware performance study for Apache Spark, which measured the I/O impact of using hard disk drives (HDDs) and Solid-state drives (SSDs) with different combinations, and discovered the relationship between computation and I/O access by changing the CPU core number. As a result, the model could be used to locate the best configuration on the public cloud. Instead of reserving running memory for caching, RubiX [19], an open source project employs SSDs. It is utilized in the Azure HDInsight data caching service, which increases the performance of Apache Spark processes [20]. Delta Lake [21] by Databricks is another open source storage layer. Which also leverages nodes' local storage for caching, and the data is cached automatically anytime a file has to be retrieved from a remote site, resulting in much faster reading speeds. The user is not required to take any action throughout the caching process. Open Cache Acceleration Software [22], works with node memory to build a multilevel cache that optimizes system memory usage and automatically chooses the appropriate cache level for active data, allowing programmes to run quicker than they would on SSDs alone.

IV. Experimental Setup

In this section, the used Big Data architecture is shown in Fig. 4, and fully described it in the following subsection then the used workloads and the implemented application are described.

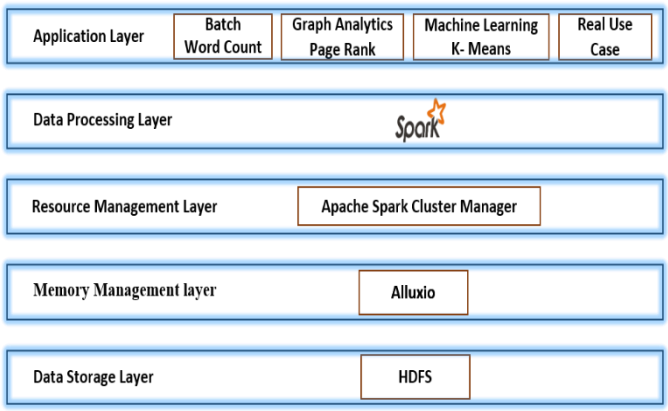

Fig. 4. Implemented Cluster's Architecture.

A. Cluster's Architecture

The experiments were deployed in a cluster at Electronics Research Institute. The Apache Spark cluster contains five servers, which is configured as one master and four slaves. There are 160 CPU cores and 640 GB of RAM in the cluster. The cluster services are shown in Fig. 5, and its specifications are listed in Table I. The data are stored on HDFS with data nodes on the same slaves.

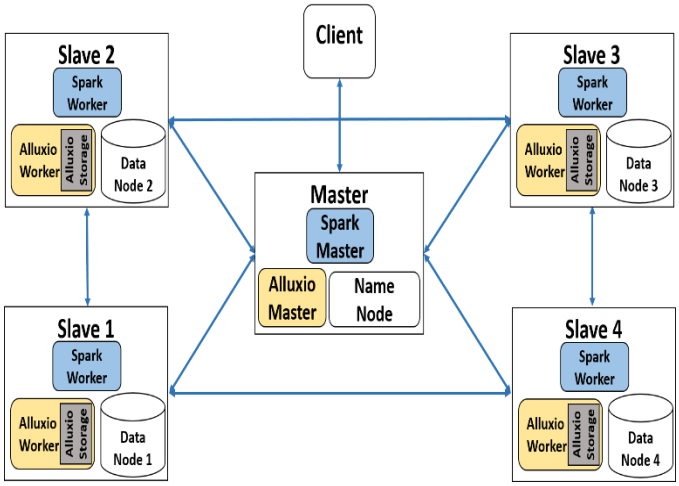

Fig. 5. Cluster's Services.

TABLE I. CLUSTER'S SPECIFICATIONS

Servers Configuration	Processor: Intel(R) Xeon(R) CPU E5-2680 0 @ 2.70GHz Main memory:128MB Local storage:1 TB CPU cores: 32
Software	Operating system: Red Hat 4.8.3-9 JDK: 1.8 Hadoop:2.7.1 Spark: 2.4.6 Alluxio: 2.3.0
Workload	HiBench 7.1.1

In the experimental setup, Alluxio [23] a virtual distributed storage platform, is used in the memory management layer. The master and workers of Alluxio are running on the same Apache Spark cluster.

Alluxio enables users to integrate their data across multiple platforms. As shown in Fig. 6, Alluxio is made up of three different components: masters, workers, and clients. All user requests and file system metadata modifications are served by the Alluxio Master. The Alluxio Job Master is a lightweight scheduler for file system activities that are then executed on Alluxio Job Workers.

A specific amount of local Alluxio storage is determined for each Alluxio worker to store hot and transient data. Client requests to read or write data are fulfilled by Alluxio workers by reading or constructing new blocks within their local resources.

There are three scenarios for reading data:

- If the requested data are already in the worker local storage, then the client will read the file directly via the local file system (Local Cache Hit).
- If requested data are stored in another worker, the client will perform a remote read from that worker that does have the data. After the client finishes reading the data, it creates a copy locally for future reads (Remote Cache Hit).
- If the data are not available in any of the running workers, the client will read the data from the storage (Cache Miss).

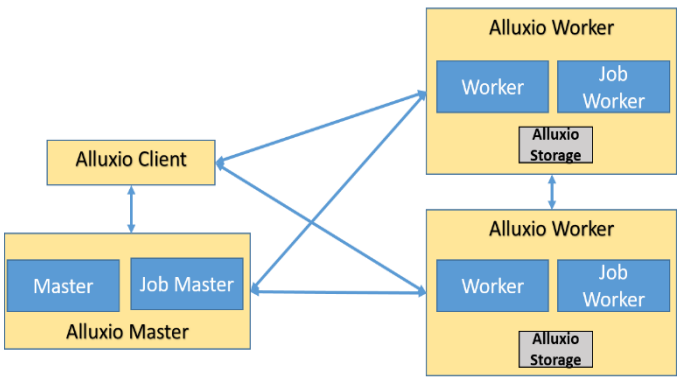

Fig. 6. Alluxio Architecture.

Workers are only in charge of managing blocks; the master is the only one who keeps track of the file-to-block mapping. Because RAM has a finite capacity, blocks in a worker may be evicted if space is full. Eviction policies as least recently used and least frequently used can be used by workers to pick which data to keep in the Alluxio space.

In our experiments some parameters must be set in order to make the Apache Spark applications access Alluxio, the Alluxio client jar must be in the classpath of all Apache Spark drivers and executors so spark.driver.extraClassPath and spark.executor.extraClassPath parameters were added to spark/conf/spark-defaults.conf and was set to the path where the alluxio-2.3.0-client.jar is located. In order for HiBench to access Alluxio, hibench.hdfs.master parameter had been set to alluxio://{Alluxio_master_Hostname}:19998 in Hibench/conf/hadoop.conf, and the Hibench/sparkbench/assemply/target/sparkbench-assemply.jar had been copied to /spark/jars.

The following configuration had been added to hadoop/coresite.xml

```
<configuration>
 <property>
  <name>fs.alluxio.impl</name>
  <value>alluxio.hadoop.FileSystem</value>
 </property>
</configuration>
```

The experiments compared between Apache Spark framework with HDFS and Apache Spark framework with a distinct memory management layer and with different RAM size per worker.

B. Workloads

For a comprehensive evaluation, first three applications from HiBench, including K-Means as a machine learning algorithm for clustering workload, Page Rank as a graph analytics workload and WordCount as a batch processing workload were used with randomly generated dataset. Those workloads were chosen to compare Apache Spark cluster performance with and without memory management layer because of their distinct properties. Then a real use case was implemented to predict real estate sale price using Gradient-Boosted Trees as a machine learning algorithm for regression.

1) K-Means: K-Means [24] is a popular unsupervised machine learning clustering algorithm for data mining and knowledge discovery. The idea of the algorithm as shown in Fig. 7 that it divides a collection of samples into K groups or clusters, In Fig. 7, K is equal to 3. In the initialization, the algorithm defines K centroids randomly, and makes iterations of calculations to define new centroids in order to minimize the Euclidean distances between the points forming each cluster and its centroid. The iterations are repeated until the most optimum centroids are defined or the maximum number of iterations is reached. The input data in our experiment were 100,000,000 samples generated by GenKMeans dataset based on uniform distribution and Gaussian distribution. Based on

each sample's attributes, the algorithm assigns each sample to one of the k groups iteratively. The algorithm's input parameters used in the experiments were 5 clusters, 20 dimensions, and 5 iterations.

2) PageRank: PageRank [25] is an iterative graph analytics algorithm, as shown in Fig. 8, it ranks items based on the number and quality of their links. The PageRank's mathematics is completely general and may be used to any graph or network in any domain. As a result, PageRank and its variations are widely used in social and information network analysis, link prediction, and recommendation systems. It's even used in road network systems analysis [26]. The PageRank algorithm used in our experiments is implemented in apache Spark MLLib. The input data were generated from web data whose hyperlinks follow the Zipfian distribution. In the experiments, the input data parameters used were 5,000,000 pages and 3 iterations.

3) WordCount: The WordCount workload is a batch processing workload. It scans the input data once and counts how many times each word appears.

Random text writer generates the input data, the input data size used in the experiments was 30 GB.

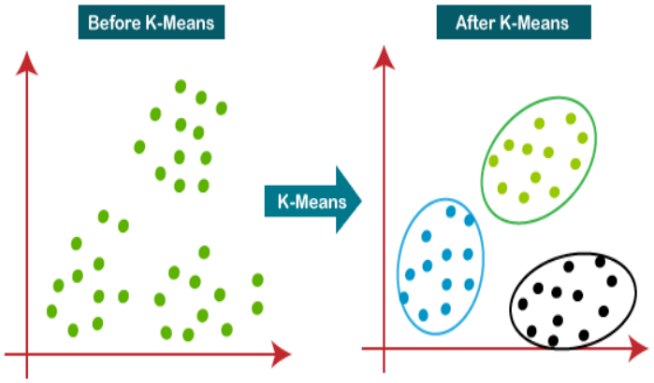

Fig. 7. K-Means Algorithm.

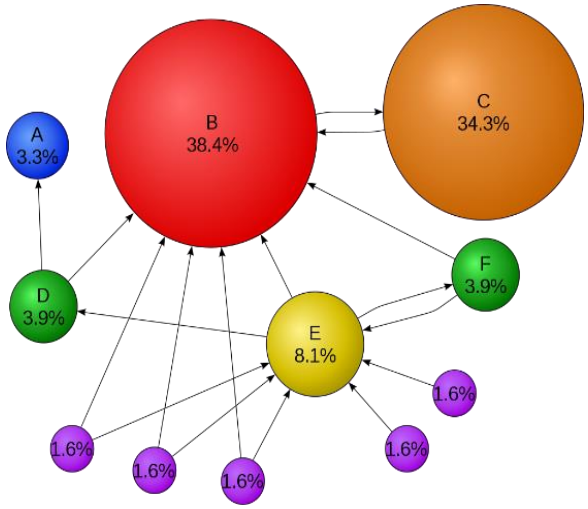

Fig. 8. Page Rank Algorithm.

4) Real estate sale price prediction: The prediction of real estate sale price in the presence of a large number of variables is a known problem, there are many models that were built with different methods to estimated house price by inputting house features [27] [28]. The data set used in this experiment has been downloaded from [29] and was collected from some popular portals for the sale of real estate in the period from 2018 to 2021. The dataset contains 5,477,005 records with 11 features. Fig. 9 presents the output of Python code to describe the dataset fields.

Where building type parameter could be { "1" for Panel, "2" for Monolithic, "3" for Brick, "4" for Blocky, "5" for Wooden and 0 for other type}. Object type indicates the apartment type and it could be {"1" for old buildings and "2" for new building}. Level indicates apartment floor. Levels indicates the number of stores in the building. Rooms indicates the number of living rooms and if the value is "-1" then it is for studio apartment. In our implementation, the data were read from HDFS once and once from Alluxio system. The code was written in Python. The dataset first passed through cleaning phase to remove any null values or negative values in the price column and take the log value of the price. Then the correlation matrix shown in Fig. 10 has been calculated to pick the most correlated features with the price.

```
df.describe()
```

	price	geo_lat	geo_lon	region	building_type	level	levels	rooms	area	kitchen_area	object_type
count	5.477006e+06	5.477006e+06	5.477006e+06	5.477006e+06	5.477006e+06	5.477006e+06	5.477006e+06	5.477006e+06	5.477006e+06	5.477006e+06	5.477006e+06
mean	4.422029e+06	5.403826e+01	5.324433e+01	4.307141e+03	1.948966e+00	6.214530e+00	1.139892e+01	1.726173e+00	5.391825e+01	1.062840e+01	3.945399e+00
std	2.150752e+07	4.622758e+00	2.074763e+01	3.308050e+03	1.038537e+00	4.957419e+00	6.535734e+00	1.082133e+00	3.335293e+01	9.792380e+00	4.558357e+00
min	-2.144967e+09	4.145906e+01	1.989020e+01	3.000000e+00	0.000000e+00	1.000000e+00	1.000000e+00	-2.000000e+00	7.000000e-02	1.000000e-02	1.000000e+00
25%	1.950000e+06	5.337768e+01	3.777790e+01	2.661000e+03	1.000000e+00	2.000000e+00	5.000000e+00	1.000000e+00	3.800000e+01	7.000000e+00	1.000000e+00
50%	2.990000e+06	5.517139e+01	4.306774e+01	2.922000e+03	2.000000e+00	5.000000e+00	1.000000e+01	2.000000e+00	4.802000e+01	9.700000e+00	1.000000e+00
75%	4.802000e+06	5.622613e+01	6.564895e+01	6.171000e+03	3.000000e+00	9.000000e+00	1.600000e+01	2.000000e+00	6.313000e+01	1.270000e+01	1.100000e+01
max	2.147484e+09	7.198040e+01	1.625361e+02	6.188800e+04	5.000000e+00	3.900000e+01	3.900000e+01	1.000000e+01	7.856000e+03	9.999000e+03	1.100000e+01

Fig. 9. Real Estate Data Set Parameters.

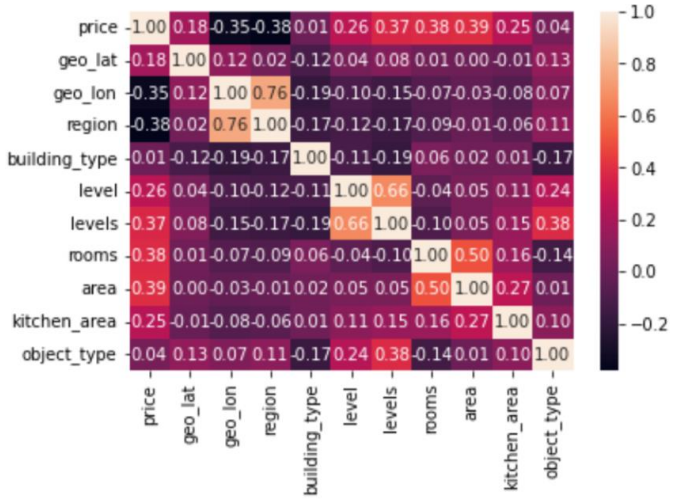

Fig. 10. Correlation Matrix.

The data set was split randomly into training set and testing set with the ratio 70:30 respectively. Gradient-Boosted Trees (GBTs) learning algorithm [30] was used to predict the price. GBTs is one of the most powerful and frequently used algorithms by data scientists for building predictive models [31]. It acts as a machine learning technique for regression and classification problems. In our application, the algorithm was used as a regression technique using pyspark.ml.regression library. Given the input variables, regression analysis estimates the conditional expectation of the price variable. The results shown in the next section were for 5 iterations, 10 max depth. To evaluate the prediction model performance, the Root Mean Squared Error (RMSE) evaluation measure was used which considers the sample standard deviation difference between the predicted and real values. The lower the RMSE, the more accurate the model predictions will be. The average RMSE in all the runs was 0.13.

V. Result and Discussion

In this section, the results obtained after running the experiments are shown and evaluated. The execution time in minutes is used to calculate performance measures. In the experiment, the difference between utilizing the Apache Spark with HDFS and Apache Spark using a distinct memory management layer with Alluxio was evaluated.

The results illustrated how Apache Spark with Alluxio in the memory management layer improves the performance in all cases. Fig. 11 shows the execution time of running Real Estate application, K-means, PageRank, and WordCount algorithms on ERI cluster. The worker's RAM size varies between 4, 8, 16, 32 and 64 GB to study the effect of the memory management layer with respect to the RAM size.

It can be seen from the results that the memory management layer has better performance with smaller RAM size.

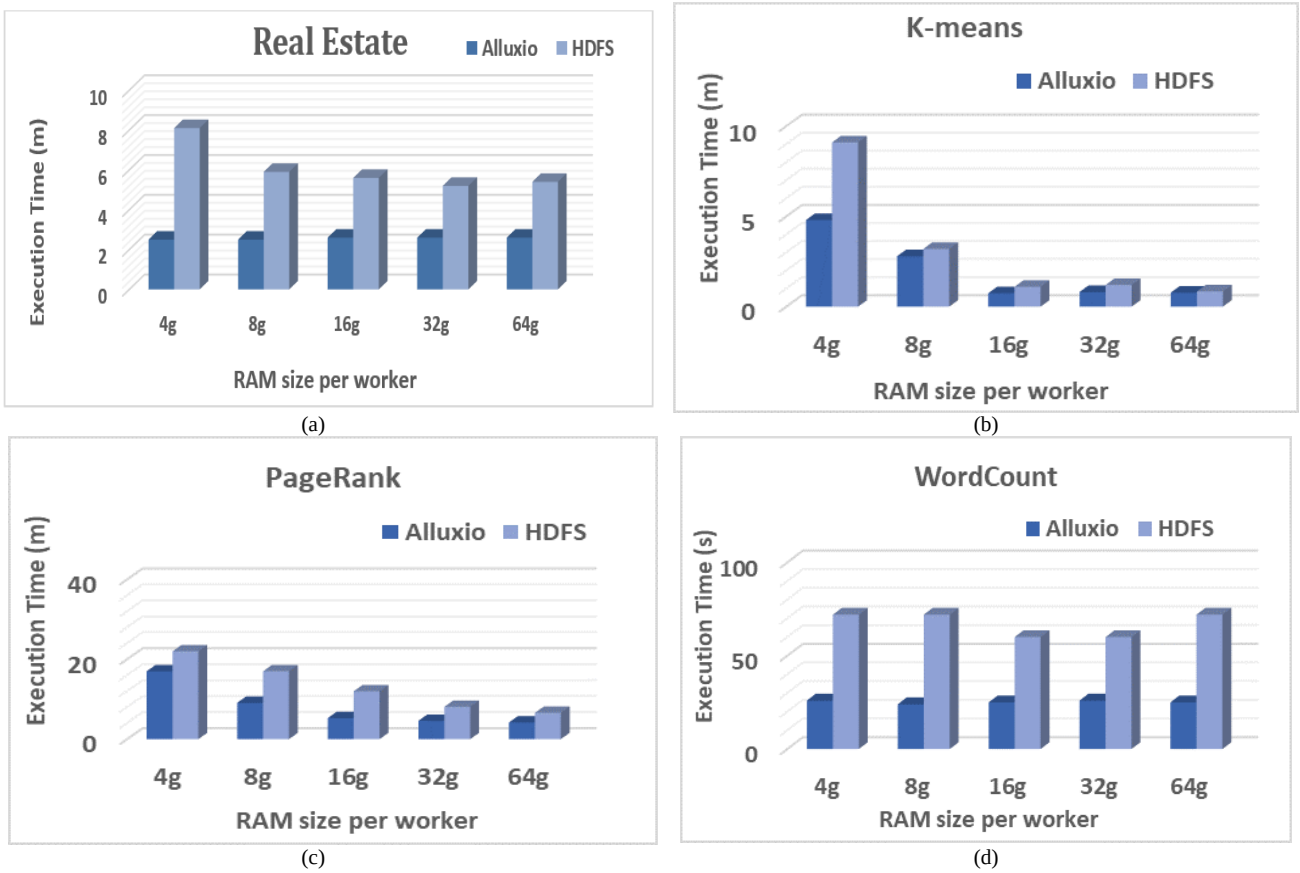

Fig. 11. The Execution Time of Running (a) Real Estate, (b) K-means; (c) PageRank and (d) WordCount on Apache Spark with HDFS and Apache Spark using Alluxio.

VI. Conclusion

In this paper, the impact of using a distinct memory management layer between the data processing layer and the data storage layer in Big Data Architecture was evaluated. This layer automatically caches the most frequently used data and handles the distributed storage system without the need for any interaction from the applications' developers.

First a HiBench benchmark was used with k-means, PageRank and WordCount algorithms with randomly generated workloads. Then a real case study with real dataset was implemented to predict real estate price using Gradient Boosting Regression tree. The results showed that when using distinct memory management layer, the execution time of the real estate application is up to three times faster than the normal case. So using memory management layer helps the applications' developers to get better performance up to three times faster with less effort. As a future work, other evaluations can be done with other tools, mentioned in section 3, other than Alluxio, such as Apache Ignite and RubiX.

Acknowledgments

The authors send their acknowledgement to the Electronics Research Institute (ERI), Cairo, Egypt for running the experiments on ERI system.

References

[1] M. S. Mahmud, J. Z. Huang, S. Salloum, T. Z. Emara, and K. Sadatdiynov, "A survey of data partitioning and sampling methods to support big data analysis," Big Data Mining and Analytics, vol. 3, no. 2, Art. no. 2, 2020.

[2] A. H. Ali, "A survey on vertical and horizontal scaling platforms for big data analytics," International Journal of Integrated Engineering, vol. 11, no. 6, Art. no. 6, 2019.

[3] "Apache Hadoop." https://hadoop.apache.org/.

[4] "Apache Spark." https://spark.apache.org/.

[5] M. Dessokey, S. M. Saif, S. Salem, E. Saad, and H. Eldeeb, "Memory Management Approaches in Apache Spark: A Review," Proceedings of the International Conference on Advanced Intelligent Systems and Informatics 2020, Cham, 2021, pp. 394–403.

[6] S. Lee, J. -Y. Jo and Y. Kim, "Survey of Data Locality in Apache Hadoop," IEEE International Conference on Big Data, Cloud Computing, Data Science & Engineering (BCD), 2019, pp. 46-53.

[7] N. Ahmed, A. L. C. Barczak, T. Susnjak, and M. A. Rashid, "A comprehensive performance analysis of Apache Hadoop and Apache Spark for large scale data sets using HiBench," Journal of Big Data, vol. 7, no. 1, Art. no. 1, Dec. 2020.

[8] Y. K. Gupta and S. Kumari, "Performance Evaluation of Distributed Machine Learning for Cardiovascular Disease Prediction in Spark," 5th International Conference on Trends in Electronics and Informatics (ICOEI), 2021, pp. 1506–1512.

[9] G.-M. Park, Y. S. Heo, and H.-Y. Kwon, "Trade-Off Analysis Between Parallelism and Accuracy of SLIC on Apache Spark," IEEE International Conference on Big Data and Smart Computing (BigComp), 2021, pp. 5–12.

[10] M. Haggag, M. M. Tantawy, and M. M. El-Soudani, "Implementing a deep learning model for intrusion detection on apache spark platform," IEEE Access, vol. 8, 2020, pp. 163660–163672.

[11] H. Sayed, M. A. Abdel-Fattah, and S. Kholief, "Predicting Potential Banking Customer Churn using Apache Spark ML and MLlib Packages: A Comparative Study," International Journal of Advanced Computer Science and Applications, vol. 9, no. 11, 2018.

[12] A. M. Mickaelian, "Big Data in Astronomy: Surveys, Catalogs, Databases and Archives," Communications of the Byurakan Astrophysical Observatory, vol. 67, pp. 159–180, 2020.

[13] "HIBench." https://github.com/Intel-bigdata/HiBench.

[14] M. Zaharia et al., "Apache spark: a unified engine for big data processing," Communications of the ACM, vol. 59, no. 11, Art. no. 11, 2016.

[15] D. Borthakur, "HDFS architecture," Document on Hadoop Wiki. URL http://hadoop. apache. org/common/docs/r0, vol. 20, 2010.

[16] "Apache Ignite." https://ignite.apache.org/.

[17] C. Stan, A. Pandelica, V. Zamfir, R. Stan, and C. Negru, "Apache Spark and Apache Ignite Performance Analysis," 22nd International Conference on Control Systems and Computer Science (CSCS), May 2019, pp. 726–733.

[18] P. Zhou, Z. Ruan, Z. Fang, M. Shand, D. Roazen, and J. Cong, "Doppio: I/O-aware performance analysis, modeling and optimization for in-memory computing framework," IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2018, pp. 22–32.

[19] "RubiX." https://github.com/qubole/rubix.

[20] "Azure HDIn-sight." https://docs.microsoft.com/en-us/azure/hdinsight/spark/apache-spark-improve-performance-iocache.

[21] "Databricks Delta Lake." [Online]. Available: https://docs.databricks.com/delta/optimizations/delta-cache.html.

[22] "Open Cache Acceleration." https://open-cas.github.io/.

[23] "Alluxio." https://www.alluxio.io/.

[24] J. Pérez-Ortega, N. N. Almanza-Ortega, A. Vega-Villalobos, R. Pazos-Rangel, C. Zavala-Díaz, and A. Martínez-Rebollar, "The k-means algorithm evolution," Introduction to Data Science and Machine Learning, IntechOpen, 2019.

[25] D. F. Gleich, "PageRank beyond the web," Siam Review, vol. 57, no. 3, Art. no. 3, 2015.

[26] J. Liu, X. Li, and J. Dong, "A survey on network node ranking algorithms: Representative methods, extensions, and applications," Science China Technological Sciences, vol. 64, no. 3, Art. no. 3, 2021.

[27] S. Jamil, T. Mohd, S. Masrom, and N. Ab Rahim, "Machine Learning Price Prediction on Green Building Prices," IEEE Symposium on Industrial Electronics Applications (ISIEA), 2020, pp. 1–6.

[28] N. H. Zulkifley, S. A. rahman, N. H. Ubaidullah, and I. Ibrahim, "House Price Prediction using a Machine Learning Model: A Survey of Literature," International Journal of Modern Education and Computer Science, vol. 12, 2020, pp. 46–54.

[29] Daniilak, Russia Real Estate 2018-2021. 2021. [Online]. Available: https://www.kaggle.com/mrdaniilak/russia-real-estate-2018-2021.

[30] C. Krauss, X. A. Do, and N. Huck, "Deep neural networks, gradient-boosted trees, random forests: Statistical arbitrage on the S&P 500," European Journal of Operational Research, vol. 259, no. 2, Jun. 2017, pp. 689–702.

[31] A. D. Lainder and R. D. Wolfinger, "Forecasting with gradient boosted trees: augmentation, tuning, and cross-validation strategies: Winning solution to the M5 Uncertainty competition," International Journal of Forecasting, 2022.