A Simple Low Cost Parallel Architecture for Big Data Analytics

Carlos Ordonez
University of Houston[§]
USA

Sikder Tahsin Al-Amin[*]
University of Houston[§]
USA

Xiantian Zhou
University of Houston[§]
USA

***Abstract*—Big Data Systems (Hadoop, DBMSs) require a complicated setup and tuning to store and process big data on a parallel cluster. This is mainly due to static partitioning when data sets are loaded or copied into the file system. Parallel processing thereafter works in a distributed manner, aiming for balanced parallel execution across nodes. Node synchronization, data redistribution and distributed caching in main memory are difficult to tune in the system. On the other hand, there exist analytical problems and algorithms, which can be computed in parallel, with minimal synchronization and fully independent computation. Moreover, some problems can be solved in one pass or few passes. In this paper, we introduce a low cost, yet useful, processing architecture in which data sets are dynamically partitioned at run-time and storage is transient. Each node processes one partition independently and partial results are gathered at the master processing node. Surprisingly, we show this architecture works well for some popular machine learning models as well as some graph algorithms. We attempt to identify which problem characteristics enable such efficient processing, and we also show the main bottleneck is the initial data set partitioning and distribution across nodes. We anticipate our architecture can benefit parallel processing in the cloud, where a dynamic number of virtual processors is decided at runtime or when the data set is analyzed for a short time.**

***Index Terms*—Parallel architecture, Big Data, Parallel Processing.**

I. Introduction

There has been a significant rising in data volumes and processing speeds for the last two decades. However, data volumes have risen at a much higher rate than the processing speeds. Though there are powerful machines with a lot of memory and disk space, it is costly and may fail when the data volume is huge. Therefore, processing and analyzing large volumes of data becomes non-feasible using a traditional serial approach. Hence, parallel processing emerges to solve the problem. Parallel processing allows a problem to be subdivided into smaller pieces that can be solved faster. Distributing the data across multiple processing units and parallel processing unit yields improved processing speeds [12].

Many abstract models of parallel processing have been introduced for partitioning, processing, and storage. Most approaches start with partitioning, that is a large data set is partitioned among multiple processing nodes where each node operates on the assigned partitioned data. Though there are some variants of parallel processing, it is often assumed that the same set of operations must be performed in each processing machine with shared-nothing architecture. For the output, most models send the partial output to the master node and combine the results to get the final result.

In this paper, our contributions are the following: (1) We propose a simple parallel architecture that can be used for parallel processing in big data analytics. (2) Our architecture does not depend on any external complicated file systems, rather we do the partition dynamically and run on commodity hardware. We use the file system "as is". (3) Our architecture is cheap, easy to set up, more machines can be added easily, and there is no need to maintain the partitions.

This is an outline of the rest of this article. Section 2 is a reference section. Section 3 presents our theoretical research contributions where we present our parallel architecture and how it comply with machine learning problems. Section 4 presents an experimental evaluation comparing our solution to the state of the art analytic systems. We discuss closely related work in Section 5. Conclusions and directions for future work are discussed in Section 6.

II. Preliminaries

In this section, we introduce the definitions and symbols used throughout the paper.

A. Input Data Set and Output Solution

We start by defining the input data set as D. Here, D is a matrix having n rows and a different number of columns, depending if the problem comes from machine learning or graphs. Matrix D can be either dense or sparse. We define the problem solution in a generalized manner as Θ. For machine learning problems, Θ is a model consisting of a list of matrices and associated metrics and for graphs, Θ is generally a vector and associated metrics.

B. Parallel Cluster

We define N as the number of processing nodes (also called workers), where each node has its own CPU and memory (i.e. a shared-nothing architecture) and it cannot directly access another node main memory or storage. There exists a separate master node controlling the computation, gathering partial results and returning a final solution. The heavy duty work I/O and CPU computation is done by the workers and the master

[§]Department of Computer Science, University of Houston, Houston TX 77204, USA
[*]Contact Author: stahsin.cse@gmail.com

node does mostly CPU computation, but on much smaller input transferred from the workers. We use I to refer to each worker or each partition, to avoid confusion with i, used for point, vertex or record id.

III. Proposed Low Cost Parallel Architecture

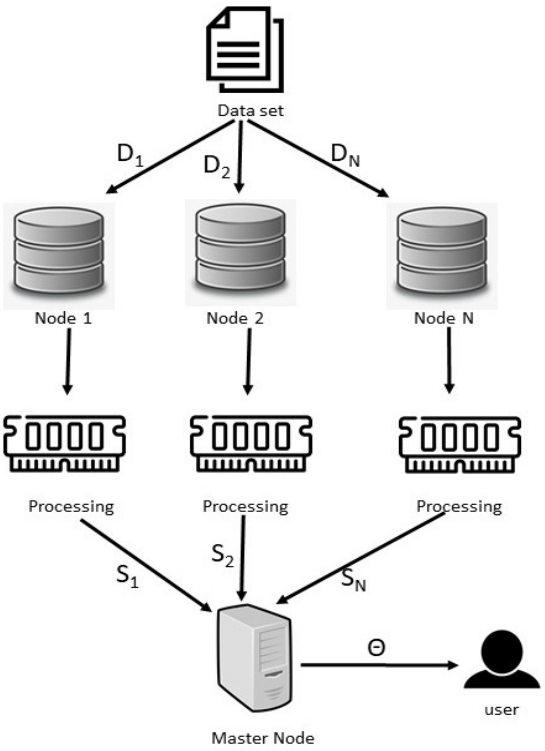

Fig. 1: Proposed Simple Parallel Architecture.

In this section, we first discuss some our of main assumptions in our low cost architecture and then we present our proposed parallel architecture in general terms. Finally, we discuss two instances of our proposed architecture: machine learning and graph algorithms. Fig 1 shows our proposed simple parallel architecture.

A. Main Assumptions in our Low Cost Architecture

- We assume $N \ll n$. This is important to guarantee each node holds a large partition of D. Our architecture is inefficient if $N = O(n)$, in which case the cost of partitioning and distribution may surpass CPU cost.
- In the case of Machine Learning we assume d dimensions, $d \ll n$ and matrix sizes in Θ is $O(d^2)$. On the other hand, in graphs we assume the solution vector in Θ is $O(n)$ or smaller, which is much smaller than $O(n^2)$ as $n \to \infty$. Therefore, we assume Θ and associated partial results at each node are much smaller than D.
- We assume each node stores complete rows of D_I (points, vertex neighbors). The data set D is horizontally partitioned on the n rows, transferring complete rows to each node. That is, each vector $i \in 1 \ldots n$ is never partitioned. Therefore, each node stores $\approx n/N$ vectors
- We assume D_I does not fit in the main memory. In other words, D_I is stored on secondary storage, reading one block at a time. On the other hand, Θ_I and associated partial results are always updated in main memory at each node. Therefore, the algorithm memory footprint is small.
- We assume 1:N communication and data transfer. The master node sends data to the N workers and the N workers send partial results to the master node. The workers do not transfer data to each other, nor they communicate in a 1:1 fashion.
- The analytic algorithm may be iterative, but it must work by sending partial results back and forth to the master node. In particular, for some problems the algorithm may work in one iteration.

B. Generic Algorithm for our Parallel Architecture

First, we discuss our parallel architecture in general terms. The general algorithm is presented in Algorithm 1.

Algorithm 1: Partial Solution of Our Generic Parallel Algorithm.

Data: Data set (D)
Result: Θ
1 Read input parameter N (Number of machines) ;
2 Sequential: Partition D in to $D_I = D_1, D_2, ..., D_N$ in the master node;
3 Sequential: Transfer D_I to $1, 2, ..., N$ machines;
4 Parallel: **for** I=1...N **do**
5 Compute Analytical Solution on Partitioned Data ;
6 Return partial summaries/result (S_I) to the master node.
7 **end**
8 Parallel/Sequential: Gather partial summaries/result (S_I) from N machines and merge in the Master node;
9 Compute and return final output (Θ);

1) Input: We assume that the input file is a text file that is stored in the master node. It can be in any text file format like .csv, .txt, and so on. However, if the file is in binary format, we need a connector to convert the binary file. As mentioned in Section II, the input file contains the input data set which we define as D. It has n records and can be a machine learning data set or graph data set. As for the storage, D may mostly contain integer and double values. The number of machines is also predefined (N) and we assume $n \gg N$, a reasonable assumption in most cases.

2) Data Set Partition: Here, we partition D at the master node and then send the N partitions to the worker nodes. In general, we send the whole row (vector) in one machine. For ML problems, we partition D based on row-id. We define the row-id as $1...n$. For uniform partitioning, we partition the input data as n/N (assuming $n \gg N$). So, D is partitioned into $D_1, D_2, ..., D_N$, where each partitioned D has the same number of rows (except for the last one in most cases). For example, if we have a file with 100 million rows and we want to partition it in 10 nodes, then each node will have 10 million

rows each. In case of graphs, we partition D based on vertex-id. That is, all the neighbours of a vertex are in the same machine. After we build the partition, we send them to the processing nodes. Though we do it in a serial manner now, a direction of future work will be to build the partition in the main memory and send them to the processing nodes. However, we are not currently performing hashing or any other mechanism to scatter the rows on the N worker nodes, as it is done by the parallel DBMSs. We leave it for future work. After transferring of the partitioned D is complete, each node will have a portion of the original D. This partitioned D will be used for processing in that node locally.

Though there are many distributed file systems that perform a similar kind of operation, our method is much simpler and we do it in a dynamic manner. We must emphasize that we are not doing this on a parallel complicated set up of lustre, DBMS, or Hadoop. Rather we are relying on the plain file system exploiting the OS. However, there are some drawbacks of our solution. There is no fault tolerance, it is hard to keep track of partitioned D when N is high, and we need to load the partitioned D to the host programming language.

3) Parallel Processing: As mentioned in the previous step, the partitioned D resides in each node. Here, we perform the processing on the partitioned D depending on the problem domain ($S()$). Usually, each node performs the same set of instructions locally. Partitioned D needed to be loaded in the host programming language and the processing happens in all the nodes in parallel. This phase uses the programming language, tools, or libraries that are used to perform the computation. We need to install the necessary tools or libraries in each machine. For example, if the user wants to compute the sum of a matrix using Python numpy library, the user needs to install Python and numpy in all the nodes. After the processing is done, the partial summaries/result resides (S_I) in each node and they are ready to send to the master node.

4) Partial Result: To return Θ, the final result, to the user, first, we need to send the partial output (S_I) from each machine to the master node. The partial outputs can be sent to the master node all at once or in a serial manner. We only send minimal information to the master node. Sending a huge chunk of output from each machine to the master node will be time consuming but it can be sent if the user requests it. Depending on the problem, the partial outputs can be merged, added, multiplied, and so on. We get the final output (Θ) from these merged outputs or doing further computations on it.

C. Instance of Our Architecture: Machine Learning

Here, we discuss an instance of our architecture, Machine Learning. Machine learning is essential to big data analytics. Within big data, data summarization has received much attention among the machine learning practitioners. Summarizing large data sets to accelerate the machine learning models can be done in parallel. Here, we take the data summarization technique to compute a summarization matrix and then compute supervised and unsupervised ML models exploiting the summarization matrix as an example to illustrate how we can integrate the proposed architecture in Section III-B with ML problems.

Let us assume the input matrix as D (Sec II), an $n \times d$ dense matrix, which is stored in the master node. Here, for ML problems, D has d columns and we assume $d < n$. We partition D as n/N (defined in Section III-B) and send the partitioned data sets to the processing machines. We partition it this way so that each machine has all the d columns. Next, we compute a summarization matrix in each machine locally. The goal is to get a final summarization matrix on the entire data set and compute machine learning models exploiting this. We should emphasize that this summarization can be any summary. In other words, here summarization matrix is an instance of our architecture.

A detailed discussion on how to compute a summarization matrix, called Gamma (Γ) is presented in [2]. As mentioned, we can compute one summarization matrix or k-summarization matrices depending on the model. Each machine locally computes the summary where data is processed one block at a time, with intermediate results updated for each block. When all the data is processed, we get the summary (partial output, S_I) for that machine. This is much smaller than the original data set D, and can be maintained in the main memory. The size of the summarization matrix mentioned in [2] is $d \times d$ and k-summarization matrix is $d + 1 \times 2k$, where d is the number of columns and k is the number of classes/clusters. As, both matrices are small in size compared to the data set D, they can be easily sent over the network to the master node. The master node collects all the local summarization matrices (ex: $S_1, S_2, ..., S_N$) and perform a simple matrix addition to compute the final summarization matrix (ex: $S = S_1 + S_2 + ... + S_N$). Based on this final summarization matrix, we can compute the ML models (Θ) on the master node. We can compute models like Linear Regression (LR), Principal Component Analysis (PCA), Naïve Bayes (NB), and so on. However, we will show the LR as a representative model in Section 4. We emphasize that we are not computing partial models in each machine. Also, we are not mentioning the programming aspects of the problem as that is not in the scope of this paper. A detail explanation can be found in [2].

D. Instance of Our Architecture: Graphs

Graph analytics is a field that is increasing its importance every day. Many graph algorithms are computationally intensive. On the other hand, graph data sets are becoming unexpectedly large as real-world information sources become more diverse and rich. Thus, a distributed system is better for processing graph problems. Partitioning the graph data sets to achieve work balance and process data in parallel becomes an essential problem. Single-source reachability is a core and fundamental computation task in graph analytics because it is the fundamental and representative problem, its solution is a guideline to solve many other harder graph problems such as transitive closure, shortest paths, connectivity, page ranks and so on. However, the single-source reachability (SSR)

problem remains one of the most computationally intensive and challenging tasks due to the large graph size.

Suppose a graph G has n vertices and m edges. The adjacency matrix of G is a $n \times n$ matrix such that the cell i, j holds 1 when exists an edge from vertex i to vertex j, while the cell i, j holds 0 when there is no edge from i to j. Each row of the adjacent matrix is stored as a record in the input file. we can also use an adjacency list to represent the graph since most real graph data sets are sparse. Each list describes the set of neighbors of a vertex in the graph. And each list is a record in the input file. Reachability from a source vertex s is the problem aimed to find the set of vertices S such that $i \in S$ iff exists a path from s to i. In a mathematics perspective, S can be a vector where $S[i]$ = 1 if exists a path between the source vertex and i, otherwise $S[i]$ = 0.

Recall that we partition the data as n/N where N is the number of worker nodes in the parallel system and n is the number of rows. Note that no matter the input file is in dense graph format or sparse graph format, each row (or record) has the source vertex and all its neighbors. we partition the n complete rows evenly across the distributed system, each working node has all neighbors of those vertices partitioned into it. After partition, we send a partition D_I to each working node and perform the SSR algorithm.

SSR can be solved with a Depth-first search (DFS) from the source vertex s, a Breadth-first search (BFS) from s, or via matrix-vector multiplications. In [4], the authors explain that a BFS starting from s can be done using a sparse vector S (initialized as $S[s] = 1$, and 0 otherwise), and multiplying iteratively D by S, as shown in the following. Note that we perform the matrix-vector multiplication by updating the input vector S.

$$S = (D^T)^k \cdot S = D^T \cdots (D^T \cdot (D^T \cdot S)) \quad (1)$$

where $\cdot$ is the regular matrix multiplication and S is a vector such that: $S[i] = 1$ when $i = s$, and 0 otherwise.

The result of matrix-vector multiplication is also a vector. We can assume the vector fits in main memory since its size is $O(n)$. S is replicated on each worker node. Note that we cannot assume the partition D_I fits in main memory. Each D_I is stored in disk. When performing the matrix-vector multiplication, the D_I is read from the disk while the vector S is updated in main memory in each node. And each working node gets the partial result of reachability.

$$S_I = D_I^T \cdot S, I \in 1...n \quad (2)$$

Then, all nodes send the partial result S_I to the master node and the master node computes the global reachability by summarizing the partial result, $S = S_1 + S_2 \cdots + S_n$. Since the result is a binary vector, a bitwise OR operator can be used to do the summary efficiently. After summarizing, the new S is sent to each working node to perform the next matrix-vector multiplication. The parallel matrix multiplication is shown in Figure2. In general, each step can be done in parallel without message transferring between nodes since S is replicated on each node. And S is updated in main memory in each node. The solution of reachability problem is multiple steps of matrix-vector multiplication.

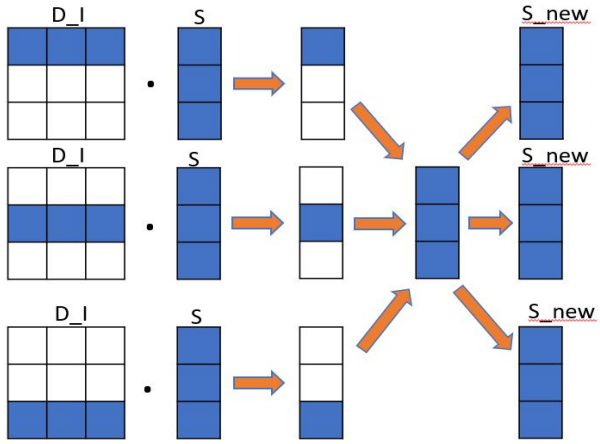

Fig. 2: Parallel computation of matrix-Vector multiplication.

IV. Experimental Evaluation

Here, we present an experimental evaluation, focusing on time performance. First, we introduce the parallel systems and input data sets. We perform benchmarking for machine learning problems where we compare our solution with other popular big data analytic systems: Spark and parallel DBMS.

A. Experimental Setup

The system used for the experiments is a parallel cluster with 8 machines where each machine has Pentium(R) Quadcore CPU running at 1.60 GHz with Linux Ubuntu as the operating system. The system has 8 GB of RAM and 1 TB of storage space. For parallel processing, we have total 64 GB of RAM and 8TB of disk space. The master node has also the same configuration as the other machines and it is not a part of parallel processing. It is only used to partition the data set at the beginning and finally gather the partial outputs from the processing nodes to form the final output.

TABLE I: Base data sets description.

Data set	n	Description	Model
synthG1M	1M	directed graph	Graph
synthG10M	10M	directed graph	Graph
YearPredictionMSD	515K	predicts if rain or not	LR
Creditcard	285K	raise in credit line	NB

The data sets used to perform the experiments are summarized in Table I. For graph problems, we used synthetic graphs And for ML, we used the YearPredictionMSD and Creditcard data set, both obtained from UCI machine learning repository.

B. Benchmarking: Machine Learning

Here, we compare our prototype solution with a machine learning problem. We compute the linear regression model and the Naïve Bayes model as mentioned in [5]. We perform the partitioning and processing as discussed in Section III-B. We use simple UNIX commands (ex: *split, scp*) to partition

TABLE II: Time (in Seconds) to compute the Linear Regression model using our solution ($N = 8$ nodes), in Spark ($N = 8$ nodes), and in a parallel DBMS ($N = 8$ nodes) (M=Millions).

			Our solution		(N=8)		Spark	(N=8)		Parallel DBMS	(N=8)
Data set	n	d	Partition	Compute LR	Total	Partition	Compute LR	Total	Partition	Compute LR	Total
Year-	1M	10	9	12	21	7	41	48	29	9	38
Prediction	10M	10	23	29	52	17	286	303	141	32	173
	100M	10	317	218	535	181	1780	1961	1711	380	2091

TABLE III: Time (in Seconds) to compute the Naïve Bayes model using our solution ($N = 8$ nodes), in Spark ($N = 8$ nodes) (M=Millions).

				Our solution	(N=8)		Spark	(N=8)
Data set	n	d	Partition	Compute NB	Total	Partition	Compute NB	Total
Credit Card	1M	10	11	13	24	7	Crash	Crash
	10M	10	23	36	59	25	Crash	Crash
	100M	10	335	252	587	231	Crash	Crash

and transfer the partitioned data sets to N machines. As mentioned in III-B, the partitioned data sets are stored in the file system. First, we load them to the host programming language. Then we compute the local summarization matrix for each partitioned data set in parallel as discussed in Section III-C. The partial outputs are sent to the master node and they are combined to get the final summarization matrix (S). The ML model (Θ) is then computed exploiting this final summarization matrix. We compare our solution with Spark and a parallel columnar DBMS (Vertica). Spark is a data processing engine developed to provide faster and easy-to-use analytics than Hadoop MapReduce.

Table II represents the time to compute the linear regression model for our solution, Spark, and DBMS in parallel ($N = 8$) machines. For Spark, we use HDFS to partition the data set, and for DBMS, we are partitioning the data set using the default "load" (COPY) command provided by the DBMS. Both have complicated algorithms behind which is hard to understand and almost difficult to tune for any average analyst. On the other hand, our method with UNIX commands are simple, straightforward and easy to understand. Then in Spark, we run the algorithm using Spark-MLlib library. For DBMS, we compute the summarization matrix using UDF and SQL queries which we adapted from [11]. We perform the experiments with varying $n = 1M, 10M, 100M$ rows and a fixed d=10 columns to demonstrate how large data sets perform on both. We can see from Table II that the time to partitioning and transferring the data set (column 'partition') is almost similar in both our solution and Spark for $n = 1M, 10M$. But parallel DBMS is much slower as it has a complicated algorithm behind data loading into the DBMS. On the other hand, computing LR part is much faster using our method than both methods as we are optimizing the data summarization and computing models as mentioned in [2]. For $n = 100M$, our method is a bit slower for partitioning than HDFS but the overall time is faster as we are doing the parallel execution much faster. In case of DBMS, our solution is faster in both cases. HDFS is faster in partitioning this case because we are partitioning the data set first and then transferring the data set over the network. This is a bottleneck and it can be addressed by building the partition in the main memory and send them to the processing nodes as mentioned in Section III-B.

Similarly, Table III shows the time to compute the Naïve Bayes model with our solution and in Spark. We compute our solution following the same procedure mentioned above. However, we used k-summarization matrix instead of summarization matrix to compute NB where k is the number of classes in D. As there is no previous implementation of the k-summarization matrix in a parallel DBMS, we do not include the comparison here. We can see that the partitioning has the similar time as before. However, computing the NB in Spark crashes as the Spark-MLlib implementation does not support negative values in the data set.

C. Benchmarking: Graph Algorithm

As explained in Section , we need to partition and send the data across the system before processing the SSR algorithm. Our proposed system is compared against popular analytic platforms, including a fast columnar DBMS and Spark which uses HDFS to partition the data set. The column-oriented DBMS could achieve magnitude performance improvement than a row-oriented DBMS [1], [3]. And it has been proved columnar DBMSs could provide significant accelerations in some queries[10]. So we use a columnar DBMS to do the comparison. We will compare the partition time of our solution with those two popular analytic platforms.

Table IV shows the comparison results. As can be seen, the partition time of our system can compete with other two popular analytic platforms. The parallel DBMS is the fastest one, but it is considered inadequate for complicated graph algorithms. Our solution is a faster than Spark which use HDFS to partition the data. So, the simple low cost parallel architecture works for graph problems. We do not include the processing times for SSR because they require significant programming effort in C++ combined with R/Python.

V. Related Work

Parallel processing is very common and many systems have been proposed for processing large data sets. Here, we discuss some of the closely related and popular systems. However,

TABLE IV: Time (in Seconds) to partition the graph data set using our solution ($N = 8$ nodes), in Spark ($N = 8$ nodes), and in a parallel DBMS ($N = 8$ nodes).

Data set	n	m	Our solution	Spark	Parallel DBMS
synthG1M	1M	10	8	8	3
synthG10M	10M	10	33	35	30

there have been some research that focuses on data partitioning on a specific problem [8], [15], [14].

HDFS [13] is the primary storage system used by Hadoop applications that allows a user to store huge data files across multiple nodes while keeping the image of a centrally located file. However, the installation procedure is complex and it does not suit well for small data. On the other hand, HBase [9] is a noSQL distributed database developed on top of HDFS which enables faster search and retrieval of the data.

Spark [16] is another distributed data processing framework that can work on very large data sets. Some researches have been done in Spark for data partitioning [7], [6]. Also, Spark has an extensive library for graph and machine learning models where it can outperform Hadoop. Similar to ours, Spark works in main memory. However, as discussed in Section IV-B, we compared our solution against Spark libraries and gained an edge in some cases.

Parallel DBMSs are also used for big data analytics in many cases [3], [17], [11]. They improve the performance of data processing by parallelizing loading, indexing, and querying data. The parallel DBMS implements the concept of horizontal partitioning by distributing parts of a large relational table across multiple nodes. This is essential to obtain scalable performance of SQL queries.

VI. Conclusions

We proposed a simplified parallel architecture that can be applied the solve ML problems in big data analytics. Our prototype is cheap and there is no need to install complicated libraries like other platforms do. Rather we do it dynamically running on commodity hardware and using the file system "as is". We also proposed a generic parallel algorithm exploiting our architecture that can work across multiple programming languages and platforms. We then studied how our solution can be used to solve machine learning problems, and what are the drawbacks if we apply it to other complicated areas such as graph algorithms. We justified why our solution is needed. Experimental results proved that even if we did not achieve superior performance than others in all cases, our prototype can indeed compete with other parallel systems including Spark and parallel DBMS. We believe easy set up, using the disk file system, flexibility to add more number of machines, and cheap architecture make our solution more attractive.

There are some shortcomings of our solution. The partitioned data set in each machine needs to be loaded into the host programming language to perform analysis. Also, there is no fault tolerance. For our future work, we plan to build the partition in the main memory and send them to the processing nodes.

References

[1] Abadi, D., Madden, S., Hachem, N.: Column-stores vs. row-stores: how different are they really? In: Proc. ACM SIGMOD Conference. pp. 967–980 (2008)

[2] Al-Amin, S.T., Ordonez, C.: Scalable machine learning on popular analytic languages with parallel data summarization. In: Big Data Analytics and Knowledge Discovery - 22nd International Conference, DaWaK 2020. vol. 12393, pp. 269–284 (2020)

[3] Al-Amin, S.T., Ordonez, C., Bellatreche, L.: Big data analytics: Exploring graphs with optimized SQL queries. In: Proc.DEXA Conference. pp. 88–100 (2018)

[4] Cabrera, W., Ordonez, C.: Scalable parallel graph algorithms with matrix-vector multiplication evaluated with queries. Distributed and Parallel Databases **35**(3-4), 335–362 (2017)

[5] Chebolu, S.U.S., Ordonez, C., Al-Amin, S.T.: Scalable machine learning in the R language using a summarization matrix. In: Database and Expert Systems Applications DEXA. pp. 247–262 (2019)

[6] Gounaris, A., Kougka, G., Tous, R., Montes, C.T., Torres, J.: Dynamic configuration of partitioning in spark applications. IEEE Trans. Parallel Distributed Syst. **28**(7), 1891–1904 (2017)

[7] Han, D., Agrawal, A., Liao, W., Choudhary, A.N.: Parallel DBSCAN algorithm using a data partitioning strategy with spark implementation. In: IEEE International Conference on Big Data (2018)

[8] Mayer, C., Mayer, R., Bhowmik, S., Epple, L., Rothermel, K.: HYPE: massive hypergraph partitioning with neighborhood expansion. In: IEEE International Conference on Big Data. pp. 458–467 (2018)

[9] Mehul Nalin Vora: Hadoop-hbase for large-scale data. In: Proceedings of 2011 International Conference on Computer Science and Network Technology. vol. 1, pp. 601–605 (2011)

[10] Ordonez, C.: Optimization of linear recursive queries in SQL. IEEE Transactions on Knowledge and Data Engineering (TKDE) **22**(2), 264–277 (2010)

[11] Ordonez, C., Zhang, Y., Cabrera, W.: The Gamma matrix to summarize dense and sparse data sets for big data analytics. IEEE Transactions on Knowledge and Data Engineering (TKDE) **28**(7), 1906–1918 (2016)

[12] Parhami, B.: Parallel processing with big data. In: Encyclopedia of Big Data Technologies. Springer (2019)

[13] Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The hadoop distributed file system. In: IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST. pp. 1–10. IEEE Computer Society (2010)

[14] Singh, D., Mohan, C.K.: Projection-svm: Distributed kernel support vector machine for big data using subspace partitioning. In: IEEE International Conference on Big Data. pp. 74–83 (2018)

[15] Wang, Z., et al., Y.C.: Scalable data cube analysis over big data. arXiv preprint arXiv:1311.5663 (2013)

[16] Zaharia, M., Chowdhury, M., Franklin, M., Shenker, S., Stoica, I.: Spark: Cluster computing with working sets. In: HotCloud USENIX Workshop (2010)

[17] Zhou, X., Ordonez, C.: Computing complex graph properties with SQL queries. In: 2019 IEEE International Conference on Big Data. pp. 4808–4816 (2019)