A Cloud System for Machine Learning Exploiting a Parallel Array DBMS

Yiqun Zhang, Carlos Ordonez, Lennart Johnsson
Department of Computer Science
University of Houston, USA

***Abstract*—Computing machine learning models in the cloud remains a central problem in big data analytics. In this work, we introduce a cloud analytic system exploiting a parallel array DBMS based on a classical shared-nothing architecture. Our approach combines in-DBMS data summarization with mathematical processing in an external program. We study how to summarize a data set in parallel assuming a large number of processing nodes and how to further accelerate it with GPUs. In contrast to most big data analytic systems, we do not use Java, HDFS, MapReduce or Spark: our system is programmed in C++ and C on top of a traditional Unix file system. In our system, models are efficiently computed using a suite of innovative parallel matrix operators, which compute comprehensive statistical summaries of a large input data set (matrix) in one pass, leaving the remaining mathematically complex computations, with matrices that fit in RAM, to R. In order to be competitive with the Hadoop ecosystem (i.e. HDFS and Spark RDDs) we also introduce a parallel load operator for large matrices and an automated, yet flexible, cluster configuration in the cloud. Experiments compare our system with Spark, showing orders of magnitude time improvement. A GPU with many cores widens the gap further. In summary, our system is a competitive solution.**

I. Introduction

Cloud computing is becoming increasingly important in big data analytics given its ability to scale out to a large number of processing nodes, with ample disk and RAM space. As business and scientific goals become more challenging they push the need for computing machine learning in the cloud. From the software side, there exists a long-time existing gap between mathematical packages and large-scale data processing platforms. On one hand, mathematical packages like R, Matlab, SAS and more recently Python provide comprehensive libraries for machine learning and statistical computation, but none of them are designed to scale to large data sets that exceed even a single machine's memory. On the other hand, Hadoop big data systems (e.g. MapReduce, Spark) and parallel DBMSs (e.g. Teradata, Oracle, Vertica) are two prominent families of platforms that offer tremendous storage and parallel processing capabilities. However, even though there is research progress, computing machine learning models remains difficult. UDFs have been shown a great mechanism to plug in analytic algorithms [1]. On the other hand, it is fair to say that R remains one of the most popular mathematical packages. Previous research attempted integrating R with Hadoop as well as DBMSs including SQL Server and Vertica. Such integration takes maximum advantage of R's mathematical capabilities and large-scale data processing capabilities from Hadoop and DBMSs. However, scalable parallel matrix computations remain difficult in cloud computing and parallel DBMSs.

Previous research has pointed out that DBMSs with specialized storage can achieve orders of magnitude in performance improvement. SciDB [4] is an innovative parallel DBMS with array storage, capable of managing unlimited size matrices (i.e. as large as space on secondary storage) and well-integrated with R (with powerful mathematical capabilities). Such scalable architecture opens research possibilities to scale machine learning algorithms. In this work, we explore how to leverage SciDB's capabilities to accelerate and scale the computation of machine learning models in the cloud (a large parallel cluster of computers). Salient technical contributions include the following:

1) Parallel data loading into 2-dimensional arrays (matrices).
2) Parallel data summarization in one scan returning a comprehensive summarization matrix, called Γ.
3) Pushing vector-vector outer product computation to RAM, resulting in a partitioned matrix multiplication, with no process synchronization and minimal overhead.
4) Accelerating parallel summarization with a GPU with many cores.
5) Automated installation and tuning of SciDB in the cloud (i.e. fast deployment).

II. Related Work

There is significant work on computing machine learning models in the cloud, before with MapReduce and currently with Spark. However, computing models with DBMS technology has received less attention [2]. The Γ summarization matrix was proposed in [3], which introduces two parallel matrix-based algorithms for dense and sparse matrices, respectively. This paper showed an array DBMS can be faster than R (removing RAM limitations), and faster than an SQL engine (using queries to summarize the data set) and surprisingly, faster than the high performance ScaLAPACK library. Later, [**?**] introduced an optimized UDF to compute the Γ matrix on a columnar DBMS, passing it to R for model computation. However, these papers did not explore the cloud angle using a parallel DBMS, with a large number of nodes, competing with a Big Data Hadoop system. Since matrix computations are CPU intensive, it was necessary to study how to further accelerate computation: GPUs were not considered. Another aspect we did not envision initially as a limitation turned out to

be a bottleneck: loading data into the array DBMS, especially in parallel. In this paper we tackle all these research issues: scale out parallelism, using GPUs, parallel matrix loading. A careful performance benchmark comparison between our system and Spark in the Amazon cloud rounds our contribution.

III. Background

This is a reference section that can be skipped if the reader is familiar with machine learning models and array DBMSs.

A. Input Data Set and Output Model

All models take a $d \times n$ matrix X as input. Let $X = \{x_1, x_2, \ldots, x_n\}$ be the input data set with n points, where each point x_i is a vector in $\mathbf{R}^d$. The goal is to compute some machine learning model Θ, minimizing some error metric. In linear regression (LR) and variable selection (VS), X is augmented with a $(d+1)$th dimension with output variable Y, represented by $\mathbf{X}$ as a $(d+1) \times n$ matrix. We use $i = 1 \ldots n$ and $j = 1 \ldots d$ as matrix subscripts. For convenience in mathematical notation we use column vectors and column-oriented matrices.

Our system allows the computation of a wide range of linear (fundamental) models including principal component analysis (PCA), linear regression (LR), variable selection (VS), Naïve Bayes classifier, K-means (and EM) clustering, logistic regression and linear discriminant analysis. Those models involve many matrix computations, which are a good fit for an array DBMS. All the models we support can derive their computations from the data summarization matrix Γ. However, non-linear models such as SVMs, decision trees and deep neural networks are not supported.

B. Overview of the SciDB Array DBMS

SciDB stores data as multi-dimensional arrays in chunks, instead of rows compared to traditional DBMSs, where each chunk is in effect a small array that is used as the I/O unit. SciDB preserves the most important features from traditional DBMSs like external algorithms, efficient I/O, concurrency control and parallel processing. From a mathematical perspective, SciDB offers a rich library with matrix operators and mathematical functions. From a systems angle, SciDB allows the development of new array operators extending its basic set of operators.

IV. Machine Learning in the Cloud with a Parallel Array DBMS

A. System Architecture

Figure 1 illustrates our major system components in an N-node parallel cluster showing how data moves through them. In order to simplify the diagram each node in the cluster runs only one SciDB instance, but we want to point out it is feasible to spawn one instance per core. We emphasize that N can be large (i.e. a cloud).

The data set file is first uploaded from a local server to the cloud. After the cluster has been set up, SciDB loads the data set into a two-dimensional array in parallel. The input matrix thereafter goes through a two-phase algorithm [3]:

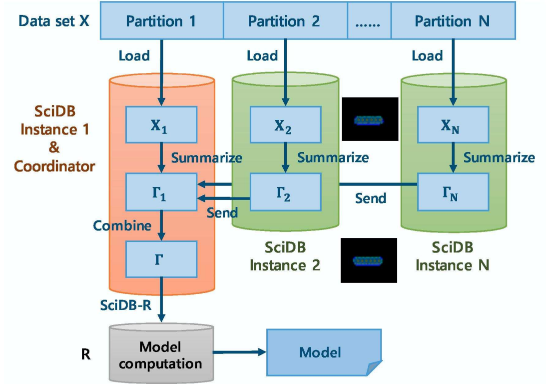

Fig. 1. Diagram of system components in an N-node cluster (1 SciDB instance per node) and data flow.

1) Parallel data summarization to get Γ;
2) Model Θ computation in RAM using R, including iterative behavior if necessary.

During the data summarization phase the Γ matrix is computed in parallel using our user-defined operator in SciDB. The Γ matrix is far smaller than the original data set X and can easily fit in the main memory. We will then compute the model exploiting the Γ matrix in R, using the SciDB-R library.

Notice that in our system, because R only plays the interface role between the user and the parallel array DBMS, it resides only on the coordinator node, not on every worker node. R by architecture is a single threaded platform for one node. Our system use SciDB to do the heavy processing in parallel, the summarization matrix is condensed in the coordinator. Based on that fact that model computation itself fits in RAM and its not I/O intensive we exploit R to compute the model. Even though we are aware that there are many efforts from people to make R work in parallel with multiple nodes, we concluded that placing R only in the coordinator is an acceptable solution because Γ fits in RAM.

B. Parallel Data Loading

SciDB provides two mechanisms for loading data into arrays: (1) Convert the CSV file into a temporary file with a SciDB-specific array format for later loading, or (2) We have to load the data into a one-dimensional tuple array first and then go through a re-dimensioning process to transform the one-dimensional array (array of tuples) into a 2-d array (matrix). Since the second mechanism can be used directly with CSV files it is the default. Unfortunately, both mechanisms are slow in a cloud environment. Both loading alternatives require 2 full disk reads and 2 full disk writes. (one full disk read from the CSV file, then one full disk write, using SciDB-specific text format or as a one-dimensional array, then 1 full disk read to read them back, finally, a full disk write into the matrix format). In addition, re-dimensioning requires completely reshaping d sets of 1D chunks and transforming them into one set of 2D chunks with a different data partitioning (i.e. slow

reshuffling of data). Since both approaches are inefficient to load the data in parallel, we developed a novel user-defined operator $load2d()$ for parallel matrix loading into SciDB.

We now explain our optimized loading operator in terms of X, an $n \times d$ matrix loaded in parallel using N nodes. We set the chunk size for our matrix in SciDB to $10000 \times d$ as a default setting. To get started, based on n and d, the operator computes how many chunks are needed: $C_{total} = \lfloor \frac{n-1}{10000} + 1 \rfloor$, and determine how many chunks each node will store: $C_{each} = \lfloor \frac{C_{total}-1}{N} + 1 \rfloor$. As a second step, the coordinator node scans the file and determines the segment of the file which each instance will need to read based on C_{each} and the vertical chunk size (10000). Then the N nodes start reading the file in parallel without locking. SciDB uses a standard round-robin algorithm to distribute and write chunks. Our loading operator directly outputs the matrix in SciDB chunk format with optimal chunk size, avoiding any intermediate files on disk and the slow re-dimensioning process. To summarize, our parallel load operator can load significantly faster than the built-in SciDB loading function because: (1) It requires less I/O work: only 2 disk scans to read data and 1 disk scan to write. (2) It saves significant CPU time by not re-dimensioning from 1D into 2D. (3) There is no data redistribution among the N nodes, which would add communication and double I/O overhead.

C. Data Summarization Matrix

This section applies to any parallel system, including Hadoop systems. A key optimization of our system in the algorithms is that we implemented our statistical model computations based on a data summarization matrix Γ instead of the large data set X or $\mathbf{X}$.

Γ *Matrix:* As mentioned in Section III-A, supervised models, like linear regression, use an *augmented* matrix $\mathbf{X}$. We introduce a more general matrix $\mathbf{Z}$, created by appending $\mathbf{X}$ with a row vector of 1s. Since X is $d \times n$, $\mathbf{Z}$ has $(d+2)$ rows and n columns, where $Z[0]$ is a row-vector of n 1s and $Z[d+1]$ is Y.

The Γ matrix is a generalization of sufficient statistics [1] and it is defined as:

$$\begin{bmatrix} n & L^T & \mathbf{1}^T \cdot Y \\ L & Q & XY^T \\ Y \cdot \mathbf{1} & YX^T & YY^T \end{bmatrix} = \begin{bmatrix} n & \sum x_i^T & \sum y_i \\ \sum x_i & \sum x_i x_i^T & \sum x_i y_i \\ \sum y_i & \sum y_i x_i^T & \sum y_i^2 \end{bmatrix}$$

Matrix Γ contains a comprehensive, accurate and sufficient summary of X to efficiently compute all models previously mentioned. Due to space limitation we will not expand in detail how the models we mentioned in this article can derive their computations from the Γ matrix. Those details are included in our previous work [3].

D. Parallel Summarization with new Array Operator

A fundamental property of the Γ matrix is that it can be computed by a single matrix multiplication using $\mathbf{Z}$. Therefore, the computation of Γ can be expressed as the matrix product $\Gamma = \mathbf{Z}\mathbf{Z}^{\mathbf{T}} = \sum_{i=1}^{n} z_i \cdot z_i^T$. Our previous research [3] examined all available programming mechanisms in SciDB to compute the Γ matrix. Experimental results showed that the user-defined operator is the most efficient approach.

Our Γ operator works fully in parallel with a partition schema of X into N subsets $X^{[1]} \cup X^{[2]} \cup \ldots \cup X^{[N]} = X$, where we independently compute $\Gamma^{[I]}$ on $X^{[I]}$ for each node I (we use this notation to avoid confusion with matrix powers and matrix entries). In SciDB terms, each worker I will independently compute $\Gamma^{[I]}$. No synchronization between instances is needed once the computation starts therefore the parallelism is guaranteed. When all workers are done the coordinator node will gather all results and compute a global $\Gamma = \Gamma^{[1]} + \ldots + \Gamma^{[N]}$ with $O(d^2)$ communication overhead per node (much smaller than $O(dn)$. This is essentially a natural parallel computation, coming from the fact that we can push the actual multiplication of $z_i \cdot z_i$ into the array operator.

E. Accelerating Summarization with a GPU

Computing Γ by evaluating $\mathbf{Z}\mathbf{Z}^T$ is a bad idea because of the cost of computing and materializing $\mathbf{Z}^T$. Instead, we evaluate the sum of vector outer products $\sum_{i=1}^{n} z_i \cdot z_i^T$, which is easier to parallelize and update in RAM. Moreover, since Γ is symmetric, we only compute the lower triangle of the matrix during computation to save execution time. Our Γ matrix multiplication algorithm works fully in parallel with a partition of X into N subsets $X^{[1]} \cup X^{[2]} \cup \ldots \cup X^{[N]} = X$, where we independently compute $\Gamma^{[I]}$ on $X^{[I]}$ for each processor I. A fundamental aspect is to optimize computation when $\mathbf{X}$ (and therefore $\mathbf{Z}$) is sparse: any multiplication by zero returns zero. Specifically, $n_{nz} \leq \sqrt{d}n$ can be used as a threshold to decide using a sparse or algorithm, where n_{nz} is the number of non-zero entries in X. Computing Γ on sparse matrices with a GPU is challenging because tracking and partitioning non-zero entries may reduce parallelism by adding more complex logic in the operator code. In this paper we focus on the dense matrix algorithm.

Our following discussion focuses on integration with an array DBMS, given its ability to manipulate matrices. From a systems perspective, we integrated our model computation using Γ with the SciDB array DBMS and the R language, with a 2-phase algorithm: (1) data summarization in one pass returning Γ; (2) exploiting Γ in intermediate computations to compute model Θ. We emphasize z_i is assembled in RAM (we avoid materializing z_i to disk). Unlike normal UDFs in a traditional DBMS, which usually need to serialize a matrix into binary/string format then deserialize it in succeeding steps, our operators in SciDB returns Γ directly in array format, which is a big step forward compared to previous in-DBMS approaches. Phase 2 takes place in R on the master node leveraging R's rich mathematical operators and functions. Although this phase does not run in parallel across nodes, it does not impact the overall performance since Γ passed from SciDB in the first step is much smaller than the data set, resulting in much faster iterations working in RAM. That is, Phase 1, computing Γ, is the main task to parallelize.

Parallel processing happens as follows. In SciDB, arrays are partitioned and stored as chunks and such chunks are only accessible by C++ iterators. In our previous work [3], we compute the vector outer product $z_i \cdot z_i^T$ as we scan the data set in the operator. We emphasize z_i is assembled in RAM (we avoid materializing z_i to disk). In general, interleaving I/O with floating point computation is not good for GPUs because it breaks the SIMD paradigm. In our GPU accelerated operator, we first extract matrix entries in each chunk into main memory. Then we transfer the in-memory subsets of X to GPU memory, one chunk at a time. Then the GPU computes the vector outer products $z_i \cdot z_i^T$ fully in parallel with its massive amount of processing units (cores). The sum is always maintained in the GPU memory. It takes $log(n)$ time to accumulate n partial Γs into one using the reduction operation. When the computation finishes for the whole data set, the Γ matrix is transferred back from GPU memory to CPU main memory. The C++ operator code is anotated with OpenACC directives to work with GPU. In our current GPU version, the CPU only does the I/O part. Since the DBMS becomes responsible for only I/O our approach also has promise in relational DBMSs.

V. Experimental Evaluation

We present benchmarking experiments in the Amazon cloud (S3). We created a configuration script to automate the complicated system setup in parallel, which makes running SciDB in the cloud as easy as running HDFS/Spark. Moreover, our system provides a GUI, where all system components can be manipulated.

As mentioned above, we developed a parallel data loading operator that offers tight integration with Amazon S3 infrastructure. Due to lack of space we did not conduct a benchmark comparing loading speed between our cloud system and Spark. But we should point out our parallel operator to load matrix X from a CSV file takes similar time to copying the CSV file from the Unix file system to HDFS plus creating data set in Spark's RDD format. We would also like to point out that the time to load data is considered less important than the time to analyze data because it happens once.

A. Benchmark on a Large Cluster

We compare our array DBMS cloud solution with Spark, the most popular analytic system from the Hadoop big data world. We tried to be as fair as possible: both systems run on the same hardware, but we acknowledge SciDB benefits from manipulating data being pre-processed in matrix form. We analyze data summarization and model computation.

Hardware: a parallel cluster with $N = 100$ nodes, where each node has 2VCPUs running at 2 GHz, 8GB RAM and 1 TB. Spark takes advantage of an additional coordinator node with 16 GB RAM (i.e. $N = 101$ for Spark). As mentioned before, we developed scripts for fast automatic cluster configuration.

In Figure 2 we compare both systems computing the Γ summarization matrix. That is, we compute the submatrices L, Q in Spark using its *gramian()* function, without actually computing the machine learning model. That is, we could simply export L, Q from Spark to R and compute the model in R in RAM. As can be seen, our system scales out better than Spark as N grows. In fact, Spark increases time when going from $N = 10$ to $N = 100$, which highlights a sequential bottleneck. At $N = 100$ the gap between both systems is close to two orders of magnitude.

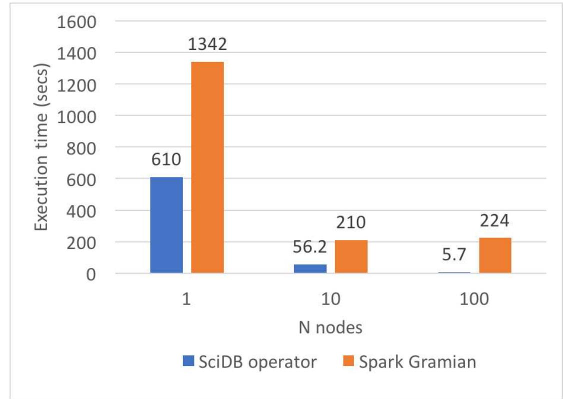

Fig. 2. Γ computation time: array DBMS vs. Spark.

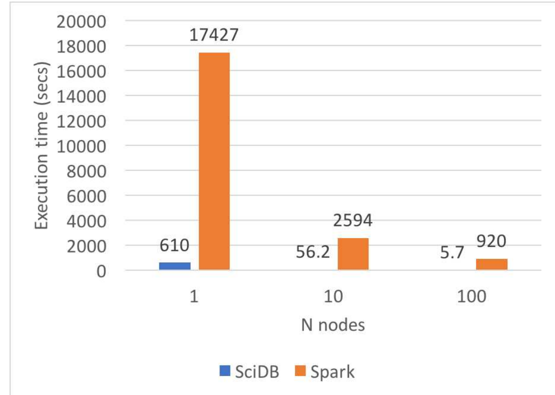

Fig. 3. Model computation time for LR: array DBMS vs. Spark.

Figure 3 compares both systems with a fundamental model: linear regression (LR). This comparison is also important because LR is computed with Stochastic Gradient Descent (SGD) in Spark, whereas ours is based on data summarization. In Spark we used 20 iterations, which was the recommended setting in the user's guide to get a stable solution. At $N = 1$ the difference is more than one order of magnitude, whereas at $N = 100$ our system is more than two orders of magnitude faster than Spark. We emphasize that even running Spark with one iteration, which would get an unstable solution, our system is 10X faster (i.e. $920/20 \approx 46$, compared to 6 seconds).

B. Benchmark with a Big GPU

The previous experiments beg the question if processing can be further accelerated with GPUs. Since Spark does not offer off the shelf functions exploiting GPUs we cannot compare it. However, we emphasize that we established that our solution is orders of magnitude faster than Spark. Therefore, it is unlikely that Spark could be faster exploiting GPUs.

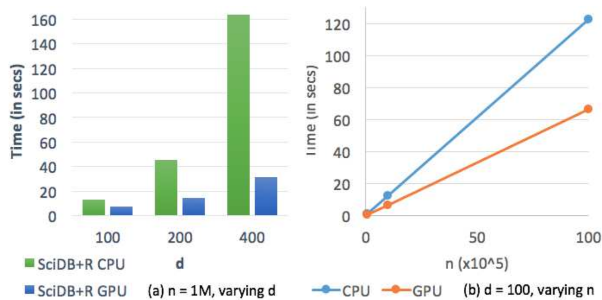

Fig. 4. Γ computation time: CPU vs. GPU.

TABLE I
COMPARING COMPUTATION OF MODEL Θ USING R, DBMS+R, AND DBMS+R+GPU; DENSE MATRIX OPERATOR; N=1 NODE (CPU=8 CORES); TIMES IN SECS.

n	d	model	CPU		GPU
			R	R+SciDB	R+SciDB
1M	100	PCA	29	14	8
1M	200	PCA	90	46	16
1M	400	PCA	fail	165	33
10M	100	PCA	fail	147	104
10M	200	PCA	fail	466	215
10M	400	PCA	fail	1598	455
10M	100	LR	fail	147	103
10M	200	LR	fail	464	212
10M	400	LR	fail	1594	451

Setup: In this section we examine how much time improvement GPU processing can bring to model computation using Γ. We ran our experiments on a parallel cluster with 4 nodes. Each node has an 8-core Intel Xeon E5-2670 processor, an NVIDIA GRID K520 GPU with 1,536 CUDA cores. The GPU card has 4GB of video memory, while the machine has 15 GB of main memory. On the software side, on those machines we installed Linux Ubuntu 14.04, which is currently the most reliable OS to run the SciDB array DBMS. The system is equipped with the latest NVIDIA GPU driver version 352.93. We also installed the PGI compilers for OpenACC and SciDB 15.7. We revised our CPU operator C++ code with OpenACC annotations marking key vector operations in the loops to be parallelized with GPU cores so that they are automatically distributed for parallelism. The data sets are synthetic, dense matrices with random numbers. We loaded the data sets into SciDB as arrays split into equal-sized chunks and evenly distributed across all parallel nodes, where each chunk holds 10,000 data points.

Figure 4 illustrates GPU impact on data summarization, which is the most time consuming step in our 2-phase algorithm. The bar figure shows the GPU has little impact at low d (we do not show times where $d < 100$ since the GPU has marginal impact), but the gap between CPU and GPU rapidly widens as d grows. On the other hand, the right plot shows the GPU has linear time complexity as n grows, an essential requirement given the I/O bottleneck to read X. Moreover, the acceleration w.r.t CPU remains constant.

We now analyze GPU impact on the overall model computation, considering machine learning models require iterative algorithms. Table I shows the overall impact of the GPU. As can be seen SciDB removes RAM limitations in the R runtime and it provides significant acceleration as d grows. The GPU provides further acceleration despite the fact the dense matrix operator is already highly optimized C++ code. The trend indicates the GPU becomes more effective as d grows. Acceleration is not optimal because there is overhead moving chunk data to contiguous memory space and transferring data to GPU memory and because the final sum phase needs to be synchronized. However, the higher d is, the more FLOP work done by parallel GPU cores.

VI. CONCLUSIONS

We presented a parallel analytic system for the cloud using a parallel array DBMS, not built on top of HDFS. That is, our system represents an alternative approach following a traditional shared-nothing architecture without a parallel file system. We introduced a summarization matrix that benefits many machine learning models. We studied how to optimize the computation of such matrix in an array DBMS, considering it has significantly different storage compared to traditional relational DBMSs. Moreover, we studied how to further accelerate summarization with GPUs. Our experiments showed our system is orders of magnitude faster than Spark, to summarize the data set and to compute a typical machine learning model. A GPU widens the gap.

There are many issues for future work. Our algorithms and optimizations are ideal for an array DBMS because they are matrix computations. However, they are applicable in any parallel DBMS, with some performance penalty. Our summarization matrix is a great fit for GPUs, but we need to study in more depth how to parallelize the vector outer products. Instead of competing with Spark we want to study how to integrate our algorithms and our system with Spark. Integrating our algorithms would require reprogramming them in Java or Scala. The main drawback about integrating systems is the need to move data between the DBMS and HDFS/Spark. A major limitation of our system, compared to Spark, is the lack of fault tolerance during computation: if a node fails the computation must be restarted.

REFERENCES

[1] C. Ordonez. Statistical model computation with UDFs. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 22(12):1752–1765, 2010.
[2] C. Ordonez. Can we analyze big data inside a DBMS? In *Proc. ACM DOLAP Workshop*, 2013.
[3] C. Ordonez, Y. Zhang, and W. Cabrera. 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.
[4] M. Stonebraker, P. Brown, D. Zhang, and J. Becla. SciDB: A Database Management System for Applications with Complex Analytics. *Computing in Science and Engineering*, 15(3):54–62, 2013.
[5] Y. Zhang, C. Ordonez, and W. Cabrera. Big data analytics integrating a parallel columnar DBMS and the R language. In *Proc. of IEEE CCGrid Conference*, 2016.