2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications

MapReduce: Limitations, Optimizations and Open Issues

Vasiliki Kalavri
KTH Royal Institute of Technology
Stockholm, Sweden
kalavri@kth.se

Vladimir Vlassov
KTH Royal Institute of Technology
Stockholm, Sweden
vladv@kth.se

***Abstract*—MapReduce has recently gained great popularity as a programming model for processing and analyzing massive data sets and is extensively used by academia and industry. Several implementations of the MapReduce model have emerged, the Apache Hadoop framework being the most widely adopted. Hadoop offers various utilities, such as a distributed file system, job scheduling and resource management capabilities and a Java API for writing applications. Hadoop's success has intrigued research interest and has led to various modifications and extensions to the framework. Implemented optimizations include performance improvements, programming model extensions, tuning automation and usability enhancements. In this paper, we discuss the current state of the Hadoop framework and its identified limitations. We present, compare and classify Hadoop/MapReduce variations, identify trends, open issues and possible future directions.**

***Keywords*-MapReduce, Big Data, Survey**

I. Introduction

Recent advances in technology have allowed organizations to collect extremely large amounts of data, anticipating high value in analyzing them. "Big Data" management and processing has been one of the biggest challenges of our era. Current approaches consist of processing systems deployed on large amounts of commodity machines and exploit massive parallelism to efficiently analyze enormous datasets. The most successful system is the Google's MapReduce framework [1], which hides the complexity of data distribution, communication and task scheduling and offers a simple programming model for writing analytical applications, while also providing strong fault-tolerance guarantees.

Several implementations of the MapReduce programming model have been proposed, with open-source Apache Hadoop framework [2] being the most widely adopted. Apart from the MapReduce programming model, Hadoop offers various other capabilities, including a distributed file system, HDFS [3] and a scheduling and resource management layer. Despite its popularity, the MapReduce model and its Hadoop implementation have also been criticized [4] and have been compared to modern parallel database management systems (DBMSs), in terms of performance and complexity [5]. There have been extensive studies on MapReduce characteristics, identifying a set of shortcomings of the model and current implementations [6], [7], [8]. Features such as the static map-shuffle-reduce pipeline, the frequent data materialization (writing data to disk), the lack of support for iterations and state transfer between jobs, the lack of indexes and schema and sensitivity to configuration parameters have been confirmed to contribute negatively in its performance, for certain classes of applications.

Numerous variations of Hadoop MapReduce have been developed during the last few years, proposing performance improvements, programming model extensions, automation of use and tuning. Each one of the extensions deals with one or more shortcomings of the vanilla Hadoop MapReduce implementation. The amount of these variations has grown significantly, making it hard for users to choose the appropriate tool. Existing surveys summarize some of these systems, however, there exists no complete study categorizing them and clarifying the trade-offs for a potential user. Having all these alternatives available, users either need to spend a lot of time researching which system would best fit their needs or resort to common Hadoop installations, even if such a choice would be suboptimal for their problem.

In this survey, we examine existing MapReduce implementations based on Hadoop. The scope of our study is strictly limited to systems extending or enhancing Hadoop and does not include more generalized data-flow systems, such as Dryad [9], Spark [10] and Stratosphere [11]. The contributions of this paper are:

- An overview of the state-of-the art in Hadoop/MapReduce optimizations;
- A comparison and classification of existing systems;
- A summary of the current state of the research field, identifying trends and open issues;
- A vision on possible future directions.

The rest of this paper is organized as follows. In Section II, we provide background on Hadoop/MapReduce and present the current state of the project. In Section III, we discuss the limitations of the model and implementation, as demonstrated in recent literature. Section IV makes a categorization and comparison of existing MapReduce variations. In Section V we focus on trends and open issues and propose future directions. We conclude in Section VI.

II. Background

This section gives an introduction to the MapReduce model and its open-source implementation, Hadoop.

A. The MapReduce Programming Model

The MapReduce programming model is designed to efficiently execute programs on large clusters, by exploiting data parallelism. A distributed file system is deployed on the same machines where the applications run, so that execution can benefit from data locality, by trying to move computation where the data reside. The model is inspired by functional programming and consists of two second-order functions, *Map* and *Reduce*, which form a static pipeline, where the Map stage is followed by the Reduce stage.

Data are read from the distributed file system, in the form of user-defined key-value pairs. These pairs are then grouped into subsets and serve as input for parallel instances of the Map function. A user-defined function must be specified and is applied to all subsets independently. The Map function outputs a new set of key-value pairs, which is then sorted by key and partitioned according to a partitioning function. The sorted data feed the next stage of the pipeline, the Reduce function. The partitioning stage of the framework guarantees that all pairs sharing the same key will be processed in the same Reduce task. In a similar way, a user-defined function is applied to the pairs, producing one output file per Reduce task, in the distributed file system.

One of the important advantages of the above schema is that the parallelization complexity is handled by the framework. The user only has to write the first-order functions that will be wrapped by the Map and Reduce functions. However, this advantage often comes with loss of flexibility. Each job must consist of exactly one Map function followed by an optional Reduce function, and steps cannot be executed in a different order. Moreover, if an algorithm requires multiple Map and Reduce steps, these can only be implemented as separate jobs, and data can only be transferred from one job to the next, through the file system.

In the initial implementations of Hadoop, MapReduce is designed as a master-slave architecture. The *JobTracker* is the master managing the cluster resources, scheduling jobs, monitoring progress and dealing with fault-tolerance. On each of the slave nodes, there exists a *TaskTracker* process, responsible for launching parallel tasks and reporting their status to the JobTracker. The slave nodes are statically divided into computing slots, available to execute either Map or Reduce tasks. The Hadoop community realized the limitations of this static model and recently redesigned the architecture to improve cluster utilization and scalability. The new design, YARN is presented in section II-C.

B. HDFS

HDFS [3] is the distributed file system used by the Hadoop project. Hadoop MapReduce jobs read their input data from HDFS and also write their output to it. HDFS has been very popular because of its scalability, reliability and capability of storing very large files.

There are two types of nodes in HDFS: the *DataNodes* and the *NameNode*. Typically, a Hadoop deployment has a single NameNode, which is the master and a set of DataNodes, which serve as slaves. The main responsibility of a DataNode is to store blocks of data and to serve them on request over the network. By default, data blocks are replicated in HDFS, for fault-tolerance and higher chance of data locality, when running MapReduce applications. The NameNode is unique in an HDFS cluster and is responsible for storing and managing metadata. It stores metadata in memory, thus limiting the number of files that can be stored by the system, according to the node's available memory.

C. YARN

YARN, *Yet Another Resource Negotiator*, is included in the latest Hadoop release and its goal is to allow the system to serve as a general data-processing framework. It supports programming models other than MapReduce, while also improving scalability and resource utilization. YARN makes no changes to the programming model or to HDFS. It consists of a re-designed runtime system, aiming to eliminate the bottlenecks of the master-slave architecture. The responsibilities of the JobTracker are split into two different processes, the *ResourceManager* and the *ApplicationMaster*. The ResourceManager handles resources dynamically, using the notion of *containers*, instead of static Map/Reduce slots. Containers are configured based on information about available memory, CPU and disk capacity. It also has a pluggable scheduler, which can use different strategies to assign tasks to available nodes. The ApplicationMaster is a framework-specific process, meaning that it allows other programming models to be executed on top of YARN, such as MPI or Spark [12]. It negotiates resources with the ResourceManager and supervises the scheduled tasks.

III. Hadoop/MapReduce Limitations

Even though YARN manages to overcome the well-known limitations of the Hadoop scheduling framework and improves the scalability and resource utilization, there still exist several opportunities for optimizations in Hadoop/MapReduce. Having studied the recent literature, we group the optimization opportunities in three main categories: performance issues, programming model extensions and usability enhancements. In this section, we discuss the limitations which lead to these optimization opportunities.

A. Performance Issues

Even though Hadoop/MapReduce has been praised for its scalability, fault-tolerance and capability of processing vast amounts of data, query execution time can often be several hours [13]. This is orders of magnitude higher than what modern DBMSs offer and prevents interactive analysis. Performance highly depends on the nature of the application, but is also influenced by inherent system characteristics and design choices. A quite large percentage of the execution time is spent in task initialization, scheduling, coordination and monitoring. Moreover, Hadoop/MapReduce does not support data pipelining or overlap of the Map and the Reduce phases. Data materialization for fault-tolerance and intensive disk I/O during the shuffling phase have also been found to significantly contribute to the overall execution time. It has been suggested that Hadoop performance would benefit from well-known optimization techniques, already used by database systems and query optimizers. Even though Hadoop lacks a built-in optimizer, many of the suggested techniques have been implemented in Hadoop extensions, as discussed in the next section. Optimizations include index creation [14], data co-location [15], reuse of previously computed results [16], exploiting sharing opportunities [17], mechanisms dealing with computational skew [18] and techniques allowing early approximate query results.

B. Programming Model Issues

Developing efficient MapReduce applications requires advanced programming skills and deep understanding of the system architecture. Common data analysis tasks usually include processing of multiple datasets and relational operations, such as joins, which are not trivial to implement in MapReduce. Therefore, the MapReduce programming model has been often characterized as too ”low-level” for analysts used to SQL-like or declarative languages. Another limitation of the programming model comes from its ”batch” nature. Data need to be uploaded to the file system and even when the same dataset needs to be analyzed multiple times, it has to be read every time. Also, the computation steps are fixed and applications need to respect the map-shuffle-sort-reduce sequence. Complex analysis queries are realized by chaining multiple MapReduce jobs, having the results of one serving as the input for the next. These characteristics make the model inappropriate for certain classes of algorithms. Various applications, including machine learning algorithms and graph processing, often require iterations or incremental computations. Since MapReduce operators are stateless, MapReduce implementations of iterative algorithms require manual management of state and chaining of iterations. Abstractions and high-level languages, have been built to facilitate MapReduce application development [19], [20]. Also, a set of domain-specific systems have emerged, extending the MapReduce programming model. We present these systems in section IV-B.

C. Configuration and Automation Issues

The third category of optimizations are related to automatic tuning and ease of use. There are numerous configuration parameters to set when deploying a Hadoop MapReduce cluster. Performance is often quite sensitive to them and users usually rely on empirical ”rules of thumb”. Options include the number of parallel tasks, the size of the file blocks and the replication factor. Proper tuning of these parameters requires knowledge of both available hardware and workload characteristics, while misconfiguration might lead to inefficient execution and underutilization of resources [13], [21]. Hadoop variations dealing with automatic tuning are discussed in Section IV-C.

IV. Hadoop/MapReduce Variations

Some of the optimizations discussed in this Section can have multiple effects, therefore, some of the presented systems could fall into more than one category. At this point, we need to stress that our categorization and comparison is based on the primary motivation of each system examined.

A. Performance Optimizations

Operator Pipelining and Online Aggregation: One of the first successful Hadoop extensions is MapReduce Online [22]. It improves performance by supporting online aggregation and stream processing, while also improving resource utilization. The motivation of MapReduce Online is to enable pipelining between operators, while preserving fault-tolerance guarantees. Pipelining is implemented both between tasks and between jobs. In the initial design, each reducer opens one TCP connection to each mapper. When a mapper computes a record, it determines to which partition it belongs and sends it via the appropriate socket. Opening a large number of TCP connections proved to be problematic, so the design was refined to use a “mixed” push/pull approach. Each reducer is allowed to open a bounded number of TCP connections, while pulling data from the rest of the mappers in the traditional Hadoop way. One problem that arises due to pipelining, is the nullification of the effect of combiners. To solve this problem, MapReduce Online buffers intermediate data up to a specified threshold, applies the combiner function on them and spills them to disk. As a side-effect of this design, early results of the jobs can be computed making approximate answers to queries available to users. This technique is called online aggregation and returns useful early results much faster than final results. Simply by applying the reduce function to the data that the reducer has seen so far, the system can provide an early snapshot. In combination to the job progress metrics, a user can appromixate the accuracy of the provided snapshot.

Approximate Results: A more sophisticated approach to approximate results in MapReduce is proposed by Laptev et al. [23]. The EARL library is a Hadoop extension which allows incremental computations of early results using sampling and the bootstrapping technique. An initial sample of the data is obtained and the error is estimated using bootstrapping. If the error is too high, the sample is expanded and the error recomputed. This process is repeated until the error is under a user-defined threshold. In order to implement EARL, Hadoop was extended to support dynamic input size expansion. First, pipelining between mappers and reducers was implemented, similarly to MapReduce Online, so that reducers can start processing data as soon as they become idle. Then, mappers are kept active and reused instead of being restarted in every iteration. This modification saves a significant amount of setup time. Finally, a communication channel was built between mappers and reducers, so that the termination condition can be easily tested. EARL is an addition to the MapReduce API and existing applications require modifications in order to exploit it.

Indexing and Sorting: Quite a few of the proposed optimizations for Hadoop/MapReduce come from well-known techniques of the database community. Long query runtimes are often caused due to lack of proper schemas and data indexing. Hadoop++ [14] and HAIL [24] are two remarkable attempts dealing with this matter. Hadoop++ is a transparent addition to Hadoop implemented using User Defined Functions (UDFs). It provides an indexing technique, the *Trojan Index*, which extends input splits with indexes at load time. Additionally to the Trojan Index, the paper also proposes a novel Join technique, the *Trojan Join*, which uses data co-partitioning in order to perform the join operation using only map tasks. HAIL proposes inexpensive index creation on Hadoop data attributes, in order to reduce execution times in exploratory use-cases of MapReduce. It modifies the upload pipeline of HDFS and creates a different clustered index per block replica. HAIL uses the efficient binary PAX representation [25] to store blocks and keeps each physical block replica in a different sort order. Sorting and indexing happen in-memory at upload time. If index information is available, HAIL also uses a modified version of the task scheduling algorithm of Hadoop, in order to schedule tasks to nodes with appropriate indexes and sort orders. The block binary representation and in-memory creation of indexes improves upload times for HDFS, while query execution times also greatly improve when index information is available. HAIL preserves Hadoop's fault-tolerance properties. However, failover times are sometimes higher, due to HAIL assigning more blocks per map task, therefore limiting parallelization during recovery. In a system with the default degree of replication, three different sort orders and indexes are available, greatly increasing the probability of finding a suitable index for the corresponding filtering attribute of the query. HAIL benefits queries with low selectivity, exploratory analysis of data and applications for which there exists adequate information for index creation.

Work Sharing: MRShare [17] is a Hadoop extension that aims to exploit sharing opportunities among different jobs. It transforms a batch of queries into a new batch, by forming an optimization problem and providing the optimal grouping of queries to maximize sharing opportunities. MRShare works on the following levels: sharing scans when the input to mapping pipelines is the same and sharing map outputs when the reducers will have to push each tuple to the correct reduce function. Hadoop was modified to support tagging of tuples and merge the tags into the keys of tuples, so that their origin jobs can be identified. Moreover, reducers were enabled to write to more than one output files.

Data Reuse: ReStore [16] is an extension to Pig [19], a high-level system built on top of Hadoop/MapReduce. It stores and reuses intermediate results of scripts, originating from complete jobs or sub-jobs. The input of ReStore is Pig's physical plan, i.e. a workflow of MapReduce jobs. ReStore maintains a repository where it stores job outputs together with the physical execution plan, the filename of the output in HDFS and runtime statistics about the MapReduce job that produced the output. The system consists of a plan matcher and rewriter which searches in the repository for possible matches and rewrites the job workflow to exploit stored data. It also has a sub-job enumerator and a sub-job selector, which are responsible for choosing which sub-job outputs to store, after a job workflow has been executed. Sub-job results are chosen to be stored in the repository based on the input to output ratio and the complexity of their operators. Repository garbage collection is not implemented, however guidelines for building one are proposed.

Skew Mitigation: SkewTune [18] is a transparent Hadoop extension providing mechanisms to detect stragglers and mitigate skew by repartitioning their remaining unprocessed input data. In order to decide when a task should be treated as a straggler, while avoiding unnecessary overhead and false-positives, SkewTune is using *Late Skew Detection*. Depending on the size of the remaining data, SkewTune may decide to scan the data locally or in parallel. In Hadoop, skew mitigation is implemented by SkewTune as a separate MapReduce job for each parallel data scan and for each mitigation. When repartitioning a map task, a map-only job is executed and the job tracker broadcasts all information about the mitigated map to all the reducers in the system. When repartitioning a reduce task, due to the MapReduce static pipeline inflexibility, an identity map phase needs to be run before the actual additional reduce task.

Data Colocation: The last system we present in this category is CoHadoop [15] and it allows applications to control where data are stored. In order to exploit its capabilities, applications need to state which files are related and might need to be processed together. CoHadoop uses this information to collocate files and improve job runtimes. While HDFS uses a random placement policy for load-balancing reasons, CoHadoop allows applications to set a new file property, in order for all copies of related files to be stored together. This property, the *locator*, is an integer and there is a N:1 relationship between files and locators, so that files with the same locator are stored on the same set of datanodes. The mapping is managed by saving information in a locator table, in the Namenode's memory. If the selected set of datanodes runs out of space, CoHadoop simply stores the files in another set of datanodes. CoHadoop may lead to skew in data distribution and also loss of more data in the presence of failures. Collocation and special partitioning are performed by adding a preprocessing step to a MapReduce job, which itself is a MapReduce job.

B. Programming model extensions

1) High-Level Languages: Developing applications using high-level languages on top of Hadoop has proven to be much more efficient regarding development time than using native MapReduce. Maintenance costs and bugs are also greatly reduced, as much less code is required. Pig [19] is one such high-level system that consists of a declarative scripting language, *Pig Latin*, and an execution engine that allows the parallel execution of data-flows on top of Hadoop. Pig offers an abstraction that hides the complexity of the MapReduce programming model and allow users to write SQL-like scripts, providing all common data operations (filtering, join, ordering, etc.).

One of the most widely-used high-level systems for Hadoop is Hive [20]. Initially developed by Facebook, Hive is not just an abstraction, but a data warehousing solution. It provides a way to store, summarize and query large amounts of data. Hive's high-level language, HiveQL, allows users to express queries in a declarative, SQL-like manner. Very similar to Pig, HiveQL scripts are compiled to MapReduce jobs and executed on the Hadoop execution engine.

Another popular query language is Jaql [26]. Jaql is less general than the systems we have introduced in this Section, as it is designed for quering semi-structured data in JSON format only. The system is extensible and supports parallelism using Hadoop. Although Jaql has been specifically designed for data in JSON format, it borrows a lot of characteristics from SQL, XQuery, LISP, and PigLatin.

Cascading [27] is a Java application framework that facilitates the development of data processing applications on Hadoop. It offers a Java API for defining and testing complex dataflows. It abstracts the concepts of map and reduce and introduces the concept of flows, where a flow consists of a data source, reusable pipes that perform operations on the data and data sinks. Cascading quickly gained popularity among the industry and Twitter even developed and open-sourced a Scala API for it, Scalding [28].

2) Domain-specific Systems:

Support for Iterations: Iterative algorithms are very common in data-intensive problems, especially in the domains of machine learning and graph processing. HaLoop [29], is a modified version of Hadoop, with built-in support for development and efficient execution of iterative applications. HaLoop offers a mechanism to cache and index *invariant* data between iterations, significantly reducing communication costs. It extends Hadoop's API, allowing the user to define loops and termination conditions easily. The authors also propose a novel scheduling algorithm, which is loop-aware and exploits inter-iteration locality. It exploits cached data in order to co-locate tasks which access the same data in different iterations.

Support for Incremental Computations: A special class of iterative applications is that of incremental computations. These include jobs which need to be run repeatedly with slightly different, most often augmented input. Performing such computations in MapReduce would obviously lead to redundant computations and inefficiencies. In order to overcome this problem, one has to specially design their MapReduce application to store and use state across multiple runs. Since MapReduce was not designed to reuse intermediate results, writing such programs is complex and error-prone. Incoop's [30] goal is to provide a transparent way to reuse results of prior computations, without demanding any extra effort from the programmer. Incoop extends Hadoop to support incremental computations, by making three important modifications: (a) Inc-HDFS. A modified HDFS which splits data depending on file contents instead of size. It provides mechanisms to identify similarities between datasets and opportunities for data reuse, while preserving compatibility with HDFS. (b) Contraction Phase. An additional computation phase added before the Reduce phase, used to control task granularity. This phase leverages the idea of Combiners to "break" the reduce task into a tree-hierarchy of smaller tasks. The process is run recursively until the last level, where the reduce function is applied. In order to result into a data partitioning suitable for reuse, content-based partitioning is again performed on every level of Combiners. (c) Memoization-aware Scheduler. An improved scheduler which takes into account data locality of previously computed results, while also using a work-stealing algorithm. The memoization-aware scheduler schedules tasks on the nodes that contain data which can be reused. However, this approach might create load imbalance if some data is very popular. To avoid this situation, the scheduler implements a simple work-stealing algorithm. When a node runs out of work, the scheduler will locate the node with the largest task queue and delegate a task to the idle node.

C. Automatic tuning

Self-Tuning: Configuring and tuning Hadoop MapReduce is usually not a trivial task for developers and administrators, often resulting to poor performance, resource underutilization and consequently increased operational costs. Starfish [21] is a self-tuning system, built as an extension to Hadoop, which dynamically configures system properties based on workload characteristics and user input. Starfish performs tuning on three levels. In the *job-level*, it uses a *Just-in-Time Optimizer* to choose efficient execution techniques, a *Profiler* to learn performance models and build job profiles and a *Sampler* to collect statistics about input, intermediate and output data and help the Profiler build approximate models. In the *workflow-level*, it uses a *Workflow-aware Scheduler*, which exploits data locality on the workflow-level, instead of making locally optimal decisions. A *What-if Engine* answers questions based on simulations of job executions. In the *workload-level*, Starfish consults the *Workload Optimizer* to find opportunities for data-flow sharing, materialize of intermediate results for reuse or reorganize jobs inside a batch and the *Elastisizer* to automate node and network configuration.

Disk I/O Minimization: Sailfish [31] is another Hadoop modification also providing auto-tuning opportunities, such as dynamically setting the number of reducers and handling skew of intermediate data. Additionally, it improves performance by reducing disk I/O due to intermediate data transfers. The proposed solution uses KFS [32] instead of HDFS, which is a distributed file system allowing concurrent modifications to multiple blocks of a single file. The authors propose *I-files*, an abstraction which aggregates intermediate data, so that they can be written to disk in batches. An index is built and stored with every file chunk and an offline daemon is responsible for sorting records within a chunk.

Data-aware Optimizations: Manimal [33] is an automatic optimization framework for MapReduce, transparent to the programmer. The idea is to apply well-known query optimization techniques to MapReduce jobs. Manimal detects optimization opportunities by performing static analysis of compiled code and only applies optimizations which are *safe*. The system's *analyzer* examines the user code and sends the resulting optimization descriptors to the *optimizer*. The optimizer uses this information and pre-computed indexes to choose an optimized execution plan, the *execution descriptor*. The *execution fabric* then executes the new plan in the standard map-shuffle-reduce fashion. Optionally, an index generation program creates an additional MapReduce job to generate an indexed version of the input data. Example optimizations performed by Manimal include *Selection* and *Projection*. In the first case, when the map function is a filter, Manimal uses a B+Tree to only scan the relevant portion of the input. In the second case, it eliminates unnecessary fields from the input records.

Table I shows a brief comparison of the systems discussed in this survey. We have excluded high-level languages from this comparison, since they share common goals and major characteristics among them.

V. Discussion

MapReduce is a quite recent paradigm and its open-source implementation, Hadoop, still has plenty of optimization opportunities to exploit. However, implementing even traditional optimization techniques can be very challenging in architectures of shared-nothing clusters of commodity machines. Scalability, efficiency and fault-tolerance are major requirements for any MapReduce framework and trade-offs between optimizations and these features need to be carefully studied.

One can identify several trends when studying the systems discussed in this survey. In contrast to traditional applications, MapReduce programs are data-intensive instead of computation-intensive and, in order to achieve good performance, it is vital to minimize disk I/O and communication. Therefore, many systems seek ways to enable in-memory processing and avoid reading from disk when possible. For the same reason, traditional database techniques, such as materialization of intermediate results, caching and indexing are also favored.

Another recurring theme in MapReduce systems is relaxation of fault-tolerance guarantees. The initial MapReduce design from Google assumed deployments in clusters of hundreds or even thousands of commodity machines. In such setups, failures are very common and strict fault-tolerance and recovery mechanisms are necessary. However, after the release of Hadoop, MapReduce has also been used by organizations much smaller than Google. Common deployments may consist of only tenths of machines [34], significantly decreasing failure rates. Such deployments can benefit from higher performance, by relaxing the fault-tolerance guarantees of the system. For example, one can avoid materialization of task results and allow pipelining of data. In this scenario, when a failure occurs, the whole job would have to be re-executed, instead of only the tasks running on the failed node.

Many important steps forward have been made since the launch of MapReduce and Hadoop, but several open issues still exist in the area. Even though it is clear that relaxing fault-tolerance offers performance gains, we believe that this issue needs to be further studied in the context of MapReduce. The trade-offs between fault-tolerance and performance need to be quantified. When these trade-offs have become clear, Hadoop could offer capabilities of tunable fault-tolerance to the users or provide automatic fault-tolerance adjustment mechanisms, depending on cluster and application characteristics.

Another open issue is clearly the lack of a standard benchmark or a set of typical workloads for comparing the different Hadoop implementations. Each system is evaluated using different datasets, deployments and set of applications. There have been some efforts in this direction [31], [35], but no complete solution has been introduced and no clear answer exists to what a ”typical” MapReduce workload would be.

As far as programming extensions are concerned, we believe that the main problem with all the specialized systems proposed is transparency to the developer. In our view, such programming extensions need to be smoothly integrated into to the framework, so that existing applications can benefit from the optimizations, automatically, without having to change or re-compile the source code.

Finally, even if successful declarative-style abstractions exist, Hadoop MapReduce is still far from offering inter-

	Optimization Type	Major Contributions	Open-Source / Available to use	Transparent to Existing Applications
MapReduce Online	performance, programming model	Pipelining, Online aggregation	yes	yes
EARL	performance	Fast approximate query results	yes	no
Hadoop++	performance	Performance gains for relational operations	no	yes
HAIL	performance	Performance gains for relational operations	no	no
MRShare	performance	Concurrent work sharing	no	no
ReStore		Reuse of previously computed results	no	yes
SkewTune	performance	Automatic skew mitigation	no	yes
CoHadoop	performance	Communication minimization by data co-locations	no	no
HaLoop	programming model	Iteration support	yes	no
Incoop	programming model	Incremental processing support	no	no
Starfish	tuning, performance	Dynamic self-tuning	no	yes
Sailfish	tuning, performance	Disk I/O minimization and automatic tuning	no	yes
Manimal	tuning, performance	Automatic data-aware optimizations	no	yes

Table I
COMPARATIVE TABLE OF HADOOP VARIATIONS

active analysis capabilities. Developing common analysis tasks and declarative queries has indeed been significantly facilitated. However, these high-level systems still compile their queries into MapReduce jobs, which are executed on top of Hadoop. According to our judgment, these systems could greatly benefit from more sophisticated query optimization techniques. Mechanisms such as data reuse and approximate answers should also be more extensively studied and exploited in high-level systems.

Unfortunately, the majority of the proposed systems are not open-source or even available to use. This prevents researchers from studying or extending them and stalls progress. Also, proposed systems usually only compare to vanilla Hadoop, not yielding very interesting results. Consequently, very few of the optimizations proposed have been incorporated to official Hadoop releases.

VI. CONCLUSIONS

In conclusion, Big Data systems and specifically MapReduce, are an active research area, still at its infancy. Currently, the interest for MapReduce is at its peak and there exist a lot of problems and challenges to be addressed. There lies a bright future ahead for Big Data, as businesses and organizations realize more and more the value of the information they can store and analyze. Developing ways to process the vast amounts of data available drives business innovation, health discoveries, science progress and allows us to find novel ways to solve problems, which we considered very hard or even impossible in the past.

ACKNOWLEDGMENT

This work was supported in part by the Erasmus Mundus Joint Doctorate in Distributed Computing (EMJD-DC) funded by the Education, Audiovisual and Culture Executive Agency (EACEA) of the European Commission under the FPA 2012-0030, and in part by the End-to-End Clouds project funded by the Swedish Foundation for Strategic Research (SSF) under the contract RIT10-0043.

REFERENCES

[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[2] "Hadoop: Open-Source implementation of MapReduce," http://hadoop.apache.org, [Online; Last accessed Jan 2013].

[3] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, ser. MSST '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–10.

[4] "MapReduce: A Major Step Backwards," http://homes.cs.washington.edu/billhowe/mapreduce_a_major_step_backwards.html, [Online; Last accessed Jan 2013].

[5] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker, "A comparison of approaches to large-scale data analysis," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, ser. SIGMOD '09. New York, NY, USA: ACM, 2009, pp. 165–178.

[6] K.-H. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, and B. Moon, "Parallel data processing with mapreduce: a survey," *SIGMOD Rec.*, vol. 40, no. 4, pp. 11–20, Jan. 2012.

[7] D. Jiang, B. C. Ooi, L. Shi, and S. Wu, "The performance of mapreduce: an in-depth study," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 472–483, Sep. 2010.

[8] A. Goyal and S. Dadizadeh, "A survey on cloud computing," University of British Columbia, Tech. Rep., 2009.

[9] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, ser. EuroSys '07. ACM, 2007, pp. 59–72.

[10] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, ser. HotCloud'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 10–10.

[11] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke, "Nephele/pacts: a programming model and execution framework for web-scale analytical processing," in *Proceedings of the 1st ACM symposium on Cloud computing*, ser. SoCC '10. New York, NY, USA: ACM, 2010, pp. 119–130.

[12] "PoweredByYarn," http://wiki.apache.org/hadoop/PoweredByYarn, [Online; Last accessed Jan 2013].

[13] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan, "An analysis of traces from a production mapreduce cluster," in *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, ser. CCGRID '10. IEEE Computer Society, 2010, pp. 94–103.

[14] J. Dittrich, J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad, "Hadoop++: making a yellow elephant run like a cheetah (without it even noticing)," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 515–529, Sep. 2010.

[15] M. Y. Eltabakh, Y. Tian, F. Özcan, R. Gemulla, A. Krettek, and J. McPherson, "Cohadoop: flexible data placement and its exploitation in hadoop," *Proc. VLDB Endow.*, vol. 4, no. 9, pp. 575–585, Jun. 2011.

[16] I. Elghandour and A. Aboulnaga, "Restore: reusing results of mapreduce jobs," *Proc. VLDB Endow.*, vol. 5, no. 6, pp. 586–597, Feb. 2012.

[17] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas, "Mrshare: sharing across multiple queries in mapreduce," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 494–505, Sep. 2010.

[18] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, "Skewtune: mitigating skew in mapreduce applications," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '12. ACM, 2012, pp. 25–36.

[19] A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava, "Building a high-level dataflow system on top of map-reduce: the pig experience," *Proc. VLDB Endow.*, vol. 2, no. 2, pp. 1414–1425, Aug. 2009.

[20] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: a warehousing solution over a map-reduce framework," *Proc. VLDB Endow.*, vol. 2, no. 2, pp. 1626–1629, Aug. 2009.

[21] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu, "Starfish: A self-tuning system for big data analytics," in *In CIDR*, 2011, pp. 261–272.

[22] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, "Mapreduce online," in *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, ser. NSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 21–21.

[23] N. Laptev, K. Zeng, and C. Zaniolo, "Early accurate results for advanced analytics on mapreduce," *Proc. VLDB Endow.*, vol. 5, no. 10, pp. 1028–1039, Jun. 2012.

[24] J. Dittrich, J.-A. Quiané-Ruiz, S. Richter, S. Schuh, A. Jindal, and J. Schad, "Only aggressive elephants are fast elephants," *Proc. VLDB Endow.*, vol. 5, no. 11, pp. 1591–1602, Jul. 2012.

[25] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis, "Weaving relations for cache performance," in *Proceedings of the 27th International Conference on Very Large Data Bases*, ser. VLDB '01. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001, pp. 169–180.

[26] K. S. Beyer, V. Ercegovac, R. Gemulla, A. Balmin, M. Y. Eltabakh, C. C. Kanne, F. Özcan, and E. J. Shekita, *PVLDB*, vol. 4, no. 12, pp. 1272–1283, 2011.

[27] "Cascading," http://www.cascading.org/, [Online; Last accessed 2012].

[28] "Scalding," https://dev.twitter.com/blog/scalding, [Online; Last accessed 2012].

[29] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "Haloop: efficient iterative data processing on large clusters," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 285–296, Sep. 2010.

[30] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquin, "Incoop: Mapreduce for incremental computations," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*, ser. SOCC '11. New York, NY, USA: ACM, 2011, pp. 7:1–7:14.

[31] S. Rao, R. Ramakrishnan, A. Silberstein, M. Ovsiannikov, and D. Reeves, "Sailfish: a framework for large scale data processing," in *Proceedings of the Third ACM Symposium on Cloud Computing*, ser. SoCC '12. New York, NY, USA: ACM, 2012, pp. 4:1–4:14.

[32] "Kosmos distributed file system," http://code.google.com/p/kosmosfs/, [Online; Last accessed Jan 2013].

[33] E. Jahani, M. J. Cafarella, and C. Ré, "Automatic optimization for mapreduce programs," *Proc. VLDB Endow.*, vol. 4, no. 6, pp. 385–396, Mar. 2011.

[34] "PoweredByHadoop," http://wiki.apache.org/hadoop/PoweredBy, [Online; Last accessed Jan 2013].

[35] Y. Chen, S. Alspaugh, and R. Katz, "Interactive analytical processing in big data systems: a cross-industry study of mapreduce workloads," *Proc. VLDB Endow.*, vol. 5, no. 12, pp. 1802–1813, Aug. 2012.