Speedup Query Processing in Hadoop Using Mapreduce Framework

Chandra Shekhar Gautam[1], Akhilesh A. Waoo[2*]

[1] Rajiv Gandhi College of Computer Application and Technology, Satna, M.P., India
[2] AKS University, Satna (M.P.), India

Email Address

shekharg84@gmail.com (Chandra Shekhar Gautam), akhileshwaoo@gmail.com (Akhilesh A. Waoo)
*Correspondence: akhileshwaoo@gmail.com

Received: 28 December 2017**; Accepted:** 20 January 2018**; Published:** 19 March 2018

Abstract:

The Internet used by 3.2 billion people in 2015. Nearly half of the global population will be using the internet by the end of this year, according to a new report. Enterprises today gain vast volumes of data from different sources and influence this information by means of data analysis to support effective decision-making and provide new functionality and services. The key requirement of data analytics is scalability, simply due to the immense volume of data that need to be extracted, processed, and analyzed in a timeline fashion. Possibly the most popular framework for current large-scale data analytics is Map-Reduce, mainly due to its salient features that include scalability, fault-tolerance, ease of programming, and edibility. However, despite its merits, MapReduce has evident performance limitations in miscellaneous analytical tasks, and this has given rise to a significant body of research that aim at improving its efficiency, while maintaining its desirable properties. The aims of this review the state-of-the-art in improving the performance of parallel query processing using MapReduce. A set of the most significant weaknesses and limitations of Map-Reduce is discussed at a high level, along with solving techniques. Taxonomy is presented for categorizing existing research on MapReduce improvements according to the specific problem they target. Based on the proposed taxonomy, a classification of existing research is provided focusing on the optimization objective. Concluding, this research article outlines interesting directions for future parallel data processing systems.

Keywords:

Hadoop, Mapreduce, Speed up Query, Performance

1. Introduction

In the time of "Big Data", characterized by the unparalleled volume of data, the velocity of data generation, and the variety of the structure of data, support for large-scale data analytics constitutes a particularly challenging task. To address the scalability requirements of today's data analytics, parallel shared-nothing architectures of commodity machines (often consisting of thousands of nodes) have been lately

established as the de-facto solution [1,2]. Various systems have been developed mainly by the industry to support Big Data analysis, including Google's MapReduce, Yahoo's PNUTS, Microsoft's SCOPE, Twitter's Storm, LinkedIn's Kafka and Wal-Mart-Labs' Muppet Also, several companies, including facebook [3,4] both use and have contributed to Apache Hadoop (an open-source implementation of MapReduce) and its ecosystem.

Zan Mo, Li., [5] MapReduce has become the most popular framework for large-scale processing and analysis of vast datasets in clusters of machines, mainly because of its simplicity. With MapReduce, the developer gets various cumbersome tasks of distributed programming for free without the need to write any code; indicative examples include a machine to machine communication, task scheduling to machines, scalability with cluster size, ensuring availability, handling failures, and partitioning of input data. Moreover, the open-source Apache Hadoop implementation of MapReduce has contributed to its widespread usage both in industry and academia. As a witness to this trend, counting of the number of papers are performed related to MapReduce and cloud computing published yearly in major national/international database conferences. It has been reported a significant increase from 12 in 2008 to 69 papers in 2012.

2. MapReduce

MapReduce is a framework [6] for parallel processing of massive data sets. A job to be performed using the MapReduce framework must be specified as two phases: the map phase as specified by a Map function (also called map per) takes key/value pairs as input, possibly performs some computation on this input, and produces intermediate results in the form of key/value pairs; and the reduce phase which processes these results as specified by a Reduce function (also called reducer). The data from the map phase are shuffled, i.e., exchanged and merge-sorted, to the machines performing the reduce phase. It should be noted that the shuffle phase can itself be more time-consuming than the two others depending on network bandwidth availability and other resources.

2.1 Input Reader

The input reader in the basic form takes input from files (large blocks) and converts them to key/value pairs. It is possible to add sup-port for other input types, so that input data can be retrieved from a database or even from main memory. The data are divided into splits, which are the unit of data processed by a map task [7,8]. A typical split size is the size of a block, which for example in HDFS is 64 MB by default, but this is conquerable.

2.2 Map Function

A map task takes as input a key/value pair from the input reader, performs the logic of the Map function on it, and outputs the result as a new key/value pair. The results from a map task are initially output to a main memory buffer and when almost full spill to disk. The spill files are in the end merged into one sorted file.

2.3 Combiner Function

This optional function is provided for the common case when there is -

(1) Significant repetition in the intermediate keys produced by each map task, and

(2) The user specified Reduce function is commutative and associative.

In this case, a Combiner function will perform partial Reduction so that pairs with same key will be processed as one group by a reduce task.

2.4 Partition Function

As default, a hashing function is used to partition the intermediate keys output from the map tasks to reduce tasks. While this in general provides good balancing, in some cases it is still useful to employ other partitioning functions, and this can be done by providing a user-defined Partition function.

2.5 Reduce Function

The Reduce function is invoked once for each distinct key and is applied on the set of associated values for that key, i.e., the pairs with same key will be processed as one group. The input to each reduce task is guaranteed to be processed in increasing key order. It is possible to provide a user specified comparison function to be used during the sort process.

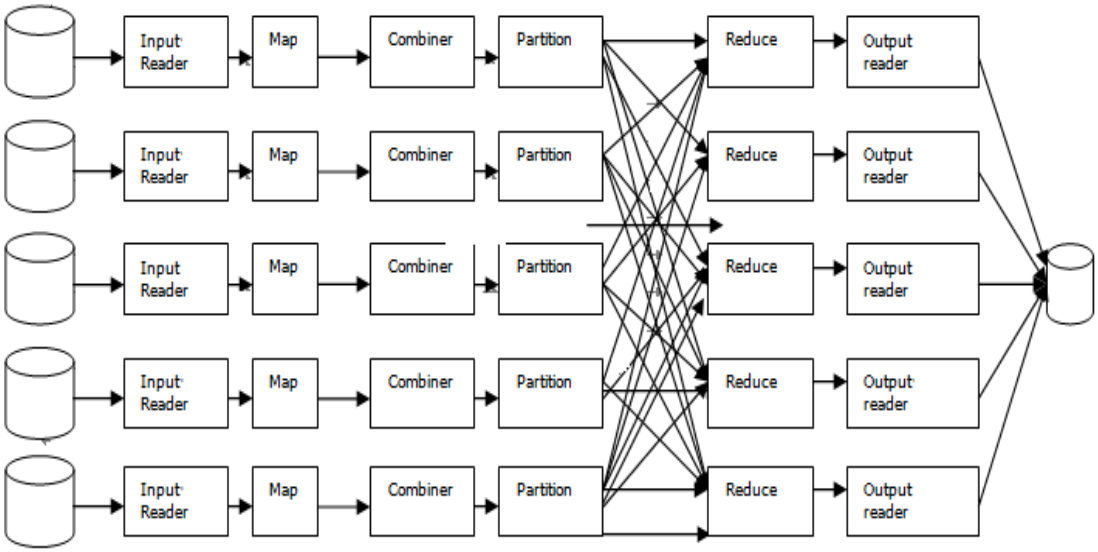

Figure 1. *Steps of Data Processing in MapReduce.*

2.6. Output Writer

The output writer is responsible for writing the output to stable storage. In the basic case, this is to a file; however, the function can be modified so that data can be stored in, e.g., a database. In more detail, the data are processed through the following 6 steps as illustrated in Figure 1.

3. Weaknesses and Limitations

Despite its evident merits, MapReduce often fails to exhibit acceptable performance for various processing tasks. Quite often this is a result of weaknesses related to the nature of MapReduce or the applications and use-cases it was originally designed for. [8] In other cases, it is the product of limitations of the processing model adopted in Map- Reduce. This paper identifies a list of issues; refer Table 1 related to large-scale data processing in MapReduce/Hadoop that significantly impacts its efficiency.

Table 1. *List of Issues Related to Large-Scale Data Processing in MapReduce/Hadoop.*

Weakness	Technique
Access to input data	Indexing and data layouts
High communication cost	Partitioning and collocation
Redundant and wasteful processing	Result sharing, batch processing of queries and incremental processing
Re computation	Materialization
Lack of early termination	Sampling and sorting
Lack of iteration	Loop-aware processing, caching, pipelining, recursion, incremental processing
Quick retrieval of approximate results	Data summarization and sampling
Load balancing	Pre-processing, approximation of the data distribution and repartitioning
Lack of interactive or real-time processing	In-memory processing, pipelining, streaming and pre-computation
Lack of support for n-way operations	Additional MR phase(s), re-distribution of keys and record duplication

4. MapReduce Improvements

In this section, an overview is provided of various methods and techniques present in the existing literature for improving the performance of MapReduce. All approaches are categorized based on the introduced improvement. The categories of MapReduce improvements in taxonomy, is organized in Table 2.

Table 2. *Categories of Techniques for improved processing in MapReduce.*

Existing Approaches	Optimization Objectives
Data Access	Indexing, Intentional Data Placement, Data Layouts
Avoidance of Redundant Processing	Incremental processing, Batch processing of Queries
Early Termination	Sampling and sorting
Iterative Processing	Looping, Caching, Pipelining, Recursion
Query Optimization	Processing Optimization Dataflow Optimization, Parameter Tuning
Fair Work Allocation	Preprocessing sampling, Batching

In above Table 2 classifies existing approaches for improved processing based on their optimization objectives.

4.1. Data Access

Data Access efficient access to data is an essential step for achieving improved performance during query processing. We identify three subcategories of data access, namely indexing, intentional data placement, and data layouts.

4.1.1. Indexing

Hadoop++ [9]is a system that provides indexing functionality for data stored in HDFS by means of User Defined Functions (UDFs), i.e., without modifying the Hadoop framework at all. The indexing information (called Trojan Indexes) is injected into logical input splits and serves as a cover index for the data inside the split. Moreover, the index is created at load time, thus imposing no overhead in query processing. Hadoop++ also supports joins by co-partitioning data and collocating

them at load time. Intuitively, this enables the join to be processed at the map side, rather than at the reduce side (which entails expensive data transfer/shuffling in the network). Hadoop++ has been compared against HadoopDB [10]. Improves the long index creation times of Hadoop++, by exploiting the n replicas (typically n=3) maintained by default by Hadoop for fault-tolerance and by building a different clustered index for each replica.

At query time, the most suitable index to the query is selected, and the replica of the data is scanned during the map phase. As a result, improves substantially the performance of MapReduce processing, since the probability of finding a suitable index for efficient data access is increased. In addition, the creation of the indexes occurs during the data upload phase to HDFS (which is I/O bound), by exploiting "unused CPU ticks", thus it does not affect the upload time significantly. Given the availability of multiple indexes, choosing the optimal execution plan for the map phase is an interesting direction for future work.

4.1.2. Intentional Data

Placement CoHadoop collocates and copartitions data on nodes intentionally, so that related data are stored on the same node. To achieve colocation, CoHadoop extends HDFS with a file-level property (locator), and files with the same locator are placed on the same set of Data Nodes. In addition, a new data structure (locator table) is added to the Name Node of HDFS. This is depicted in the example of Figure 4 using a cluster of five nodes (one Name Node and four Data Nodes) where 3 replicas per block are kept. All blocks (including replicas) of files A and B are collocated on the same set of Data Nodes, and this is described by a locator present in the locator table (shown in bold) of the Name Node [11].

The contributions of CoHadoop include colocation and copartitioning, and it targets a specific class of queries that benefit from these techniques. For example, joins can be efficiently executed using a map-only join algorithm, thus eliminating the overhead of data shuffling and the reduce phase of MapReduce. However, applications need to provide hints to CoHadoop about related files, and therefore one possible direction for improvement is to automatically identify related files [12]. CoHadoop is compared against Hadoop++, which also supports copartitioning and collocating data at load time, and demonstrates superior performance for join processing.

4.1.3. Data Layouts

A nice detailed overview of the exploitation of different data layouts in MapReduce is presented in [15]. We provide a short overview in the following. The use of a columnar file (called CFile) for data storage. [14] The idea is that data are partitioned in vertical groups; each group is sorted based on a selected column and stored in column-wise format in HDFS. This enables selective access only to the columns used in the query. In consequence, more efficient access to data than traditional row-wise storage is provided for queries [16] that involve a small number of attributes. Cheetah [17] also employs data storage in columnar format and applies different compression techniques for different types of values appropriately. In addition, each cell is further compressed after it is created using GZIP. Cheetah employs the PAX layout [13] at the block level, so each block contains the same set of rows as in row-wise storage, only inside the block column layout is employed. Compared to Llama, the important benefit of Cheetah is that all data that belong to a record are stored in the same block, thus avoiding expensive network access (as in the case of CFile).

4.2 Avoiding Redundant Processing

MRShare is a sharing framework that identifies different queries (jobs) that share portions of identical work. Such queries do not need to be recomputed each time from scratch [18]. The focus of this work is to save I/O, and therefore, sharing opportunities are identified in terms of sharing scans and sharing map output. MRShare transforms sets of submitted jobs into groups and treat each group as a single job, by solving an optimization problem with objective to maximize the total savings.

To process a group of jobs as a single job, MRShare modifies Hadoop to

a. Tag map output tuples with tags that indicate the tuple's originating jobs, and

b. Write to multiple output files on the reduce side.

Open problems include extending MRShare to support jobs that use multiple inputs (e.g., joins) as well as sharing parts of the Map function.

4.3. Early Termination

Sampling based [19] on predicates raises the issue of lack of early termination of map tasks even when a job has read enough input to produce the required output. The problem addressed by is how to produce a fixed-size sample (that satisfies a given predicate) of a massive data set using MapReduce. To achieve this objective, some technical issues need to be addressed [20] on Hadoop, and thus two new concepts are defined. A new type of job is introduced, called dynamic, which can dynamically control its data access. In addition, the concept of Input Provider is introduced in the execution model of Hadoop. The Input Provider is provided by the job together with the Map and Reduce logic. Its role is to make dynamic decisions about the access to data by the job [21]. At regular intervals, the Job Client provides statistics to the Input Provider, and based on this information, the Input Provider can respond in three different ways:

(1) "End of input", in which case the running map tasks can complete, but no new map tasks are invoked, and the shuffle phase is initiated,

(2) "Input available", which means that additional input needs to be accessed, and

(3) "Input unavailable", which indicates that no decision can be made at this point and processing should continue as normal until the next invocation of Input Provider.

4.4. Iterative Processing

P. S. Rani et. al. [22] explain the straightforward way of implementing iteration is to use an outsider driver program to control the execution of loops and launch new MapReduce jobs in each iteration. For example, this is the approach followed by Mahout. In the following, we review iterative processing using looping constructs, caching and pipelining, recursive queries, and incremental iterations.

4.5. Query Optimization

Processing Optimizations HadoopDB [23] is a hybrid system, aiming to exploit the best features of MapReduce and parallel DBMSs. The basic idea behind HadoopDB is

to install a database system on each node and connect these nodes by means of Hadoop as the task coordinator and network communication layer. Query processing on each node is improved by assigning as much work as possible to the local database. In this way, one can harvest all the benefits of query optimization provided by the local DBMS.

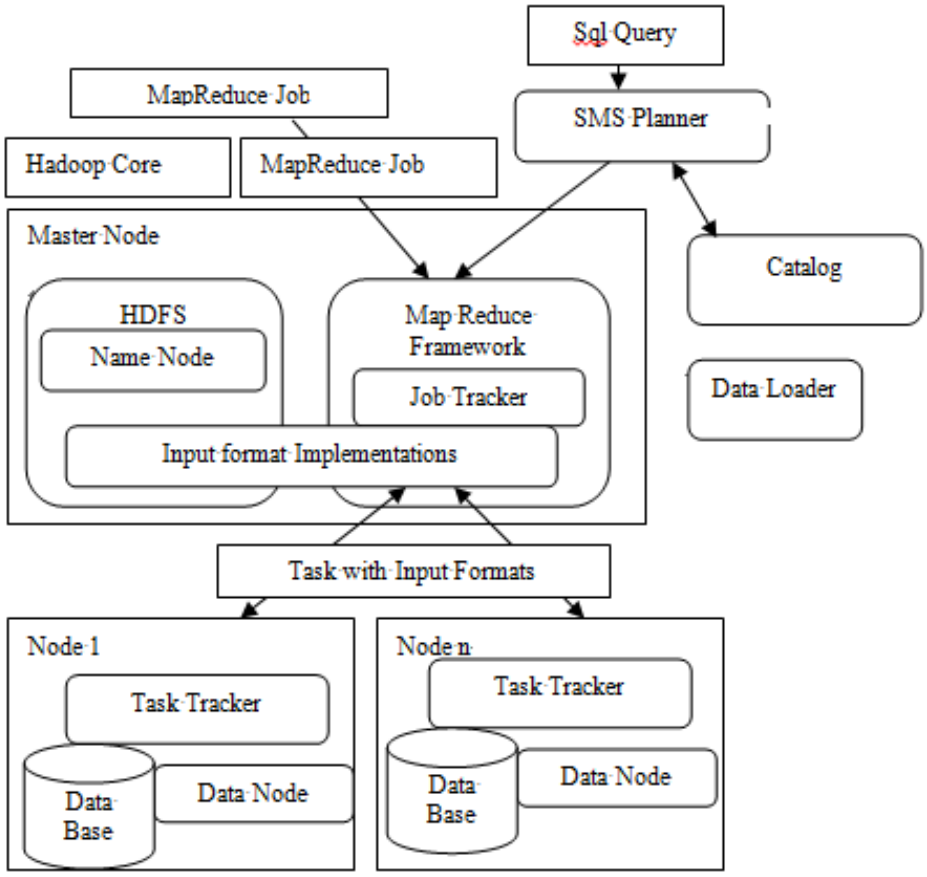

Figure 2. *The Architecture of Hadoop DB.*

Particularly, in the case of joins, where Hadoop does not support data colocation, a significant amount of data needs to be repartitioned by the join key to be processed by the same Reduce process [24]. In HadoopDB, when the join key matches the partitioning key, part of the join can be processed locally by the DBMS, thus reducing substantially the amount of data shuffled.

The architecture of HadoopDB is depicted in Figure 2 where the newly introduced components are marked with bold lines and text [25]. The database connector is an interface between database systems and Task Trackers. It connects to the database, executes a SQL query, and returns the result in the form of key-value pairs. The catalog maintains metadata about the databases, such as connection parameters as well as metadata on data sets stored, replica locations, data partitioning properties [26]. The data loader globally repartitions data based on a partition key during data upload, breaks apart single node data into smaller partitions, and bulk loads the databases with these small partitions. Finally, the SMS planner extends Hive and produces query plans that can exploit features provided by the available database systems [27].

For example, in the case of a join, some tables may be collocated, thus the join can be pushed entirely to the database engine (like a map-only job).

4.6. Fair Work Allocation

Since reduce tasks work in parallel, an overburdened reduce task may stall the completion of a job. To assign the work fairly, techniques such as pre-processing and sampling, repartitioning, and batching are employed.

4.6.1. Pre-processing and Sampling

Load balancing in MapReduce [28] in the context of entity resolution, where data skew may assign workload to reduce tasks unfairly. This problem naturally occurs in other applications that require pairwise computation of similarities when data is skewed, and thus it is considered a built-in vulnerability of MapReduce. Two techniques (Block Split and Pair Range) are proposed, and both rely on the existence of a pre-processing phase (a separate MapReduce job) that builds information about the number of entities present in each block. Ramakrishna et al. aim at solving the problem of reduces keys with large loads [29]. This is achieved by splitting large reduce keys into several medium-load reduce keys, and assigning medium-load keys to reduce tasks using a bin-packing algorithm. Identifying such keys is performed by sampling before the MapReduce job starts, and information about reduce-key load size (large and medium keys) is stored in a partition file.

4.6.2. Repartitioning

Handling data skew by means of an adaptive load balancing strategy [30]. A cost estimation method is proposed to quantify the cost of the work assigned to reduce tasks, to ensure that this is performed fairly. By means of computing local statistics in the map phase and aggregating them to produce global statistics, the global data distribution is approximated [31]. This is then exploited to assign the data output from the map phase to reduce tasks in a way that achieves improved load balancing.

5. Conclusions

MapReduce has brought new excitement in the parallel data processing landscape. This is due to its salient features that include scalability, fault-tolerance, simplicity, and edibility. Still, several of its shortcomings hint that MapReduce is not perfect for every large-scale analytical task. It is therefore natural to ask ourselves about lessons learned from the MapReduce experience and to hypothesize about future frameworks and systems.

The overall believe is that the next generation of parallel data processing systems for massive data sets should combine the merits of existing approaches. The strong features of MapReduce clearly need to be retained; how-ever, they should be coupled with efficiency and query optimization techniques present in traditional data management systems. Hence, future systems will not extend MapReduce, but instead redesign it from scratch, to retain all desirable features but also introduce additional capabilities.

Conflicts of Interest

The authors declare that there is no conflict of interest regarding the publication of this article

References

[1] Abadi, D.J. Data management in the cloud: Limitations and opportunities. *IEEE Data Engineering Bulletin.* 2009, 32(1), 3-12.

[2] Afrati, F.N.; Borkar, V.R.; Carey, M.J.; Polyzotis, N.; Ullman J.D. Map-reduce extensions and recursive queries. *In Proceedings of International Conference on Extending Database Technology (EDBT),* 2011, 1-8.

[3] Aiyer, A.S.; Bautin, M.; Chen, G.J.; Damania, P.; Khemani, P.; Muthukkaruppan, K.; Ranganathan, K.; Spiegelberg, N.; Tang, L.; Vaidya, M. Storage infrastructure behind Facebook Messages: using HBase at scale. *IEEE Data Engineering Bulletin.* 2012, 35(2), 4-13.

[4] Afrati, F.N.; Ullman, J.D. Optimizing joins in a Map-Reduce environment. *In Proceedings of International Conference on Extending Database Technology (EDBT),* 2010, 99-110.

[5] Zan, M.; Li. Research of Big Data Based on the Views of Technology and Application, 2015, 5, 192-197.

[6] Rani, S.; Rama, B. MapReduce with Hadoop for Simplified Analysis of Big Data. *International Journal of Advanced Research in Computer Science,* 2017, 8(5), 853-856.

[7] Agarwal, S.; Kandula, S.; Bruno, N.; Wu, M.C.; Stoica, I.; Zhou, J. Re-optimizing data-parallel computing. In Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2012, 21, 1-14.

[8] Agarwal, S.; Panda, A.; Mozafari, B.; Milner, H.; Madden, S.; Stoica, I. BlinkDB: queries with bounded errors and bounded response times on very large data. In Proceedings of European Conference on Computer systems (EuroSys), 2013.

[9] Agrawal, D.; Das, S.; Abbadi, A. E. Big Data and cloud computing: current state and future opportunities. In Proceedings of International Conference on Extending Database Technology (EDBT), 2011, 530-533.

[10] Joseph, C.W.; Pushpalatha, B. A Survey on Big Data and Hadoop, International Journal of Innovative Research in Computer and Communication Engineering. ISSN(Online): 2320-9801, March 2017, 5(3), 5525-5530.

[11] Afrati, F.N.; Sarma, A.D.; Menestrina, D.; Parameswaran, A.G.; Ullman, J.D. Fuzzy joins using MapReduce. In Proceedings of International Conference on Data Engineering (ICDE), 2012, 498-509.

[12] Siddaraju; Sowmya, C.; Rashmi, K.; Rahul, M. Efficient Analysis of Big Data Using Map Reduce Framework, International Journal of Recent Development in Engineering and Technology, 2014, 2(6), 64-68.

[13] Ananthanarayanan, G.; Ghodsi, A.; Wang, A.; Borthakur, D.; Kandula, S.; Shenker, S.; Stoica, I. PACMan: coordinated memory caching for parallel jobs. In Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2012, 20, 1-14.

[14] Bhatotia, P.; Wieder, A.; Rodrigues, R.; Acar, U.A.; Pasquin, R. Incoop: MapReduce for incremental computations. In ACM Symposium on Cloud Computing (SoCC), 2011, 7, 1-14.

[15] Dittrich, J.; Quian′e-Ruiz, J.A. Efficient Big Data processing in Hadoop MapReduce. Proceedings of the VLDB Endowment (PVLDB), 2012, 5(12), 2014-2015.

[16] Chen, S. Cheetah: a high performance, custom data warehouse on top of MapReduce. Proceedings of the VLDB Endowment (PVLDB), 2010, 3(2), 1459-1468.

[17] Borthakur, D.; Gray, J.; Sarma, J.; Muthukkaruppan, K.; Spiegelberg, N.; Kuang, H.; Ranganathan K.; Molkov D.; Menon A.; Rash S.; Schmidt R.; Aiyer A.S. Apache Hadoop goes realtime at Facebook. In Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD), 2011, 1071-1080.

[18] Borthakur, D.; Gray, J.; Sarma, J.; Muthukkaruppan, K.; Spiegelberg, N.; Kuang, H.; Ranganathan K.; Molkov D.; Menon A.; Rash S.; Schmidt R.; Aiyer A.S. Apache Hadoop goes realtime at Facebook. In Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD), 2011, 1071-1080.

[19] Bu, Y.; Howe, B.; Balazinska, M.; Ernst, M.D. The HaLoop approach to large-scale iterative data analysis. *VLDB Journal*. 2012, 21(2), 169-190.

[20] Grover, R.; Carey, M.J. Extending map-reduce for efficient predicate-based sampling. *In Proceedings of International Conference on Data Engineering (ICDE)*, 2012, 486-497.

[21] Cattell, R. Scalable SQL and NoSQL data stores. *SIGMOD Record*, 2010, 39(4), 12-27.

[22] Rani, P.S.; Shalini, S.; Rukmani J.; Shanthini A. Energy Efficient Scheduling of Map Reduce for Evolving Big Data Applications. *International Journal of Advanced Research in Computer and Communication Engineering*, 2016, 5(2), 54-58.

[23] Ewen, S.; Tzoumas, K.; Kaufmann, M.; Markl, V. Spinning fast iterative data flows. *Proceedings of the VLDB Endowment (PVLDB)*, 2012, 5(11), 1268-1279.

[24] Chattopadhyay, B.; Lin, L.; Liu, W.; Mittal, S.; Aragonda, P.; Lychagina, V.; Kwon, Y.; Wong, M. Tenzing a SQL implementation on the MapReduce framework. *Proceedings of the VLDB Endowment (PVLDB)*, 2011, 4(12), 1318-1327.

[25] Yang, H.C.; Dasdan, A. Hsiao, R.L.; Parker, D.S. Map-reduce-merge: simplified relational data processing on large clusters. *SIGMOD'07,* 2007, 1029-1040.

[26] Condie, T.; Conway, N.; Alvaro, P.; Hellerstein, J.M.; Elmeleegy K.; Sears R. MapReduce online. (NSDI), 2010.

[27] Doulkeridis, C.; Nørv˚ag, K. On saying "enough already!" (Cloud-I), 2012.

[28] Bu, Y.; Borkar, V.R.; Carey, M.J.; Rosen, J.; Polyzotis, N.; Condie, T.; Weimer, M.; Ramakrishnan, R. Scaling datalog for machine learning on big data. The Computing Research Repository (CoRR), abs/1203.0160, 2012,

[29] Borkar, V.R.; Carey, M.J.; Grover, R.; Onose, N.; Vernica, R. Hyracks: a flexible and extensible foundation for data-intensive computing. *In Proceedings of International Conference on Data Engineering (ICDE)*, 2011, 1151-1162.

[30] Goodhope, K.; Koshy, J.; Kreps, J.; Narkhede, N.; Park, R.; Rao, J.; Ye, V.Y. Building LinkedIn's real-time activity data pipeline. *IEEE Data Engineering Bulletin,* 2012, 35(2), 33-45.

[31] Candan, K.S.; Kim, J.W.; Nagarkar, P.; Nagendra, M.; RanKloud, R.Y. Scalable multimedia data processing in server clusters. *IEEE MultiMedia,* 2011, 18(1), 64-77.