Incremental Map Reduce Framework for Efficient Mining Evolving in Big Data Environment

Lekha.L[1] Dhanya Krishnan P.K[2] Hasmitha.D[3] Kalpana A.V[4]
[1,2,3]U.G. Student [4]Assistant Professor
[1,2,3,4]Department of Computer Science Engineering
[1,2,3,4]Velammal Institute of Technology, Tamil Nadu, India

Abstract— Keeping in mind the constant growth of data, we require more efficient ways to store and retrieve data. As the data mining techniques that are currently being used become ineffective, new methods or some modifications in the application of the existing techniques becomes a necessity. In this project, the data mining technique that is being used is incremental Map reduce. When the volume of data increases, the computation process becomes tedious. To avoid this problem and to increase the efficiency of data mining we can make use of incremental map reduce process. Wherein, the past output is used to find the key-value pair rather than working from the scratch every time using the Bipartite Graph. By iterating the MapReduce process we can handle dynamic changes in the data efficiently.

Key words: Map Reduce, Data mining, incremental process, Bipartite graph, computation, iteration

I. INTRODUCTION

Due to the increase in amount of data from many areas like e-commerce, social networks, finance, health care etc, need for efficient data mining techniques becomes a necessity. In recent years, a large number of computing frameworks [1], [2], [3], [4], [5], [6], [7], [8], [9], [10] have been developed for big data analysis. Among these frameworks, MapReduce [1] (with its open-source implementations, such as Hadoop) is the most widely used in production because of its simplicity, generality, and maturity. We focus on improving MapReduce in this paper.

Big data is constantly evolving. As new data and updates are being collected, the input data of a big data mining algorithm will gradually change, and the computed results will become stale and obsolete over time. In many situations, it is desirable to periodically refresh the mining computation in order to keep the mining results up-to-date. Incremental processing is a promising approach to refreshing mining results. Given the size of the input big data, it is often very expensive to rerun the entire computation from scratch. Incremental processing exploits the fact that the input data of two subsequent computations A and B are similar, with only a very small fraction of change in the input data. The idea is to save states in computation A, re-use A's states in computation of B, and perform re-computation only for states that are affected by the changed input data. In this paper, we investigate the realization of this principle in the context of the MapReduce computing framework.

Incoop [13] extends MapReduce to support incremental processing. First, Incoop supports only task-level incremental processing. That is, it saves and reuses states at the coarseness of individual Map and reduce tasks. Every task generally processes an oversized number of key-value pairs (kvpairs). If Incoop detects any data changes within the input of a task, it'll rerun the whole task. Whereas this approach simply leverages existing MapReduce options for state savings, it should incur a large quantity of redundant computation if solely a small fraction of kv-pairs have modified during a task. Second, Incoop supports solely one-step computation, whereas important mining algorithms, like PageRank, need iterative computation. Incoop would treat every iteration as a separate MapReduce job. However, a small number of input data changes could gradually propagate to affect a large portion of intermediate states due to variety of iterations, leading to expensive global re-computation after.

We propose incremental MapReduce, an extension to MapReduce that supports incremental processing for both one step and iterative computation. Incremental MapReduce incorporates the following features:

Incremental MapReduce supports kv-pair level fine-grain incremental processing in order to minimize the amount of re-computation as much as possible. We model the kv-pair level data flow and data dependence in a MapReduce computation as a bipartite graph, called BGraph. A BG-Store is designed to preserve the fine-grain states in the BGraph and support efficient queries to retrieve fine-grain states for incremental processing.

Incremental MapReduce provides general purpose support, for not only one-to-one but also one-to-many, many-to-one, and many-to-many correspondence. While users need to slightly modify their algorithms in order to take full advantage of incremental MapReduce, such modification is modest compared to the effort to re-implement algorithms on a completely different programming paradigm.

Incremental iterative processing is substantially more challenging than incremental one-step processing because even a small number of updates may propagate to affect a large portion of intermediate states, after a number of iterations. To address this problem, we can reuse the converged state from the previous computation. We also enhance the BG-Store to better support the access patterns in incremental iterative processing.

II. MAPREDUCE BACKGROUND

The MapReduce algorithm contains two important tasks, namely Map and Reduce.

The Map task takes a set of data and converts it into another set of data, where individual elements are broken down into tuples (key-value pairs).

The Reduce task takes the output from the Map as an input and combines those data tuples (key-value pairs) into a smaller set of tuples.

The reduce task is always performed after the map job.

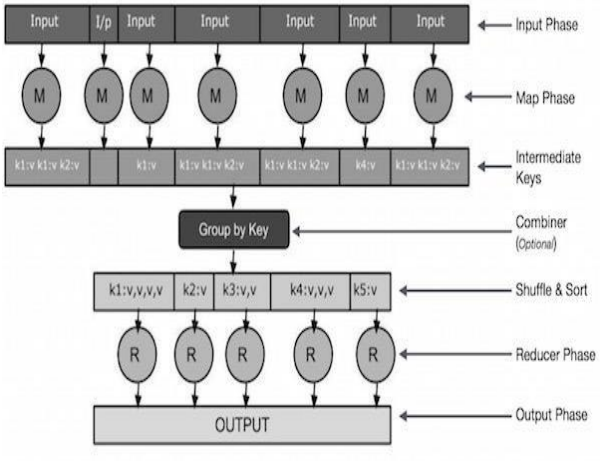

Fig. 1: Map Reduce Process

A. *Map Reduce In Hadoop*

A MapReduce system (e.g., Apache Hadoop) usually reads the input data and writes the final results of MapReduce computation to a distributed file system (e.g., HDFS), which divides a file into equal-sized (e.g., 64 MB) blocks and stores the blocks across a cluster of machines. For a MapReduce program, the MapReduce system runs a JobTracker process on a master node to monitor the job progress, and a set of TaskTracker processes on worker nodes to perform the actual Map and Reduce tasks.

The JobTracker starts a Map task per data block, and typically assigns it to the TaskTracker on the machine that holds the corresponding data block in order to minimize communication overhead.

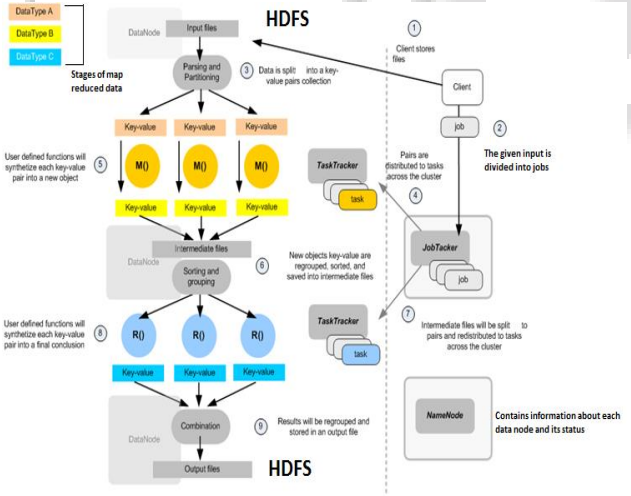

Fig. 2: MapReduce Lifecycle in Hadoop

III. METHODOLOGY

A. *Incremental Processing For One-Step Computation*

The basic idea of incremental processing is as follows: Incremental MR framework enables incremental process of new small data within large data set, it will take state as an implicit input and join it with new data. According to this, new splits within large data set Map tasks are created, at the same time reduce task obtain their inputs which have the state and intermediate results of new Map task. Incremental MR is a enhanced model for large-scale incremental data processing, that uses the original MapReduce model. Without any modification in existing Map-Reduce model it delivers incremental processing. It supports incremental data processing, intermediate state preservation and incremental map and reduce functions. In this model the Map tasks are same as MapReduce model map tasks, and reduce tasks executions are same as MapReduce Model's map tasks and reduce task. But there is a change in copy phase of reduce task, due to the inclusion of current map output with state in data resource. It allows MapReduce based applications without any modification. It enables incremental processing with task-level re-computation, but in this framework user themselves have to manipulate the states.

B. *Basic Idea*

Consider two MapReduce jobs A and A` performing the same computation on input data set D and D`, respectively. D` = D+ΔD, where ΔD consists of the updated data. An update can be either a deletion or an insertion. Our goal is to re-compute only the Map and Reduce function call instances that are affected by ΔD.

Incremental computation for Map is straightforward. We simply invoke the Map function for the inserted or deleted (K1, V1)s. Since the other input kv-pairs are not changed, their Map computation would remain the same. The delta intermediate values, include inserted and deleted (K2, V2)s are computed.

To perform incremental Reduce computation, we need to save the fine-grain states of job A, denoted M, which includes (K2, {V2})s. We will re-compute the Reduce function for each K2 in ΔM. The other K2 in M does not see any changed intermediate values and therefore would generate the same final result. For a K2 in ΔM, typically only a subset of the list of V2 has changed. Here, we retrieve the saved (K2, {V2}) from M, and apply the inserted and/or deleted values from ΔM to obtain an updated Reduce input. We then re-compute the Reduce function on this input to generate the changed final results (K3, V3)s.

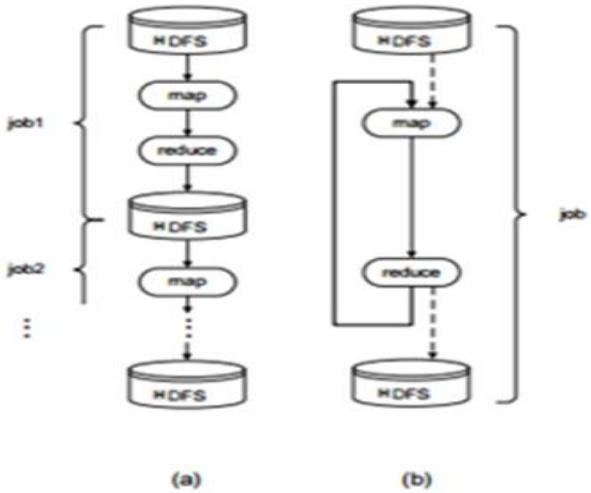

Fig. 3: (a) Data flow of MapReduce (b) Data flow of incremental MapReduce

C. *Bipartite Graph*

A Bipartite Graph can be incorporated in the Map Reduce process, to represent the data and the changes made. Each vertex in the Map task represents an individual Map function call instance on a pair of (K1, V1). Each vertex in the Reduce task represents an individual Reduce function call instance on a group of (K2, {V2}). An edge from a Map

instance to a Reduce instance means that the Map instance generates a (K2, V2) that is shuffled to become part of the input to the Reduce instance.

BGraph edges are the fine-grain states M that we would like to preserve for incremental processing. An edge contains three pieces of information: (i) the source Map instance, (ii) the destination Reduce instance (as identified by K2), and (iii) the edge value (i.e., V2). Since Map input key K1 may not be unique, incremental MapReduce generates a globally unique Map key MK for each Map instance. Therefore, incremental MapReduce will preserve (K2, MK, V2) for each BGraph edge.

As the data is given, it can be represented as a bipartite graph after the Map and Reduce functions are performed. The graph structure, evolves over time as and when changes are made to the data or new data is provided.

D. Initial Run And Bgraph Preserving

The initial run performs a normal MapReduce job, as shown in Fig. 4a. The Map input is the adjacency matrix of the graph. Every record corresponds to a vertex in the graph. K1 is vertex id, and V1 contains a destination vertex. The shuffling phase groups the edge weights by the destination vertex. Then the Reduce function computes the final values.
For incremental processing, we preserve the fine-grain BGraph edge states. A question arises: should the states be preserved at the Map side or at the Reduce side? We preserve the Reduce side, because during incremental processing original intermediate values can be obtained at the Reduce side without any shuffling overhead. The engine transfers the globally unique MK along with (K2, V2) during the shuffle phase. Then it saves the states (K2, MK, V2) in a BGraph file at every Reduce task.

Delta input: Incremental MapReduce expects delta input data that contains the newly inserted, deleted, or modified kv-pairs as the input to incremental processing. Fig. 4b shows the delta input for the updated application graph. A '+' symbol indicates a newly inserted kv-pair, while a '-' symbol indicates Fig. 4b shows the delta input for the updated application graph. A '+' symbol indicates a newly inserted kv-pair, while a '-' symbol indicates a deleted kv-pair. For example, the deletion of vertex 1 and its edge are reflected as (1, 2:0.4,'-'). The insertion of vertex 3 and its edge leads to (3, 0:0.1,'+'). The modification of the vertex 0's edges are reflected by a deletion of the old record (0, 1:0.3;2:0.3;'-') and an insertion of a new record (0, 2:0.6,'-').

Incremental map computation to obtain the delta Bipartite Graph. The engine invokes the Map function for every record in the delta input. For an insertion with '+', its intermediate results (K2, MK, V2`)s represent newly inserted edges in the BGraph. For a deletion with '-', its intermediate results indicate that the corresponding edges have been removed from the BGraph. The engine replaces the V2`s of the deleted BGraph edges with '-'. During the Map-Reduce shuffle phase, the intermediate (K2, MK, V2`)s and (K2, MK, '-')s with the same K2 will be grouped together. The delta graph will contain only the changes to the BGraph and sorted by the K2 order.

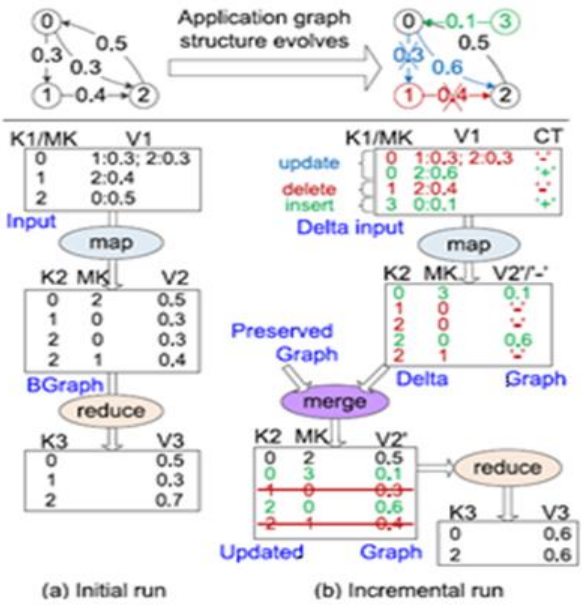

Fig. 4: Incremental processing for an application that computes the sum of in-edge weights for each vertex

Incremental reduce computation. The engine merges the delta graph and the preserved graph to obtain the updated MRB Graph using the algorithm in Section 3.4. For each (K2, MK, '-'), the engine deletes the corresponding saved edge state. For each (K2, MK, V2`), the engine first checks duplicates, and inserts the new edge if no duplicate exists, or else updates the old edge if duplicate exists.

E. Bgraph-Store

The BGraph-Store supports the preservation and retrieval of fine-grain states for incremental processing. We see two main requirements on the BG-Store. First, the BG-Store must incrementally store the evolving BGraph. Consider a scenario where there are constant changes made to the data. As the data changes, intermediate states in the BGraph will also change. In such cases, it will be wasteful to store the entire data in each subsequent job. To avoid this, we can obtain and store only the updated data in a BGraph. Second, the BG-Store must support efficient retrieval of preserved states of given Reduce instances. For incremental Reduce computation, incremental MapReduce re-computes the Reduce instance associated with each changed BGraph edge. For a changed edge, it queries the BG-Store to retrieve the preserved states of the in-edges of the associated values, and merge the preserved states with the newly computed edge changes. Fig. 5 depicts the structure of the BG-Store. We describe how the components of the BG-Store work together to achieve the above two requirements.

1) Initial State Retrieval And Merging

A BGraph file stores fine-grain intermediate states for a Reduce task. In Fig. 5, we see that the kv-pairs with the same k value are stored contiguously as a chunk. Since a chunk corresponds to the input to a Reduce instance, the design treats chunk as the basic unit, and always reads, writes, and operates on entire chunks. The contents of a delta BGraph file are shown on the bottom left of Fig. 5.

The merging of the delta BGraph with the BGraph file in the BG-Store is essentially a join operation using K2 as the join key. Since the size of the delta BGraph is typically much smaller than the BGraph file, it is wasteful to read the entire BGraph file. Therefore, an index for selective

access to the BGraph file is used. Given a K2, the index returns the chunk position in the BGraph file. As only point lookup is required, we employ a hash-based implementation for the index. The index is stored in an index file and is preloaded into memory before Reduce computation. We apply the index nested loop join for the merging operation.

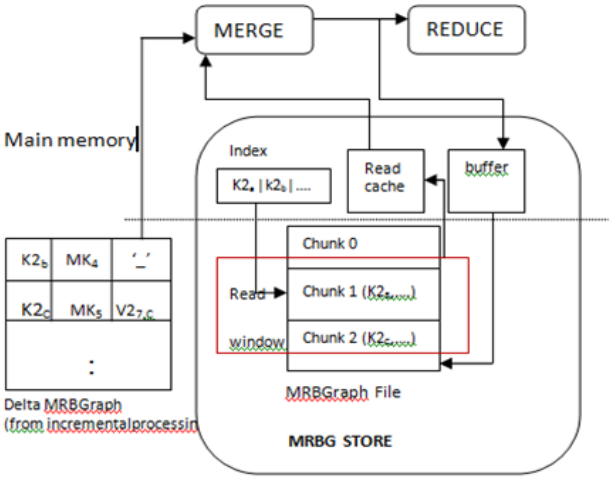

Fig. 5: Structure of BG-store

2) Incremental Storage Of Bgraph Changes

As shown in Fig. 5, the outputs of the merge operation, which are the up-to-date BGraph states (chunks), are used to invoke the Reduce function. In addition, the outputs are also buffered in an append buffer in memory. When the append buffer is full, the BG-Store performs sequential I/Os to append the contents of the buffer to the end of the BGraph file. When the merge operation completes, the BG-Store flushes the append buffer, and updates the index to reflect the new file positions for the updated chunks. Note that obsolete chunks are NOT immediately updated in the file (or removed from the file) for I/O efficiency. The BGraph file is reconstructed off-line when the worker is idle. In this way, the BG-Store efficiently supports incremental storage of BGraph Changes.

As a result of the incremental storage, the BGraph file may contain multiple segments of sorted chunks, each resulting from a merge operation. This situation frequently appears in iterative incremental computation, for which we enhance the above query algorithm with a multi-window technique to efficiently process the multiple segments.

F. General-Purpose Iterative Mapreduce Model

A number of recent efforts have been targeted at improving iterative processing on MapReduce, including Twister [9], HaLoop [8], and iMapReduce [10]. In general, the improvements focus on two aspects:

1) Reducing Job Startup Costs

In vanilla MapReduce, every algorithm iteration runs one or several MapReduce jobs. Hadoop may take over 20 seconds to start a job with 10–100 tasks. If the computation of each iteration is relatively simple, job startup costs may consist of an overly large fraction of the run time. The solution is to modify MapReduce to reuse the same jobs across iterations, and kill them only when the computation completes. For this, we modify Hadoop to allow jobs to stay alive across multiple iterations.

2) Caching Structure Data

Structure data is immutable during computation. It is also much larger than state data in many applications (e.g., PageRank, Kmeans, Single source shortest path and GIM-V). Therefore, it is wasteful to transfer structure data for every iteration. An optimization is to cache structure data in local file systems to avoid the cost of network communication and reading from HDFS. For this aspect, however, a design must separate structure data from state data, and consider how to match interdependent structure and state data in the computation. HaLoop [8] uses an extra MapReduce job to match structure and state data in each iteration. We would like to avoid such heavy-weight solution. iMapReduce [10] creates the same number of Map and Reduce tasks, and connects every Reduce task to a Map task with a local connection to transfer the state data output from a Reduce task to the corresponding Map task.

G. Energy Efficient Computation

In our current situation energy wastage is the major problem more of the IT firms. More workload and more computational will increase high energy cost. Main aim of ours, to reduce the energy cost from efficient Map reducing concepts. To optimize the mining results, we evaluate MapReduce using a one-step algorithm and three iterative algorithms with diverse computation characteristics for efficient mining also improve the energy. In this paper we also include the algorithm for incremental processing approach named as Energy map reduce scheduling algorithm .EMRSA[18] is algorithm provide more energy and less maps. Priority based scheduling is a task will allocate the schedules based on necessary and utilization of the Jobs. For reducing the maps, it will reduce the system work so easily energy has improve.

Algorithm 1 EMRSA-X

1: Create an empty priority queue $\mathcal{Q}^m$
2: Create an empty priority queue $\mathcal{Q}^r$
3: **for all** $j \in \mathcal{A}$ **do**
4: $ecr_j^m = \min_{i \in \mathcal{M}} \frac{\tau_{ij}}{p_{ij}}$, for EMRSA-I; or
$ecr_j^m = \frac{\sum_{i \in \mathcal{M}} \frac{\tau_{ij}}{p_{ij}}}{|\mathcal{M}|}$, for EMRSA-II
5: $\mathcal{Q}^m$.enqueue(j, ecr_j^m)
6: **for all** $j \in \mathcal{B}$ **do**
7: $ecr_j^r = \min_{i \in \mathcal{R}} \frac{\tau_{ij}}{p_{ij}}$, for EMRSA-I; or
$ecr_j^r = \frac{\sum_{i \in \mathcal{R}} \frac{\tau_{ij}}{p_{ij}}}{|\mathcal{R}|}$, for EMRSA-II
8: $\mathcal{Q}^r$.enqueue(j, ecr_j^r)
9: $D^m \leftarrow \infty$; $D^r \leftarrow \infty$
10: **while** $\mathcal{Q}^m$ is not empty **and** $\mathcal{Q}^r$ is not empty **do**
11: $j^m = \mathcal{Q}^m$.extractMin()
12: $j^r = \mathcal{Q}^r$.extractMin()
13: $f = \frac{\sum_{i \in \mathcal{M}} p_{ij^m}}{\sum_{i \in \mathcal{R}} p_{ij^r}}$
14: T^m: sorted unassigned map tasks $i \in \mathcal{M}$ based on p_{ij^m}
15: T^r: sorted unassigned reduce tasks $i \in \mathcal{R}$ based on p_{ij^r}
16: **if** $T^m = \emptyset$ **and** $T^r = \emptyset$ **then break**
17: ASSIGN-LARGE()
18: ASSIGN-SMALL()
19: **if** $D^m = \infty$ **then**
20: $D^m = D - p^r$
21: $D^r = p^r$
22: **if** $T^m \neq \emptyset$ **or** $T^r \neq \emptyset$ **then**
23: No feasible schedule
24: **return**
25: **Output:** X, Y

Fig. 1: Algorithms

IV. RELATED WORK

A. Iterative Processing

We discuss the frameworks that improve MapReduce. HaLoop [8], a modified version of Hadoop, improves the efficiency of iterative computation by making the task scheduler loop-aware and by employing caching mechanisms. Twister [9] employs a lightweight iterative MapReduce runtime system by logically constructing a Reduce-to-Map loop. iMapReduce [10] supports iterative processing by directly passing the Reduce outputs to Map and by distinguishing variant state data from the static data. Incremental MapReduce improves upon these previous proposals by supporting an efficient general-purpose iterative model. Pregel[4] follows Bulk Synchronous Processing (BSP) model. The computation is broken down into a sequence of supersteps. In each superstep, a Compute function is invoked on each vertex. It communicates with other vertices by sending and receiving messages and performs computation for the current vertex. This model can efficiently support a large number of iterative graph algorithms.

B. Incremental Processing For One-Step Application

Besides Incoop [13], several recent studies aim at supporting incremental processing for one-step applications. Stateful Bulk Processing [11] addresses the need for stateful dataflow programs. It provides a groupwise processing operator Translate that takes state as an explicit input to support incremental analysis. But it adopts a new programming model that is very different from MapReduce. In addition, several research studies [16], [17] support incremental processing by task-level re-computation, but they require users to manipulate the states on their own. In contrast, incremental MapReduce exploits a fine-grain kv-pair level re-computation that are more advantageous.

C. Incremental Processing For Iterative Application

Naiad [12] proposes a timely dataflow paradigm that allows stateful computation and arbitrary nested iterations. To support incremental iterative computation, programmers have to completely rewrite their MapReduce programs for Naiad. In comparison, we extend the widely used MapReduce model for incremental iterative computation. Existing Map- Reduce programs can be slightly changed to run on incremental MapReduce for incremental processing.

V. CONCLUSION

We have described incremental MapReduce, a MapReduce-based framework for incremental big data processing. Incremental MapReduce combines a fine-grain incremental engine, a general-purpose iterative model, and a set of effective techniques for incremental iterative computation. Real-machine experiments show that incremental MapReduce can significantly reduce the run time for refreshing big data mining results compared to re-computation on both plain and iterative MapReduce. And also an energy efficient computation can be done by following EMRSA.

REFERENCES

[1] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in Proc. 6th Conf. Symp. Opear. Syst. Des. Implementation, 2004, p. 10.
[2] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for, in-memory cluster computing," in Proc. 9th USENIX Conf. Netw. Syst. Des. Implementation, 2012, p. 2.
[3] R. Power and J. Li, "Piccolo: Building fast, distributed programs with partitioned tables," in Proc. 9th USENIX Conf. Oper. Syst. Des. Implementation, 2010, pp. 1–14.
[4] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in Proc. ACM SIGMOD Int. Conf. Manage. Data, 2010, pp. 135–146.
[5] S. R. Mihaylov, Z. G. Ives, and S. Guha, "Rex: Recursive, deltabased data-centric computation," in Proc. VLDB Endowment, 2012, vol. 5, no. 11, pp. 1280–1291.
[6] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning and data mining in the cloud," in Proc. VLDB Endowment, 2012, vol. 5, no. 8, pp. 716–727.
[7] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl, "Spinning fast iterative data flows," in Proc. VLDB Endowment, 2012, vol. 5, no. 11, pp. 1268–1279.
[8] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "Haloop: Efficient iterative data processing on large clusters," in Proc. VLDB Endowment, 2010, vol. 3, no. 1–2, pp. 285–296.
[9] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: A runtime for iterative mapreduce," in Proc. 19th ACM Symp. High Performance Distributed Comput., 2010, pp. 810–818.
[10] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "imapreduce: A distributed computing framework for iterative computation," J. Grid Comput., vol. 10, no. 1, pp. 47–68, 2012.
[11] D. Logothetis, C. Olston, B. Reed, K. C. Webb, and K. Yocum, "Stateful bulk processing for incremental analytics," in Proc. 1st ACM Symp. Cloud Comput., 2010, pp. 51–62.
[12] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: A timely dataflow system," in Proc. 24th ACM Symp. Oper. Syst. Principles, 2013, pp. 439–455.
[13] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquin, "Incoop: Mapreduce for incremental computations," in Proc. 2nd ACM Symp. Cloud Comput., 2011, pp. 7:1–7:14.
[14] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "Priter: A distributed framework for prioritized iterative computations," in Proc. 2nd ACM Symp. Cloud Comput., 2011, pp. 13:1–13:14.
[15] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "Accelerate large-scale iterative computation through asynchronous accumulative updates," in Proc. 3rd Workshop Sci. Cloud Comput. Date, 2012, pp. 13–22.

[16] C. Yan, X. Yang, Z. Yu, M. Li, and X. Li, "IncMR: Incremental data processing based on mapreduce," in Proc. IEEE 5th Int. Conf. Cloud Comput., 2012, pp.pp. 534–541.
[17] T. J€org, R. Parvizi, H. Yong, and S. Dessloch, "Incremental recomputations in mapreduce," in Proc. 3rd Int. Workshop Cloud Data Manage., 2011, pp. 7–14.
[18] Q.Zhang, L.Mashayekhy et al Energy-Aware Scheduling of MapReduce Jobs for Big Data Applications IEEE transactions on parallel and distributed systems, 2720-2733 vol:26 Issue:10,2015.

