ISSN (ONLINE): 2395-695X
ISSN (PRINT): 2395-695X

International Journal of Advanced Research in Biology, Engineering, Science and Technology (IJARBEST)
Vol. 2, Issue 4, April 2016

I^{2}MAPREDUCE: INCREMENTAL MAPREDUCE USING FUZZY C-MEANS CLUSTERING FOR MINING EVOLVING BIG DATA

Mr.T.Kumaragurubaranm M.Tech,(Ph.D)., , Ms.M.Santha lakshmi M.E(CSE).,
Assistant Professor , CSE, Mohamed Sathak Engineering College,kilakarai,India
Final year Student, CSE, Mohamed Sathak Engineering College,kilakarai,India

Abstract: **MapReduce is a processing technique and a program model for distributed computing based on java. This process contains two important tasks namely Map and Reduce. This framework is widely used for mining the big data. In the real world new data and their updates are constantly evolving, which cause the stale of data mining results. In order to overcome these challenges, we propose an Incremental Processing named i2Mapreduce an extension to MapReduce. Our i2Mapreduce (i) performs key-value pair level incremental processing, (ii) supports not only one-step computation but also more sophisticated iterative computation, which is widely used in data mining applications, and (iii) incorporates a set of novel techniques to reduce I/O overhead for accessing preserved fine-grain computation states. Experimental results on Amazon EC2 show significant performance improvements of i2MapReduce compared to both plain and iterative MapReduce performing re-computation.**

***IndexTerms*—Incremental processing, MapReduce, iterative computation, big data**

I INTRODUCTION

Today huge amount of digital data is being accumulated in many important areas, including e-commerce, social network, finance, health care, education, and environment. It has become increasingly popular to mine such big data in order to gain insights to help business decisions or to provide better personalized, higher quality services. 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. For example, the PageRank algorithm computes ranking scores of web pages based on the webgraph structure for supporting web search. However, the web graph structure is constantly evolving; Web pages and hyper-links are created, deleted, and updated. As the underlying web graph evolves, the PageRank ranking

A number of previous studies (including Percolator , CBP , and Naiad) have followed this principle and designed new programming models to support incremental processing. Unfortunately, the new programming models (BigTable observers in Percolator, stateful translate opera-tors in CBP, and timely dataflow paradigm in Naiad) are drastically different from MapReduce, requiring pro-grammers to completely re-implement their algorithms.

On the other hand, Incoop extends MapReduce to support incremental processing. However, it has two main limitations. First, Incoop supports only task-level incremental processing. That is, it saves and reuses states at the granularity of individual Map and Reduce tasks. Each task typically processes a large number of key-value pairs (kv-pairs). If Incoop detects any data changes in the input of a task, it will rerun the entire task. While this approach easily leverages existing MapReduce features for state savings, it may incur a large amount of redundant computation if only a small fraction of kv-pairs have changed in a task. Second, Incoop supports only one-step computation, while important mining algorithms, such as PageRank, require iterative computation. Incoop would treat each iteration as a separate MapReduce job. However, a small number of input data changes may gradually propagate to affect a large portion of intermediate states after a number of iterations, resulting in expensive global re-computation afterwards.

I propose i^{2}MapReduce, an extension to MapReduce that supports fine-grain incremental processing for both one-step and iterative computation. Compared to previous solutions, i^{2}MapReduce incorporates the following three novel features:

- Finegrain incremental processing using MRBGstore.
- General purpose iterative computation.
- Incremental processing for iterative computation

II MAPREDUCE BACKGROUND

A MapReduce program is composed of a Map function and a Reduce function [1], as shown in Fig. 1. Their APIs are as follows:

mapðK1; V 1Þ ! ½hK2; V 2i&

reduceðK2; fV 2gÞ ! ½hK3; V 3i&:

The Map function takes a kv-pair hK1; V 1i as input and computes zero or more intermediate kv-pairs hK2; V 2is. Then all hK2; V 2 is are grouped by K2. The Reduce function takes a K2 and a list of fV 2g as input and computes the final output kv-pairs hK3; V 3is.

A MapReduce system (e.g., Apache Hadoop) usually reads the input data of the MapReduce computation from and writes the final results to a distributed file sys-tem (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 job progress, and a set of TaskTracker pro-cesses 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. Each Map task calls the Map function for every input hK1; V 1i, and stores the intermediate kv-pairs hK2; V 2is on local disks. Intermediate results are shuffled to Reduce tasks according to a partition function (e.g., a hash function) on K2. After a Reduce task obtains and merges intermediate results from all Map Tasks, it invokes the Reduce function on each hK2; fV 2gi to generate the final output kv-pairs hK3; V 3.

III FINE GRAIN INCREMENTAL PROCESSING FOR ONE STEP COMPUTATION

I begin by describing the basic idea of fine-grain incremental processing , i present the main design, including the MRBGraph abstraction.

1 Basic Idea

Consider two MapReduce jobs A and A^0 performing the same computation on input data set D and D^0, respectively. D^0 ¼ D þ DD, where DD consists of the inserted and deleted input hK1; V 1is^1. An update can be represented as a dele-tion followed by an insertion. Our goal is to re-compute only the Map and Reduce function call instances that are affected by DD.

Incremental computation for Map is straightforward. We simply invoke the Map function for the inserted or deleted hK1; V 1is. Since the other input kv-pairs are not changed, their Map computation would remain the same. We now have computed the delta intermediate values, denoted DM, including inserted and deleted hK2; V 2is.

To perform incremental Reduce computation, we need to save the fine-grain states of job A, denoted M, which includes hK2; fV 2gis. We will re-compute the Reduce func-tion for each K2 in DM. The other K2 in M does not see any changed intermediate values and therefore would generate the same final result. For a K2 in DM, typically only
of V 2 have changed. Here, we retrieve the saved hK2; fV 2gi

ISSN (ONLINE): 2395-695X
ISSN (PRINT): 2395-695X

International Journal of Advanced Research in Biology, Engineering, Science and Technology (IJARBEST)
Vol. 2, Issue 4, April 2016

from M,

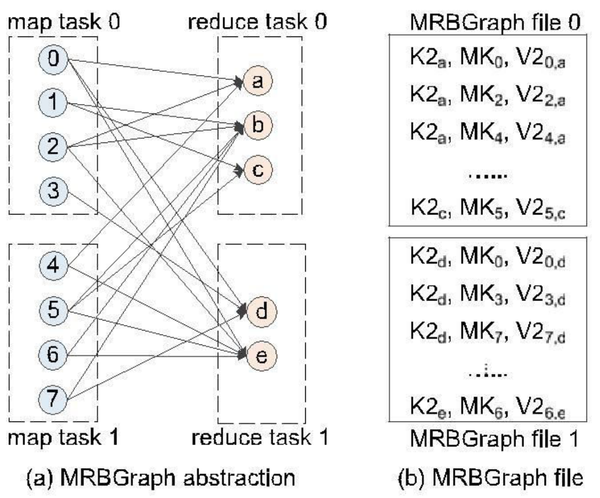


Fig No1:MRBGraph

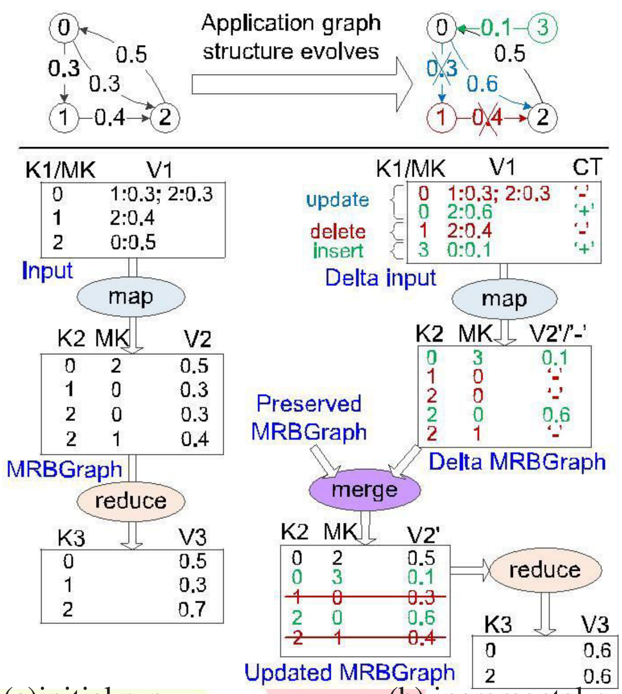
and apply the inserted and/or deleted values from DM to obtain an updated Reduce input. We then re-compute the Reduce function on this input to gener-ate the changed final results hK3; V 3.

2 MRBGraph Abstraction

We use a MRBGraph (Map Reduce Bipartite Graph) abstrac-tion to model the data flow in MapReduce, as shown in Fig. 2a. Each vertex in the Map task represents an individual Map function call instance on a pair of hK1; V 1i. Each vertex in the Reduce task represents an individual Reduce function call instance on a group of hK2; fV 2gi. An edge from a Map instance to a Reduce instance means that the Map instance

1 assume that new data or new updates are captured via incre-mental data acquisition or incremental crawling [16], [17]. Incremental data acquisition can significantly save the resources for data collection; it does not re-capture the whole data set but only capture the revisions since the last time that data was captured.generates a hK2; V 2i that is shuffled to become part of the input to the Reduce instance. For example, the input of Reduce instance a comes from Map instance 0, 2, and 4.

MRBGraph edges are the fine-grain states M that we would like to preserve for incremental processing. An edge contains



(a)initial run (b) incremental run

Fig No 2:. Incremental processing for an application that computes the sum of in-edge weights for each vertex.

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., V 2). Since Map input key K1 may not be unique, i^2MapReduce generates a globally unique Map key MK for each Map instance. There-fore, i^2MapReduce will preserve (K2, MK, V 2) for each MRBGraph edge.

3 Fine-Grain Incremental Processing Engine

Fig.2 illustrates the fine-grain incremental processing engine with an example application, which computes the sum of in-edge weights for each vertex in a graph. As shown at the top of Fig. 3, the input data, i.e., the graph structure, evolves over time. In the following, we describe how the engine performs incremental processing to refresh the analysis results.

Initial run and MRBGraph preserving. The initial run per-forms a normal MapReduce job, as shown in Fig. 3a. The Map input is the adjacency matrix of the graph. Every record corresponds to a vertex in the graph. K1 is vertex id i, and V 1 contains "j_1:$w_{i;j1}$; j_2:$w_{i;j2}$; ..." where j is a destination vertex and $w_{i;j}$ is the weight of the out-edge ði; jÞ. Given such a record, the Map function outputs intermediate kv-pair hj; $w_{i;j}$i for every j. The shuffling phase groups the edge weights by the destination vertex. Then the Reduce function computes for a vertex j the sum of all its in-edge weights as $_i$ $w_{i;j}$.

For incremental processing, we preserve the fine-grain MRBGraph edge states. A question arises: shall the states be preserved at the Map side or at the Reduce side? We choose the latter 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 hK2; V 2i during the shuffle phase. Then it saves the states (K2; MK; V 2) in a MRBGraph file at every Reduce task, as shown in Fig. 2b.

Delta input. i^2MapReduce expects delta input data that contains the newly inserted, deleted, or modified kv-pairs as the input to incremental processing. Note that identifying the data changes is beyond the scope of this paper; Many incremental data acquisition or incremental crawling tech-niques have been developed to improve data collection per-formance [16], [17].

Fig. 3b shows the delta input for the updated application graph. A 'þ' symbol indicates a newly inserted kv-pair, while a '_' symbol indicates a deleted kv-pair. An update is represented as a deletion followed by an insertion. For example, the deletion of vertex 1 and its edge are reflected as h1; 2:0:4;'_'i. The insertion of vertex 3 and its edge leads to h3; 0:0:1;'þ'i. The modification of the vertex 0's edges are reflected by a deletion of the old record h0; 1:0:3;2:0:3;'_'i and an insertion of a new record h0; 2:0:6;'_'i.

Incremental map computation to obtain the delta MRBGraph. The engine invokes the Map function for every record in the delta input. For an insertion with 'þ', its intermediate results hK2; MK; V 2^0 is represent newly inserted edges in the MRBGraph. For a deletion with '_', its intermediate results indicate that the corresponding edges have been removed from the MRBGraph. The engine replaces the V 2^0s of the deleted MRBGraph edges with '_'. During the Map-Reduce shuffle phase, the intermediate hK2; MK; V $2^0$is and hK2; MK;'_'is with the same K2 will be grouped together. The delta MRBGraph will contain only the changes to the MRBGraph and sorted by the K2 order.

Incremental reduce computation. The engine merges the delta MRBGraph and the preserved MRBGraph to obtain the updated MRBGraph using the algorithm in Section 3.4. For each hK2; MK;'_'i, the engine deletes the corresponding saved edge state. For each hK2; MK; V 2^0i, the engine first checks duplicates, and inserts the new edge if no duplicate exists, or else updates the old edge if duplicate exists. (Note that (K2, MK) uniquely identifies a MRBGraph edge.) Since an update in the Map input is represented as a deletion and an insertion, any modification to the intermediate edge state (e.g., h2; 0; _i in the example) consists of a deletion (e.g., h2; 0;'_'i) followed by an insertion (e.g., h2; 0; 0:6i). For each affected K2, the merged list of V 2 will be used as input to invoke the Reduce function to generate the updated final results.

4 MRBG-Store

The MRBG-Store supports the preservation and retrieval of fine-grain MRBGraph states for incremental processing. We see two main requirements on the MRBG-Store. First, the MRBG-Store must incrementally store the evolving MRBGraph. Consider a sequence of jobs that incrementally refresh the results of a big data mining algorithm. As input data evolves, the intermediate states in the MRBGraph will also evolve. It would be wasteful to store the entire

MRBGraph of each subsequent job. Instead, we would like to obtain and store only the updated part of the MRBGraph. Second, the MRGB-Store must Reduce instances. For incremental Reduce computation, i^2MapReduce re-computes the Reduce instance associated with each changed MRBGraph edge, as described in Section 3.3. For a changed edge, it queries the MRGB-Store to retrieve the preserved states of the in-edges of the

associated K2, and merge the preserved states with the newly computed edge changes.
Fig. 4 depicts the structure of the MRBG-Store. We describe how the components of the MRBG-Store work together to achieve the above two requirements.

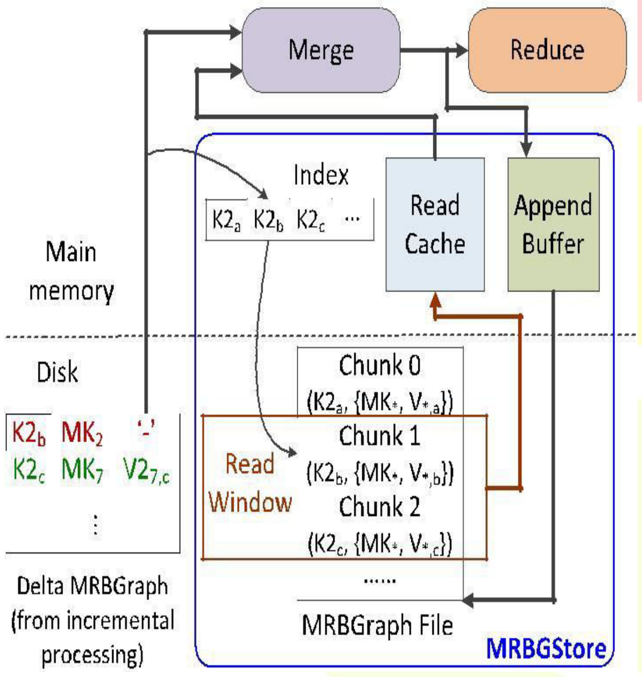


Fig No 4: Structure of MRBG store

Fine-grain state retrieval and merging. A MRBGraph file stores fine-grain intermediate states for a Reduce task, as illustrated previously in Fig. 2b. In Fig. 4, we see that the hK2; MK; V 2is with the same K2 are stored contiguously as a chunk. Since a chunk corresponds to the input to a Reduce instance, our design treats chunk as the basic unit, and always reads, writes, and operates on entire chunks.

The contents of a delta MRBGraph file are shown on the bottom left of Fig. 4. Every record represents a change in the original (last preserved) MRBGraph. There are two kinds of records. An edge insertion record (in green color) contains a valid V 2 value; an edge deletion record (in red color) contains a null value (as marked by '_')

The merging of the delta MRBGraph with the MRBGraph file in the MRBG-Store is essentially a join operation using K2 as the join key. Since the size of the delta MRBGraph is typically much smaller than the MRBGraph file, it is waste-ful to read the entire MRBGraph file. Therefore, we construct an index for selective access to the MRBGraph file: Given a K2, the index returns the chunk position in the MRBGraph 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. both the delta MRBGraph and the MRBGraph file are in the order generated by the shuffling phase. That is, the two files are sorted in K2 order. Therefore, we introduce a read cache and a dynamic read window technique for further optimization. Fig. 4 shows the idea. Given a sequence of K2s, there are two ways to read the corresponding chunks: (i) performing an individual I/O operation for each chunk; or (ii) per-forming a large I/O that covers all the required chunks. The former may lead to frequent disk seeks, while the latter may result in reading a lot of useless data. Fortunately, we know the list of sorted K2s to be queried. Using the index, we obtain their chunk positions. We can estimate the costs of using a large I/O versus a number of individual I/Os, and intelligently determine the read window size w based on the cost estimation.

Algorithm 1 shows the query algorithm to retrieve the the chunk k given a query key k and the list of queried keys L ¼ fL$_1$; L$_2$; . . .g. If the chunk k does not reside in the read cache (line 1), it will compute the read window size w by a heuristic, and read w bytes into the read cache. The loop (line 4–8) probes the gap between two consecutive queried chunks (chunk L_i and chunk $L_{iþ1}$). The gap size indicates the wasted read effort. If the gap is less than a threshold T (T ¼ 100 KB by default), we consider that the benefit of large I/O can compensate for the wasted read effort, and enlarge the window to cover chunk $L_{iþ1}$. In this way, the algorithm finds the read window size w by balancing the cost of a large I/O versus a number of individual I/Os. It also ensures that the read window size does not exceed the read cache. Then the algorithm read the next w bytes into the read cache (line 9) and retrieves the requested chunk k from the read cache (line 11). Incremental storage of MRBgraph changes. As shown in Fig. 4, the outputs of the merge operation, which are the up-to-date MRBGraph states (chunks), are used to invoke the Reduce function. In addition, the outputs are also buffered in an append buffer in memory

```
Algorithm 1. Query Algorithm in MRBG-Store
Input queried key: k; the list of queried keys: L
Output chunk k
 1: if ! read-cache.contains(k) then
 2:     gap    0, w    0
 3:     i      k's index in L          == That is, Li ¼ k
 4:     while gap < T and w þ gap þ lengthðLiÞ < read -cache:
        size do
 5:         w   w þ gap þ lengthðLiÞ
 6:         gap   posðLiþ1Þ _ posðLiÞ _ lengthðLiÞ
 7:         i   i þ 1
 8:     end while
 9:     starting from posðkÞ, read w bytes into read cache
10: end if
11: return read cache.get_chunk(k)
```

. When the append buffer is full, the MRBG-Store performs sequential I/Os to append the contents of the buffer to the end of the MRBGraph file. When the merge operation completes, the MRBG-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 MRBGraph file is reconstructed off-line when the worker is idle. In this way, the MRBG-Store efficiently supports incre-mental storage of MRBGraph Changes.

IV GENERAL PURPOSE SUPPORT FOR ITERATIVECOMPUTATION

value $R_{i;j}$ I first analyze several representative iterative algorithms in Section 4.1. Based on this analysis, we propose a general-purpose MapReduce model for iterative computation in Section 4.2, and describe how to efficiently support this model in Section 4.3.

1 Analyzing Iterative Computation

PageRank. PageRank [11] is a well-known iterative graph algorithm for ranking web pages. It computes a ranking score for each vertex in a graph. After initializing all ranking scores, the computation performs a MapReduce job per iteration, as shown in Algorithm 2. i and j are vertex ids, N_i is the set of out-neighbor vertices of i, R_i is i's ranking score that is updated iteratively. 'j' means concatenation. All R_i's are initialized to one.[2] The Map instance on vertex i sends ¼ R_i=jN_i j to all its out-neighbors j, where jN_i j is the number of i's out-neighbors. The Reduce instance on vertex j updates R_j by summing the $R_{i;j}$ received from all its in-neighbors i, and applying a damping factor d.

```
Algorithm 2. PageRank in MapReduce
Map Phase input: < i, Ni|Ri >
1: output < I, Ni >
2: for all j in Ni do
3: Ri;j ¼ jRi j jNi j
4:     output < j, Ri;j >
5: end for
Reduce Phase input: < j, fRi;j; Njg >
6: Rj ¼ d  ¡ Ri;j þ ð1 _ dÞ
7: output < j, Nj|Rj >
```

Kmeans. Kmeans is a commonly used clustering algo-rithm that partitions points into k clusters. We denote the ID of a point as pid, and its feature values pval. The computation starts with selecting k random points as cluster centroids set fcid; cvalg. As shown in Algorithm 3, in each iteration, the Map instance on a point pid assigns the point to the nearest centroid. The Reduce instance on a centroid cid updates the centroid by averaging the values of all assigned points fpvalg.

```
Algorithm 3. Kmeans in MapReduce
Map Phase input: < pid, pval|fcid; cvalg >
1: cid   find the nearest centroid of pval in fcid; cvalg
2: output < cid, pval >
Reduce Phase input: < cid, fpvalg >
3: cval   compute the average of fpvalg
4: output < cid, cval >
```

GIM-V. Generalized Iterated Matrix-Vector multiplica-tion (GIM-V) is an abstraction of many iterative graph mining operations (e.g., PageRank, spectral clustering, diameter estimation, connected components). These graph mining algorithms can be generally represented by operat-ing on an n _ n matrix M and a vector v of size n. Suppose both the matrix and the vector are divided into sub-blocks. Let $m_{i;j}$ denote the ði; jÞth block of M and v_j denote the jth block of v. The computation steps are

ISSN (ONLINE): 2395-695X
ISSN (PRINT): 2395-695X

International Journal of Advanced Research in Biology, Engineering, Science and Technology (IJARBEST)
Vol. 2, Issue 4, April 2016

similar to those of the matrix-vector multiplication and can be abstracted into three operations: (1) $mv_{i;j}$ = combine2($m_{i;j}$; v_j); (2) v^0_i = combineAll$_i$({$mv_{i;j}$}); and (3) v_i = assign(v_i; v^0_i). We can compare combine2 to the multiplication between $m_{i;j}$ and v_j, and compare combineAll to the sum of $mv_{i;j}$ for row i. Algorithm 4 shows the MapReduce implementation with two jobs for each iteration. The first job assigns vector block v_j to multiple matrix blocks $m_{i;j}$ (8i) and performs com-bine2($m_{i;j}$; v_j) to obtain $mv_{i;j}$. The second job groups the $mv_{i;j}$ and v_i on the same i, performs the combineAll ({$mv_{i;j}$}) operation, and updates v_i using assign(v_i; v^0_i).

V INCREMENTAL ITERATIVE PROCESSING

In this section, we present incremental processing techni-ques for iterative computation. Note that it is not sufficient to simply combine the above solutions for incremental one-step processing (in Section 3) and iterative computation (in Section 4). In the following, we discuss three aspects that we address in order to achieve an effective design

VI EXPERIMENTS

I implement a prototype of i^2MapReduce by modifying Hadoop-1.0.3. In order to support incremental and iterative processing, a few MapReduce APIs are changed or added. We summarize these API changes in for more details). In this section, we perform real-machine experiments to evaluate i^2MapReduce.

1 Experiment Setup

Solutions to compare. Our experiments compare four solutions: (i) PlainMR recomp, re-computation on vanilla Hadoop; (ii) iterMR recomp, re-computation on Hadoop optimized for iterative computation (as described in Section 4); (iii) HaLoop recomp, re-computation on the iterative MapReduce framework HaLoop [8], which optimizes MapReduce by providing a structure data caching mechanism; (iv) i^2MapReduce, our proposed solution. To the best of our knowledge, the task-level coarse-grain incremental processing system, Incoop , is not publicly available. Therefore, we cannot compare i^2MapReduce with Incoop. Nevertheless, our statistics show that without careful data partition, almost all tasks see changes in the experiments, making task-level incremental processing less effective.

Experimental environment. All experiments run on Amazon EC2. We use 32 m1.medium instances. Each m1. medium instance is equipped with 2 ECUs, 3.7 GB memory, and 410 GB storage.

Applications. We have implemented four iterative mining algorithms, including PageRank (one-to-one correlation), Single Source Shortest Path (SSSP, one-to-one correlation), Kmeans (all-to-one correlation), and GIM-V (many-to-one correlation). For GIM-V, we implement iterative matrix-vector multiplication as the concrete application using GIM-V model

VII CONCLUSION

Big Data is constantly evolving day to day, so that the data mining applications are stale and obsolete overtime, to overcome the challenges we propose an incremental processing method called i2MapReduce, an extension of MapReduce which is used for Mining Big Data. The i2MapReduce is a MapReduce based framework for incremental big data processing. This approach has three steps they are, a fine grained incremental engine, a general-purpose iterative model, and a set of effective techniques for incremental iterative computation called change propagation control. Our experimental results show significance performance on our dataset which has improvements of i2MapReduce compared to both plain and iterative MapReduce that performs re-computation.

VIII FUTURE WORK

Our Future work describes the i2MapReduce which is an extension of MapReduce. This work used for the incremental big data processing, which uses fine-grained incremental engine, a general purpose iterative model that includes iteration algorithms such as

PageRank, Possibilistic fuzzy c-means, Generalized Iterated Matrix-Vector multiplication. Finally the performance and comparison result and resultant graph are displayed.

IX REFERENCES

1. S. Brin, and L. Page, “The anatomy of a large-scale hypertextual web search engine,” Comput. Netw. ISDN Syst., vol. 30, no. 1–7, pp. 107–117, Apr. 1998.

2. D. Peng and F. Dabek, “Large-scale incremental processing using distributed transactions and notifications,” in Proc. 9th USENIX Conf. Oper. Syst. Des. Implementation, 2010, pp. 1–15.

3. 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.

4. 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.

5. 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.

6. J. Cho and H. Garcia-Molina, “The evolution of the web and implications for an incremental crawler,” in Proc. 26th Int. Conf. Very Large Data Bases, 2000, pp. 200–209.

7. C. Olston and M. Najork, “Web crawling,” Found. Trends Inform. Retrieval, vol. 4, no. 3, pp. 175–246, 2010.

8. S. Lloyd, “Least squares quantization in PCM,” IEEE Trans. Inform. Theory., vol. 28, no. 2, pp. 129–137, Mar. 1982.

9. U. Kang, C. Tsourakakis, and C. Faloutsos, “Pegasus: A peta-scale graph mining system implementation and observations,” in Proc. IEEE Int. Conf. Data Mining, 2009, pp. 229–238.

10. Y. Zhang, S. Chen, Q. Wang, and G. Yu, “i2mapreduce: Incremental mapreduce for mining evolving big data,” CoRR, vol. abs/ 1501.04854, 2015.