© 2016 IJSRSET | Volume 2 | Issue 2 | Print ISSN : 2395-1990 | Online ISSN : 2394-4099
Themed Section: Engineering and Technology

Incremental and Iterative Map reduce For Mining Evolving the Big Data in Banking System

Gayathri S A, Preethi. M, Shanmugapriya S, Sivagami S
Information Technology, Velammal Institute of Technology, Chennai, Tamilnadu, India

ABSTRACT

Big data is a broad term for datasets so large and complex that the traditional data processing applications are inadequate, so i2mapreduce based framework for incremental and iterative computations are done in big data. State level processing computation easily retrieve the data and also time consuming. Incremental and iterative mapreduce- mapreduce is the most widely used big data processing tool incremental processing is a promising approach to refresh the mining results. Use the same computation logic (update function) to process the data many times. The previous iteration's output is the next iteration's input, Stop when the iterated results converges to a fixed point. This concept using the online banking such as create account, withdraw, deposit and to get the details in a effective way. Finally, upload all the data to the cloud using AES algorithm. I2mapreduce is one step algorithm and four iterative algorithm with diverse computation characteristics. It is very secure to all the data are stored in binary format 0 and 1.

Keywords: Big data, Mining, Map reduce, Hadoop, CBP, MRBGraph

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 have been developed for big data analysis. Among these frameworks, MapReduce (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 datamining 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 resultsup-to-date. For example, the PageRank algorithm computes ranking scores of web pages based on the web graph 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 results gradually become stale, potentially lowering the quality of web search. Therefore, it is desirable to refresh the PageRank computation regularly. 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. Only a very small fraction of the input data has changed. The idea is to save states in computation A, re-use A's states in computation 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. A number of previous studies have followed this principle and designed new programming models to support incremental processing. Unfortunately, the new programming models (BigTable observers in Percolator, stateful translate operators in CBP, and timely dataflow paradigm in Naiad) are drastically different from MapReduce, requiring programmers 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 hangs may gradually propagate to affect a large portion of intermediate states after a number of iterations, resulting in expensive global re-computation afterwards.

II. METHODS AND MATERIAL

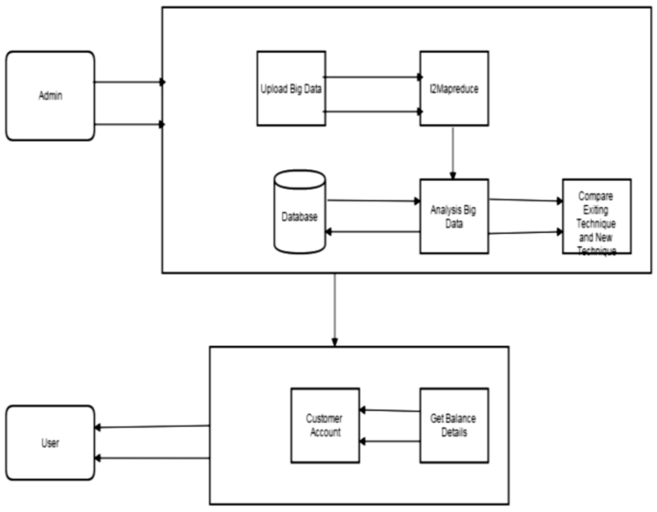
A. Existing System

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 MapReduce-based framework for incremental big data processing. MapReduce combines a fine-grain incremental engine, a general-purpose iterative model, and a set of effective techniques for increment MapReduce reschedules the failed Map/Reduce task in case task failure is detected. However, the interdependency of prime Reduce tasks and prime Map tasks in MapReduce requires more complicated fault-tolerance solution. i2MapReduce checkpoints the prime Reduce task's output state data and MRBGraph file on HDFS To the best of our knowledge, the task-level coarse-grain incremental processing system, Incoop is not publicly available. Therefore, we cannot compare i2 MapReduce 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.

Disadvantages of Existing System

- Task-level incremental processing less effective.
- Plain and iterative MapReduce performing re-computation.
- MapReduce re-computation takes long time.
- Performance is Low in Runtime.

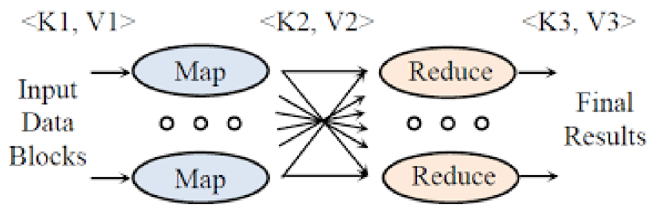
Architecture



Fine-Grain Incremental Processing For One-Step Computation

We begin by describing the basic idea of fine-grain incremental processing in Section 3.1. In Sections 3.2 and 3.3, we present the main design, including the MRBGraph abstraction and the incremental processing engine Then in Sections 3.4 and 3.5, we delve into two aspects of the design, i.e., the mechanism that preserves the fine-grain states, and the handling of a special but popular case where the Reduce function performs accumulation operations. Basic Idea Consider two MapReduce jobs A and A0 performing the same computation on input data set D and D0, respectively. D0 ¼ DþDD, where DD consists of the inserted and deleted input hK1; V1is1. An update can be represented as a deletion 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;V1 is. 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 deletedhK2;V2is.

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 function 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 a subset of the list of V2 has changed. Here, we retrieve the saved hK2; fV2gi from M, 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 generate the changed final resultshK3;V3is. It is easy to see that results generated from this incremental computation are logically the same as the results from completely re-computing A0.



B. MRB graph Abstraction

We use a MRBGraph (Map Reduce Bipartite Graph) abstraction 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 ofhK1;V1i. Each vertex in the Reduce task represents an individual Reduce function call instance on a group ofhK2;f V2gi. An edge from a Map instance to a Reduce instance means that the Map instance generates a hK2;V2i 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 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, i2MapReduce generates a globally unique Map key MK for each Map instance. Therefore, i2MapReduce will preserve (K2, MK, and V 2) for each MRBGraph edge.

C. Managing Bank Accounts

It provide two modules for managing a bank account. One is intended to be used by the bank, and the other by the customer. The approach is to implement a general-purpose parameterized functor providing all the needed operations, then apply it twice to the correct parameters, constraining it by the signature corresponding to its final user: the bank or the customer.This set of functions provide the minimal operations on an account. The creation operation takes as arguments the initial balance and the maximal overdraft allowed. Excessive withdrawals may raise the Bad Operation exception. We keep unspecified for now the types of the log keys (type *tkey*) and of the associated data (type *tinfo*), as well as the data structure for storing logs (type *t*). We assume that new informations added with the add function are kept in sequence.

D. Upload Big Data to File

Analyzing transactional data is at the core of the data at a financial institution’s disposal. Transaction data can uncover powerful insights into customer needs, preferences and behaviors. However, transaction data represents only one type of insight that financial institutions possess. Other types of insight that reside within an organization include both structured data (demographic profiles, product ownership, balances, etc.) and unstructured internal data (call center logs, channel interactions, correspondence, etc.).In addition to internal data sources, banks and credit unions can also take advantage of external data. Social media represents a largely untapped source of insight that financial organizations can use to develop a more holistic view of their customers. Financial organizations and their executives to improve their customer experience levels to differentiate themselves and to stay ahead of competitors. This, in turn, will improve acquisition results, engagement and cross-sell effectiveness as well as customer loyalty and growth. Banks and credit unions can achieve this by leveraging existing and historical consumer data to target consumers at the individual level and foster a more custom and personalized experience.

E. I2mapreduce and Analyzing Iterative Computation

Cloud intelligence applications often perform iterative computations (e.g., Online Bank) on constantly changing data sets (e.g., Web graph). While previous studies extend MapReduce for efficient iterative computations, it is too expensive to perform an entirely new large-scale MapReduce iterative job to timely accommodate new changes to the underlying data sets. In this paper, we propose i2MapReduce to support incremental iterative computation. We observe that in many cases, the changes impact only a very small fraction of the data sets, and the newly iteratively converged state is quite close to the previously converged state. i2MapReduce exploits this observation to save re-computation by starting from the previously converged state, and by performing incremental updates on the changing data. Our preliminary result is quite promising. i2MapReduce sees significant performance improvement over re-computing iterative jobsMapReduce.

III. RESULTS AND DISCUSSION

General-Purpose Support For Iterative Computation

We first analyze several representative iterative algorithms in Section 4.1. Based on this analysis, we propose a generalpurpose MapReduce model for iterative computation in Section 4.2, and describe how to efficiently support this model in Section 4.3.

Analyzing Iterative Computation PageRank.

PageRank 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, Ni is the set of out-neighbor vertices of i, Ri is i’s ranking score that is updated iteratively. ‘|’ means concatenation.

All Ri’s are initialized to one2. The Map instance on vertex i sends value Ri,j = Ri/|Ni| to all its out-neighbors j, where |Ni| is the number of i’s out-neighbors. The Reduce instance on vertex j updates Rj by summing the Ri,j received from all its in-neighbors i, and applying a damping factor d. Algorithm 2 PageRank in MapReduce

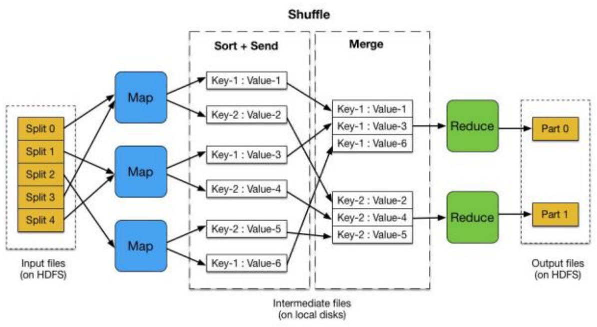
```
Map Phase input: < i, Ni|Ri >
1: output < i, Ni >
2: for all j in Ni do
3: Ri,j = Ri |Ni|
4: output < j, Ri,j >
5: end for
Reduce Phase input: < j, {Ri,j,Nj} >
6: Rj = dPi Ri,j + (1 − d)
7: output < j, Nj|Rj >
```

Kmeans. Kmeans [15] is a commonly used clustering algorithm 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 {cid,cval}. 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 {pval}.

Algorithm 3 Kmeans in MapReduce

```
Map Phase input: < pid, pval|{cid,cval} >
1: cid ← find the nearest centroid of pval in {cid,cval}
2: output < cid, pval >
Reduce Phase input: < cid, {pval} >
3: cval ← compute the average of {pval}
4: output < cid, cval >
```

GIM-V. Generalized Iterated Matrix-Vector multiplication (GIM-V) [13] 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 operating 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 mi,j denote the (i,j)-th block of M and vj denote the jth block of v. The computation steps are similar to those of the matrix-vector multiplication and can be abstracted into three operations: (1) mvi,j = combine2(mi,j,vj); (2) v′ i = combineAlli({mvi,j}); and (3) vi = assign(vi,v′ i). We can compare combine2 to the multiplication between mi,j and vj, and compare combineAll to the sum of mvi,j for row i.

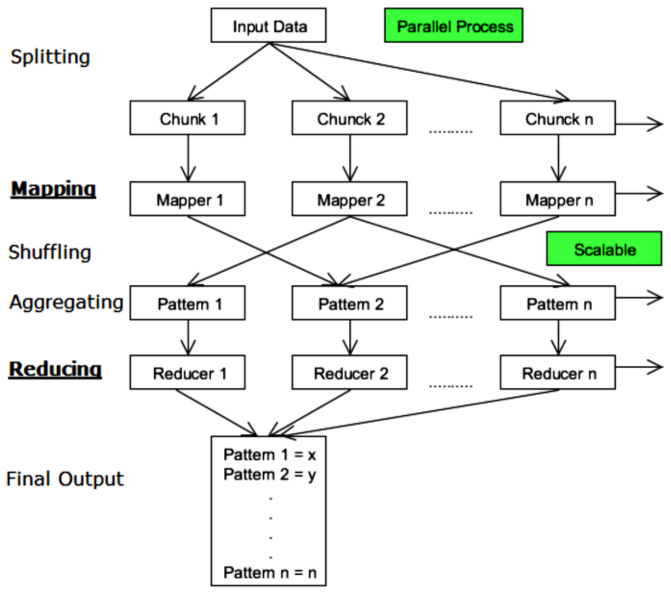


Algorithm 4 shows the MapReduce implementation with two jobs for each iteration. The first job assigns vector block vj to multiple matrix blocks mi,j (∀i) and performs combine2(mi,j,vj) to obtain mvi,j. The second job groups the mvi,j and vi on the same i, performs the combineAll({mvi,j}) operation, and updates vi using assign(vi,v′ i). The computed PageRank scores will be |N| times larger, where |N| is the number of vertices in the graph.

Algorithm 4 GIM-V in MapReduce

```
Map Phase 1 input: < (i,j),mi,j > or < j,vj >
1: if kv-pair is < (i,j),mi,j > then
2: output < (i,j),mi,j >
3: else if kv-pair is < j,vj > then
4: for all i blocks in j's row do
5: output < (i,j),vj >
6: end for
7: end if
Reduce Phase 1 input: < (i,j),{mi,j,vj} >
8: mvi,j = combine2(mi,j, vj)
9: output < i, mvi,j >, < j, vj >

Map Phase 2: output all inputs
```



```
Reduce Phase 2 input: < i,{mvi,j,vi} >

10: v′ i ← combineAll({mvi,j})
11: vi ← assign(vi, v′ i) 12: output < i, vi >
```

Supporting Smaller Number of State kv-pairs. In some applications, the number of state keys is smaller than n. Kmeans is an extreme case with only a single state kv-pair. In these applications, the total size of the state data is typically quite small. Therefore, the backward transfer overhead is low. Under such situation, i2MapReduce does not apply the above partition functions. Instead, it partitions the structure kv-pairs using MapReduce's default approach, while replicating the state data to each partition.

IV. CONCLUSION

We have described i2MapReduce, a MapReduce-based framework for incremental big data processing. i2 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 i2 MapReduce can significantly reduce the run time for refreshing big data mining results compared to re-computation on both plain and iterative MapReduce.

V. FUTURE WORK

In Future Besides Incoop several recent studies aim at supporting incremental processing for one-step applications. Stateful Bulk Processing 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 support incremental processing by task-level re-computation, but they require users to manipulate the states on their own. In contrast, i2MapReduce exploits a fine-grain kv-pair level re-compute To support incremental iterative computation, programmers have to completely rewrite their Map Reduce programs for Naiad. In comparison, we extend the widely used MapReduce model for incremental iterative computation. Existing MapReduce programs can be slightly changed to run on i2MapReduce for incremental processing.

VI. 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. Implemen-tation, 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. 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.

[12] 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.