i^{2}MapReduce: Incremental MapReduce for Mining Evolving Big Data (Extended Abstract)

Yanfeng Zhang*, Shimin Chen†, Qiang Wang*, Ge Yu*
*Northeastern University
†Institute of Computing Technology, CAS

***Abstract*—As new data and updates are constantly arriving, the results of data mining applications become stale and obsolete over time. Incremental processing is a promising approach to refresh mining results. It utilizes previously saved states to avoid the expense of re-computation from scratch. In this paper, we propose i^{2}MapReduce, a novel incremental processing extension to MapReduce. Compared with the state-of-the-art work on Incoop, i^{2}MapReduce (i) performs key-value pair level incremental processing rather than task level re-computation, (ii) supports not only one-step computation but also more sophisticated iterative computation, 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 i^{2}MapReduce compared to both plain and iterative MapReduce performing re-computation.**

I. Introduction

Big data is constantly evolving. As new data and updates are being collected, the input of a mining algorithm will gradually change. The 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 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, to reuse A's states in computation B, and to 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 most widely used MapReduce computing framework.

A number of previous studies (including Percolator [1], CBP [2], and Naiad [3]) 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 [4] 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.

Therefore, a scalable MapReduce extension that supports both iterative and incremental computations is desired. In this paper, we propose i^{2}MapReduce that supports *fine-grain* incremental processing for both *one-step* and *iterative* computation.

II. Ideas and Techniques

As we know, a MapReduce program is composed of a Map function and a Reduce function. The Map function takes a kv-pair $\langle K1, V1\rangle$ as input and produces intermediate kv-pairs $\langle K2, V2\rangle$s. Then all $\langle K2, V2\rangle$s are grouped by $K2$, i.e., $\langle K2, [V2]\rangle$, which are processed by the Reduce function to generate the final output kv-pairs $\langle K3, V3\rangle$s. i^{2}MapReduce extends the MapReduce model with the following techniques.

Fine-grained Incremental Processing. Consider two MapReduce jobs A and A' performing the same computation on the input data set D and D', respectively. $D' = D + \Delta D$, where ΔD consists of the inserted and deleted input $\langle K1, V1\rangle$s. 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 $\langle K1, V1\rangle$s. We now have computed the delta intermediate values, denoted ΔM, including inserted and deleted $\langle K2, V2\rangle$s. To perform incremental Reduce computation, we need to save the fine-grain states of job A, denoted M, which includes $\langle K2, [V2]\rangle$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 $V2$ have changed. Here, we retrieve the saved $\langle K2, [V2]\rangle$ 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 $\langle K3, V3\rangle$s.

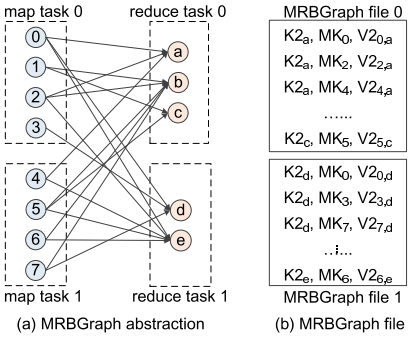

Fig. 1. MRBGraph.

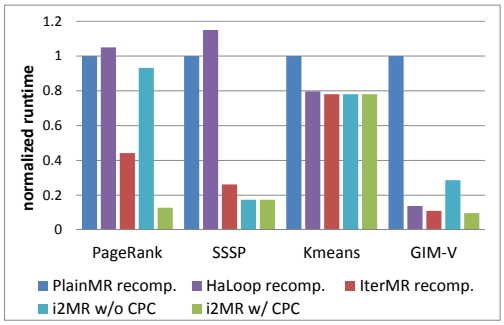

Fig. 2. Normalized runtime.

MRBGraph and MRBG-Store. We use a MRBGraph (Map Reduce Bipartite Graph) abstraction to model the kv-pair level data flow and data dependence in MapReduce, as shown in Fig. 1. Each vertex represents an individual Map or Reduce function call instance on a pair of $\langle K1, V1\rangle$ or on a group of $\langle K2, [V2]\rangle$. An edge from a Map instance to a Reduce instance means that the Map instance generates a $\langle K2, V2\rangle$ that is shuffled to become part of the input to the Reduce instance. 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, i^{2}MapReduce generates a globally unique Map key MK for each Map instance. Therefore, i^{2}MapReduce will preserve ($K2$,MK,$V2$) for each MRBGraph edge. Furthermore, MRBG-Store is designed to preserve MRBGraph and to support efficient queries to retrieve fine-grain states for incremental processing.

General-Purpose Iterative MapReduce Model. Our previous work proposed iMapReduce [5] to efficiently support iterative computation on the MapReduce platform. However, it targets types of iterative computation where there is a one-to-one/all-to-one correspondence from Reduce output to Map input. i^{2}MapReduce not only integrates the existing iterative MapReduce optimizations, e.g. consistent task scheduling to avoid repetitive startup cost and loop-invariant data caching to avoid unnecessary static data communication, but also provides general-purpose support, including one-to-one, one-to-many, many-to-one, and many-to-many correspondence. We propose a Project API to express the correspondence from Reduce to Map. Based on the user-specified correspondence, i^{2}MapReduce co-locates the corresponded Map and Reduce tasks and take full advantages of data locality to improve performance. This will speedup the MapReduce processing for both iterative and incremental iterative computations.

Incremental Processing for Iterative Computation. 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 propose to reuse the converged state from the previous computation and employ a change propagation control mechanism. In addition, the access patterns to MRBGraph file is different from one-step incremental processing. We also enhance the MRBG-Store to reduce random I/Os of MRBGraph file by proposing a multi-dynamic-window caching technique.

III. Experimental Results

We compare four solutions: (i) *PlainMR recomp*, re-computation on vanilla Hadoop; (ii) *iterMR recomp*, re-computation on Hadoop optimized for iterative computation; (iii) *HaLoop recomp*, re-computation on the iterative MapReduce framework HaLoop [6]; (iv) *i^{2}MapReduce*. The experiments are performed on Amazon EC2 with 32 m1.medium instances. The comparisons are in the context of APriori algorithm and four typical iterative algorithms, including PageRank, SSSP, Kmeans, and GIM-V.

We first use APriori algorithm to understand the benefit of incremental one-step processing. i^{2}MapReduce sees a 12x speedup over MapReduce re-computation. Fig. 2 shows the normalized runtime of the four iterative algorithms while 10% of input data has been changed. "1" corresponds to the runtime of PlainMR recomp. For PageRank, iterMR reduces the runtime of PlainMR recomp by 56%. i^{2}MapReduce improves the performance further with fine-grain incremental processing and change propagation control (CPC), achieving a speedup of 8 folds (i2MR w/ CPC). We also show that without change propagation control the changes it will return the exact updated result but at the same time prolong the runtime (i2MR w/o CPC). For SSSP, the performance gain of i^{2}MapReduce is similar to that for PageRank. For Kmeans, small portion of changes in input will lead to global re-computation. Therefore, we turn off the MRBGraph functionality. As a result, i^{2}MapReduce falls back to iterMR recomp. For GIM-V, both plainMR and HaLoop run two MapReduce jobs in each iteration, one of which joins the structure data and the state data. In contrast, our general-purpose iterative support removes the need for this extra job. iterMR and i^{2}MapReduce see dramatic performance improvements.

References

[1] D. Peng and F. Dabek, "Large-scale incremental processing using distributed transactions and notifications," in *OSDI '10*, 2010, pp. 1–15.

[2] D. Logothetis, C. Olston, B. Reed, K. C. Webb, and K. Yocum, "Stateful bulk processing for incremental analytics," in *SOCC '10*, 2010.

[3] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: A timely dataflow system," in *SOSP '13*, 2013, pp. 439–455.

[4] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquin, "Incoop: Mapreduce for incremental computations," in *SOCC '11*, 2011.

[5] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "imapreduce: A distributed computing framework for iterative computation," *J. Grid Comput.*, vol. 10, no. 1, 2012.

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