i^{2}MapReduce: Incremental Iterative MapReduce

Yanfeng Zhang
Computing Center
Northeastern University, China
zhangyf@cc.neu.edu.cn

Shimin Chen
Institute of Computing Technology
Chinese Academy of Sciences
chensm@ict.ac.cn

ABSTRACT

Cloud intelligence applications often perform iterative computations (e.g., PageRank) 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 i^{2}MapReduce 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. i^{2}MapReduce 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. i^{2}MapReduce sees significant performance improvement over re-computing iterative jobs in MapReduce.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed applications

General Terms

Algorithms, Design, Experimentation, Performance

Keywords

Incremental processing, Iterative computation, MapReduce

1. INTRODUCTION

Iterative computations are widely used in cloud intelligence applications, such as the well-known PageRank [4] algorithm in web search engines, gradient descent [1] algorithm for optimization, and many other iterative algorithms for applications including recommender systems [2] and link prediction [10]. In the era of "big data", iterative computations often process a large amount of data and take hours or even days to complete. As new data (e.g., changes to web graph) are being constantly collected, the previous iterative computation results (e.g., PageRank scores) become stale and obsolete over time. It is thus desirable to periodically refresh the iterative computation. The shorter the refresh period is, the more timely the results can reflect the fresh data, and thereby the better the user experience is. For example, in this way, new changes in the web graph can be quickly reflected in the computed PageRank and hence web search results. Newly established friendship on a social network can be immediately utilized for behavior targeting and recommendation. Unfortunately, it is too expensive to re-compute the entire iterative job from scratch on the up-to-date data set. The ability to perform incremental processing on mutating data sets is very important, which is the focus of this paper.

Today, MapReduce [6] is the most prominent platform for big data analysis in the cloud. Several previous studies extend MapReduce for efficient iterative computation [5, 7, 20]. However, our preliminary result shows that starting a new iterative computation from scratch is still extremely expensive. McSherry et al. proposed Naiad [14] for incremental iterative computations based on differential dataflow, which is a new computation model that is drastically different from the MapReduce programming model. To the best of our knowledge, for the most popular platform, MapReduce, no solution has so far been demonstrated to be able to efficiently handle incremental data changes for complex iterative computations. A new MapReduce-compatible model that supports incremental iterative computation is desired.

In this paper, we propose i^{2}MapReduce that extends the MapReduce platform to support incremental iterative computations. We introduce a *Map-Reduce Bipartite Graph* (MRBGraph) model to represent iterative and incremental computations, which contains a loop between mappers and reducers. A converged iterative computation means that the MRBGraph state is stable. We observe that in many cases, only a very small fraction of the underlying data set has changed, and the newly converged state is quite close to the previously converged state. Based on this observation, we present the design and implementation of i^{2}MapReduce to efficiently utilize the converged MRBGraph state to perform incremental updates. An existing MapReduce application needs only slight code modification to take advantage of i^{2}MapReduce.

The rest of the paper is structured as follows. Section 2 provides the necessary background. Section 3 describes our incremental processing approach on MRBGraph. We propose i^{2}MapReduce in Section 4. Section 5 reviews the related work in the literature. Finally, Section 6 concludes and outlines the future work.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
Cloud-I '13, August 26 2013, Riva del Garda, Trento, Italy
Copyright 2013 ACM 978-1-4503-2108-2/13/08 ...$15.00.

2. PRELIMINARIES

Iterative Computation. An iterative algorithm typically performs the same computation on a data set in every iteration, generating a sequence of improving results. The computation of an iteration can be represented by an update function F:

$$v^k = F(v^{k-1}, D),$$

where D is the input data set, and v is the result set being computed. After initializing v with a certain v^0, the iterative algorithm computes an improved v^k from v^{k-1} and D in the k-th iteration. This process continues until it converges to a fixed point v^*. In practice, this means that the difference between the result sets of two consecutive iterations is small enough. Then the iterative computation will return the converged result v^*. Note that while v is updated in every iteration, D is static in the computation. We refer to D as the static *structure data*, and v as the dynamic *state data*.

For example, the well-known PageRank algorithm [4] iteratively computes the PageRank vector R that contains the ranking scores of all pages in a web graph, using the following update function:

$$R^{(k)} = dWR^{(k-1)} + (1-d)E,$$

where W is a column-normalized matrix that represents the web linkage graph, d is a damping factor, and E is a size-$|V|$ vector denoting each page's preference. The PageRank score vector R is updated iteratively, while the web linkage graph matrix W stays the same across iterations. R is state data and W is structure data. The typical convergence condition for PageRank is that the manhattan distance between $R^{(k)}$ and $R^{(k-1)}$ is less than a threshold. Then, $R^{(k)}$ is returned as the final result.

Iterative Computation in MapReduce. Using vanilla MapReduce, a user has to submit a series of MapReduce jobs for an iterative algorithm. An iteration often requires at least one MapReduce job. The map function processes the state data v^{k-1} as well as the structure data D, while the reduce function combines the intermediate data to produce the updated state data v^k, which will be stored in the underlying distributed file system to be used as the input to the next job that implements the next iteration. Since every job incurs significant overhead for job start-up and distributed file access, MapReduce is known to poorly support iterative computation.

Previous studies [7, 5, 20] have extended MapReduce for efficient iterative computation. The improvements mainly lie in two aspects: (i) building an internal data flow within a single MapReduce job by sending the reduce output directly to the map; and (ii) caching iteration-invariant data (*i.e.*, structure data D). The former reduces job start-up costs, while the latter avoids the cost of re-reading D in every iteration.

3. INCREMENTAL ITERATIVE PROCESSING ON MRBGRAPH

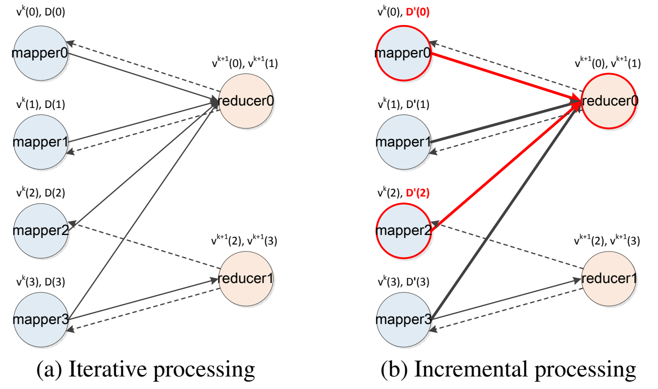
The structure data (e.g., the web graph) in the iterative computation often changes over time. Suppose D becomes $D' = D + \Delta D$. We would like to update the result v^* to reflect this change periodically. The iterative update function becomes:

$$v^k = F(v^{k-1}, D + \Delta D).$$

In many cases, $|\Delta D| \ll |D|$, *i.e.* the structure data is only slightly changed. The converged fixed point $v^{*\prime}$ is often also only slightly different from the previous fixed point v^*. Therefore, it is a good idea to start the incremental iterative computation on the changed structure data from the previously converged state data v^* rather than from an arbitrary initial point v^0.

In the following, we introduce an MRBGraph model to represent iterative computations on MapReduce, and describe our ideas to realize incremental iterative processing using MRBGraph.

Map-Reduce Bipartite Graph. In order to implement the internal loop in MapReduce, reducers' outputs are sent back to the correlated mappers. We model this behavior as a Map-Reduce Bipartite Graph (*MRBGraph*) as shown in Figure 1a. A mapper operates on a state data record $v^k(i)$ and a structure data record $D(i)$. A reducer operates on the intermediate data and produces the updated



(a) Iterative processing (b) Incremental processing

Figure 1: Map-Reduce bipartite graph.

state data records $v^{k+1}(i)$, which are sent back to the correlated mappers or replicated to several mappers for the next iteration.

In an MRBGraph, there are two types of vertices, the mapper vertices and the reducer vertices. The edges from mappers to reducers (*MR-Edge*) represent the shuffled intermediate data, while the edges from reducers to mappers (*RM-Edge*) represent the iterated state data. The iterative computation continues to refine the MRBGraph state iteratively, including the MR-Edge state and the RM-Edge state. When the MapReduce iterative algorithm converges, the MRBGraph state becomes stable. We will exploit the converged MRBGraph state to perform incremental processing.

Incremental Iterative Processing on MRBGraph. As discussed at the beginning of this section, starting from the previously converged state data v^* will accelerate the incremental iterative computation. In an MRBGraph, v^* is the converged RM-Edge state.

In the first iteration starting from v^*, since the input structure data set D is only slightly changed with ΔD, we perform the necessary computation only for the changes. ΔD may add new data, remove data, and/or modify data in D. Therefore, we only perform mappers for these changes, which will affect the MR-Edge state. As illustrated in Figure 1b, the highlighted $D(0)$ and $D(2)$ are affected by ΔD. Therefore, we only need to perform mapper0 and mapper2. Then, the reducers on the receiving end of the changed MR-Edges (e.g., reducer0 in Figure 1b) will be performed. The affected reducers may combine previously converged MR-Edge state from the un-affected MR-Edges (e.g., from mapper1 and mapper3 in Figure 1b) with the updated MR-Edge state. The first iteration completes when the output of the reducers are sent back to the correlated mappers. In the next iteration, a portion of the dynamic state data on RM-Edges may have changed from the previous iteration. Consequently, the mappers corresponding to the changed RM-Edges need to be performed. As the iteration continues, the number of the affected RM-Edges and MR-Edges will increase. Finally, a new converged state $v^{*\prime}$ is reached.

Intuitively, the process is like throwing a few pebbles (ΔD) into a lake of still water (stable MRBGraph). The pebbles first affect the lake surface where they drop (correlated MR-Edges). Then they create ripples, which gradually propagate and affect more lake surface (more RM-Edges and MR-Edges). After a short while, the ripples are absorbed by the lake (in a new stable MRBGraph).

The above idea aims to reduce uncessary computation as much as possible. A map function is performed only when its input state data or structure data are different from the past iteration. A reduce function is performed only when the state of an incoming MR-Edge changes. The challenge is to efficiently preserve and utilize MRBGraph state to realize this idea, while minimizing modifications to the MapReduce programming model to reduce users' programming efforts. We present the design of i^{2}MapReduce in the following.

4. i^2MAPREDUCE

We describe our incremental iterative processing framework based on Hadoop MapReduce.

MapReduce Extension for Building MRBGraph. In the vanilla MapReduce, the life cycle of a map/reduce task ends when all its input data has been processed. To build the loop in the MRBGraph, i^2MapReduce makes map/reduce tasks persistent during the iterative computation. The system sends reduce tasks' output back to the correlated map tasks. When a map/reduce task consumes all its currently assigned data, it goes into an idle waiting state until it is waken up to process new data in subsequent iterations. In this way, the constructed MRBGraph is capable of performing iterative computation in even a single MapReduce job.

Moreover, the input arguments of a vanilla map function is a key-value pair of $\langle MK, MV\rangle$. For iterative computation, MV contains both the state data (v) and the structure data (D). However, the two types of data have very different characteristics. Therefore, i^2MapReduce distinguishes them by splitting the value argument into two. Now, a map function takes $\langle MK, DV, SV\rangle$, where DV is the dynamic state value and SV is the structure value for the key MK. Users are required to implement the new map interface.

MRBGraph State Preserving. In order to support incremental processing, we first preserve the converged MRBGraph state, including both the RM-Edge state and the MR-Edge state. The RM-Edge state is the final reduce output, which is sent back and recorded at the correlated map tasks. i^2MapReduce records $\langle MK, DV\rangle$, where DV is the converged dynamic state data and MK is the mapper key.

The MR-Edge state is the intermediate results communicated from map tasks to reduce tasks and is recorded at reduce tasks. The intermediate results consist of key-value pairs $\langle RK, RV\rangle$s, where RK is the group-by key and determines the destination reducer. Hadoop stores the intermediate results using an IFile format. In addition, MR-Edge state also requires the source mapper. Therefore, i^2MapReduce extends the Hadoop IFile format to record $\langle RK, MK, RV\rangle$, where MK is the source mapper key.

Incremental Processing. After a period of time, part of the input structure data D is changed. i^2MapReduce performs incremental processing starting from the converged state data.

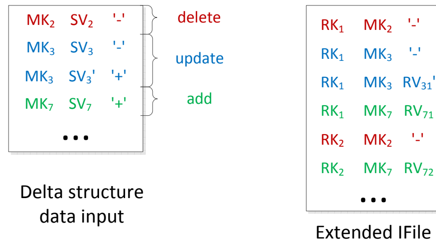


Figure 2: An examples input file and an example extended IFile in incremental processing.

In the first iteration, i^2MapReduce processes only the changed structure data ΔD rather than the whole updated structure data D'. As shown in Figure 2, the changes are specified with a delta structure data input. '$-$' denotes the deletion of a $\langle MK, SV\rangle$ and '$+$' denotes a new $\langle MK, SV\rangle$. An update to an existing structure record can be specified with a $\langle MK, SV, '-'\rangle$ followed by a $\langle MK, SV, '+'\rangle$. For each MK in the delta structure data input, i^2MapReduce retrieves the corresponding $\langle MK, DV\rangle$ from the preserved converged RM-Edge state, and calls the map function. The map output for a $\langle MK, SV, '-'\rangle$ will be the deleted MR-Edges (in the form of $\langle RK, MK, '-'\rangle$), and that of a $\langle MK, SV, '+'\rangle$ will be the updated/new MR-Edges (in the form of $\langle RK, MK, RV\rangle$).

After data shuffling, a reduce task receives the intermediate data that reflects insertion/deletion/update of MR-Edges, *i.e.*, a set of $\langle RK, MK, '-'\rangle$s and $\langle RK, MK, RV\rangle$s. i^2MapReduce performs reduce computation only for the RKs that appear in this set. Note that for the other RKs, their MR-Edge state stays the same and therefore the computed reduce output would be the same as the previously preserved RM-Edge state. For a RK that appears, the system retrieves and merges the preserved MR-Edges with the received intermediate data as follows: i) the received deleted MR-Edges are removed from the preserved MR-Edge set; ii) the received new MR-Edges are added to the preserved MR-Edge set; and iii) the received updated MR-Edges are used to replace the corresponding preserved MR-Edges. Then i^2MapReduce updates the preserved state with the merged MR-Edges, forms the intermediate value list, and calls the reduce function. The reduce outputs, *i.e.*, the changed RM-Edge state, will be sent back to correlated map tasks.

Subsequent iterations work similarly as the first iteration. The only difference is that i^2MapReduce performs map computation for MK that appears not only in the delta structure data input but also in the changed RM-Edge state that is sent from reduce tasks.

Fine-grained Convergence Detection. Normally, convergence is detected by computing the difference between the subsequent reduce outputs, e.g., when the Manhattan distance between $R^{(k)}$ and $R^{(k-1)}$ in PageRank is below a global threshold. However, portions of the MRBGraph may converge faster. We propose an optimization that allows fine-grained convergence detection per RM-Edge. Users can specify a fine-grained filter threshold. If the difference of the RM-Edge state between two iterations is smaller than the filter threshold, the RM-Edge is considered converged. For example, in PageRank, the filter threshold can be set as the global threshold divided by the number of pages. If the Mahattan distance between two subsequent PageRank scores of a page is smaller than the filter threshold, we consider the PageRank score for the page has converged. The converged RM-Edge can be excluded from the next iteration's computation[1].

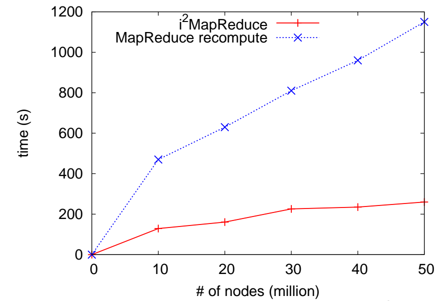


Figure 3: Incremental PageRank performance of i^2MapReduce and MapReduce both initialized with the previously converged state.

Preliminary Results. We have implemented the basic functionalities of i^2MapReduce by extending Hadoop. Figure 3 shows preliminary results in the context of the PageRank application. The X-axis varies the number of nodes in web graphs. We generate synthetic web graphs with the node in-degrees following the power-law distribution. After PageRank converges, we randomly change 10 thousand edges in the synthetic web graphs. The Y-axis reports the performance of incremental processing on a 20-node cluster. We compare i^2MapReduce with MapReduce recomputation that is

[1] Note that as computation goes on, it is possible that a previously converged RM-Edge sees larger changes than the filter threshold. When this occurs, the RM-Edge state will be sent to mappers again.

also initialized with the previously converged state. We see that i^{2}MapReduce outperforms MapReduce recomputation by up to a 5X speedup. This shows that i^{2}MapReduce can effectively reduce unnecessary computation to speed up incremental processing.

5. RELATED WORK

Iterative Processing. A series of distributed frameworks have recently emerged for large-scale iterative computation in the cloud [17, 12, 13, 15, 8, 18, 19]. We discuss the frameworks that improve MapReduce. HaLoop [5], a modified version of Hadoop, improves the efficiency of iterative computations by making the task scheduler loop-aware and by employing caching mechanisms. Twister [7] employs a lightweight iterative MapReduce runtime system by logically constructing a reduce-to-map loop. Our previous work, iMapReduce [20], supports iterative processing by directly passing the reduce outputs to map and by distinguishing variant state data from the static data. i^{2}MapReduce builds upon iMapReduce to further support incremental iterative processing.

One-time Incremental Processing. Several recent studies propose techniques to support efficient incremental processing on one-time computation in the cloud. Stateful Bulk Processing [11] was proposed to address the need for stateful dataflow programs. It provides a flexible, groupwise processing operator that takes state as an explicit input to support incremental analysis. Incoop [3] is a generic MapReduce framework for incremental computations. It detects changes to the input and automatically updates the output by employing an efficient, fine-grained result re-use mechanism. IncMR [16] supports MapReduce incremental processing by combining the old state with new data. Map tasks are created according to new splits instead of entire splits while reduce tasks fetch their inputs including the state and the intermediate results of new map tasks. [9] recomputes MapReduce results in an incremental fashion by adapting view maintenance techniques, which provides a general solution for the incremental maintenance of MapReduce programs that compute self-maintainable aggregates. Rather than one-time computation, i^{2}MapReduce addresses the challenge of supporting incremental processing for iterative computation.

Incremental Iterative Processing. Naiad [14] is recently proposed to support incremental iterative computations. It is based on a new model called differential computation, which extends traditional incremental computation to allow arbitrarily nested iteration. Naiad is designed as a dataflow system, while we extend the popular MapReduce model for incremental iterative computation. Existing MapReduce programs can be slightly changed to run on i^{2}MapReduce framework for incremental processing.

6. CONCLUSIONS AND OUTLOOK

In this paper, we described i^{2}MapReduce, a MapReduce-based framework for incremental iterative computations in the cloud. We are currently fine tuning the implementation and investigating several optimization techniques to further improve the performance. First, the MRBGraph state is saved in disk files. Query and update of MRBGraph state will result in large amount of disk I/Os, which can become performance bottleneck of the system. We are working on an efficient index structure for the MRBGraph state to improve the I/O performance. Second, our current implementation considers only one-to-one mapping between the state records and the structure records (*i.e.*, each mapper operates on a state record and a structure record). There exist other types of mapping relationships. For example, In the K-means clustering algorithm, all the structure records (e.g., points in K-means) are correlated to all the state records (e.g., centroids in K-means). We are extending i^{2}MapReduce to support more complex state-to-structure relationships. Last but not least, different iterative applications have different CPU requirements and will result in different I/O costs. A one-size-fit-all execution plan might not be suitable for all applications. We are studying cost-aware execution optimization that intelligently uses the MRBGraph state and selects the optimal execution strategy based on online cost analysis.

7. ACKNOWLEDGMENTS

This work was supported by Fundamental Research Funds for the Central Universities (N120416001, N120816001), China Mobil Labs Fund (MCM20122051), and MOE-Intel Special Fund of Information Technology (MOE-INTEL-2012-06).

8. REFERENCES

[1] M. Avriel. *Nonlinear programming: analysis and methods*. Courier Dover Publications, 2003.

[2] S. Baluja, R. Seth, D. Sivakumar, Y. Jing, J. Yagnik, S. Kumar, D. Ravichandran, and M. Aly. Video suggestion and discovery for youtube: taking random walks through the view graph. In *Proc. of WWW '08*, pages 895–904, 2008.

[3] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquin. Incoop: Mapreduce for incremental computations. In *Proc. of SOCC '11*, 2011.

[4] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Comput. Netw. ISDN Syst.*, 30:107–117, April 1998.

[5] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. Haloop: efficient iterative data processing on large clusters. *PVLDB*, 3(1-2):285–296, Sept. 2010.

[6] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proc. of OSDI '04*, 2004.

[7] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: a runtime for iterative mapreduce. In *Proc. of MAPREDUCE '10*, 2010.

[8] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl. Spinning fast iterative data flows. *PVLDB*, 5(11):1268–1279, July 2012.

[9] T. Jörg, R. Parvizi, H. Yong, and S. Dessloch. Incremental recomputations in mapreduce. In *Proc. of CloudDB '11*, 2011.

[10] D. Liben-Nowell and J. M. Kleinberg. The link-prediction problem for social networks. *JASIST*, 58(7):1019–1031, 2007.

[11] D. Logothetis, C. Olston, B. Reed, K. C. Webb, and K. Yocum. Stateful bulk processing for incremental analytics. In *Proc. of SOCC '10*, 2010.

[12] 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. *PVLDB*, 5(8):716–727, Apr. 2012.

[13] 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. of SIGMOD '10*, 2010.

[14] F. McSherry, D. G. Murray, R. Isaacs, and M. Isard. Differential dataflow. In *Proc. of CIDR '13*, 2013.

[15] S. R. Mihaylov, Z. G. Ives, and S. Guha. Rex: recursive, delta-based data-centric computation. *PVLDB*, 5(11):1280–1291, July 2012.

[16] C. Yan, X. Yang, Z. Yu, M. Li, and X. Li. Incmr: Incremental data processing based on mapreduce. In *Proc. of CLOUD '12*, 2012.

[17] 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. of NSDI '12*, 2012.

[18] Y. Zhang, Q. Gao, L. Gao, and C. Wang. Priter: A distributed framework for prioritized iterative computations. In *Proc. of SOCC '11*, 2011.

[19] Y. Zhang, Q. Gao, L. Gao, and C. Wang. Accelerate large-scale iterative computation through asynchronous accumulative updates. In *Proc. of ScienceCloud '12*, 2012.

[20] Y. Zhang, Q. Gao, L. Gao, and C. Wang. imapreduce: A distributed computing framework for iterative computation. *J. Grid Comput.*, 10(1):47–68, Mar. 2012.