International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395 -0056
Volume: 03 Issue: 06 | June-2016 www.irjet.net p-ISSN: 2395-0072

# IMPROVING MAPREDUCE FOR MINING EVOLVING BIG DATA USING TOP K RULES

**Vishakha B. Dalvi[1], Ranjit R. Keole[2]**

*[1]CSIT, HVPM's College of Engineering & Technology,*
*SGB Amravati University, Maharashtra, INDIA*
*[2] Professor, CSIT, HVPM's College of Engineering & Technology,*
*SGB Amravati University, Maharashtra, INDIA*

---

**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 refreshing mining results. It utilizes previously saved states to avoid the expense of re-computation from scratch. This paper proposes i[2]MapReduce, a novel incremental processing extension to MapReduce, the most widely used framework for mining big data. Compared with the state-of-the-art work on Incoop, (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, 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. In this paper i[2]MapReduce is evaluated using a one-step algorithm and four iterative algorithms with diverse computation characteristics. The proposal uses a modified version of the A-priori algorithm, named as Top K rules, which finds and recommends only the best K rules of the system. Experimental results on Amazon EC2 show significant performance improvements of i[2]MapReduce compared to both plain and iterative MapReduce performing re-computation.*

***Key Words***: Top K rules, MapReduce, Data mining, Key value pairs, Incremental processing,

## 1. 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] 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. This paper will focus on improving MapReduce. 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 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. The realization of this principle in the context of the MapReduce computing framework is investigated. A number of previous studies (including Percolator [7], CBP [8], and Naiad [9]) have followed this principle and designed new programming models to support incremental processing.

On the other hand, Incoop [10] 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. This paper proposes i²MapReduce, an extension to MapReduce that supports fine-grain incremental processing for both one step and iterative computation. Compared to previous solutions, i²MapReduce incorporates the following three novel features:

- Fine-grain incremental processing using MRBG-Store. Unlike Incoop, i²MapReduce supports kv-pair level fine-grain incremental processing in order to minimize the amount of re-computation as much as possible. This paper models the kv-pair level data flow and data dependence in a MapReduce computation as a bipartite graph, called MRBGraph. A MRBG-Store is designed to preserve the fine-grain states in the MRBGraph and support efficient queries to retrieve fine-grain states for incremental processing.

- General-purpose iterative computation with modest extension to MapReduce API. Previous work proposed iMapReduce [6] 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. In comparison, this paper provides general-purpose support, including not only one-to-one, but also one-to-many, many-to-one, and many-to-many correspondence. The system enhances the Map API to allow users to easily express loop-invariant structure data, and proposes a Project API function to express the correspondence from Reduce to Map. While users need to slightly modify their algorithms in order to take full advantage of i²MapReduce, such modification is modest compared to the effort to re-implement algorithms on a completely different programming paradigm, such as in Percolator [7], CBP [8], and Naiad [9].

- 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 [1]. To address this problem, this paper proposes to reuse the converged state from the previous computation and employ a change propagation control (CPC) mechanism. This paper also enhances the MRBG-Store to better support the access patterns in incremental iterative processing. To our knowledge, i²MapReduce is the first MapReduce-based solution that efficiently supports incremental iterative computation.

Researchers implemented i²MapReduce by modifying Hadoop-1.0.3. Researchers evaluate i²MapReduce using a one-step algorithm (A-Priori) and four iterative algorithms (PageRank, SSSP, Kmeans, GIM-V) with diverse computation characteristics. Experimental results on Amazon EC2 show significant performance improvements of i²MapReduce compared to both plain and iterative MapReduce performing re-computation. For example, for the iterative PageRank computation with 10 percent data changed, i²MapReduce improves the run time of re-computation on plain MapReduce by an eight fold speedup [1]. This paper uses a modified version of the A-priori algorithm, named as Top K rules, which finds and recommends only the best K rules of the system, not considering the redundant rules, and giving only the rules which are better for describing the system behavior.

## 2. SURVEY RELATED DETAILS

Previous work Incoop, [10] 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.

Previous work proposed iMapReduce, [6] 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.

Previous work Incoop, [10] supports incremental one-step processing. 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.

In the previous work [1] the researchers have developed fast techniques for evolving data, and its mapping. But the reduction part still needs improvement. In the work, they have described various mapping and reducing techniques, but if reduction is not optimized then the overall system efficiency is low and might lead to a slow response for a real time system.

Previous works have following problems:

- Does not supports key-value pair level incremental processing and supports only task level incremental processing.
- Does not supports General-purpose iterative computation and only supports one-to-one/all-to-one correspondence from Reduce output to Map input.
- Does not supports incremental processing for iterative computations and only supports incremental one-step processing.
- Speed of the mining process is low.

## 3. PROPOSED WORK

Current paper proposes, a system which overcomes the drawback of slower reduction times, and uses a method which reduces the input data faster as compared to any proposed algorithm, thereby improving the overall efficiency of the system. The proposal uses a modified version of the A-priori algorithm, named as Top K rules, which finds and recommends only the best K rules of the system, not considering the redundant rules, and giving only the rules which are better for describing the system behavior and giving the best possible recommendations for the system. This method will improve the overall speed and accuracy of the rule mining process and make the entire Map Reduce structure perform in real time with highest level of accuracy. Proposed approach works in the following manner,

**Step 1:** Collection of evolving datasets

- The evolving datasets will be collected for mapping and reduction.

**Step 2:** Development of mapping technique

- MapReduce allows for distributed processing of the map and reduction operations. Provided that each mapping operation is independent of the others, all maps can be performed in parallel – though in practice this is limited by the number of independent data sources and/or the number of CPUs near each source.
- Map function Maps input key/value pairs to a set of intermediate key/value pairs. Maps are the individual tasks which transform input records into intermediate records. The transformed intermediate records need not be of the same type as the input records. A given input pair may map to zero or many output pairs. Map() is run exactly once for each *K1* key value, generating output organized by key values *K2*.

**Step 3:** Development of reduction technique

- A set of 'reducers' can perform the reduction phase, provided that all outputs of the map operation that share the same key are presented to the same reducer at the same time, or that the reduction function is associative. Reduce() is run exactly once for each *K2* key value produced by the Map step.

### Logical View of MapReduce process

The *Map* and *Reduce* functions of *MapReduce* are both defined with respect to data structured in (key, value) pairs. *Map* takes one pair of data with a type in one data domain, and returns a list of pairs in a different domain:

Map(k1,v1) → list(k2,v2)

The *Map* function is applied in parallel to every pair in the input dataset. This produces a list of pairs for each call. After that, the MapReduce framework collects all pairs with the same key from all lists and groups them together, creating one group for each key.

The *Reduce* function is then applied in parallel to each group, which in turn produces a collection of values in the same domain:

Reduce(k2, list (v2)) → list(v3)

Each *Reduce* call typically produces either one value v3 or an empty return, though one call is allowed to return more than one value. The returns of all calls are collected as the desired result list.

Thus the MapReduce framework transforms a list of (key, value) pairs into a list of values.

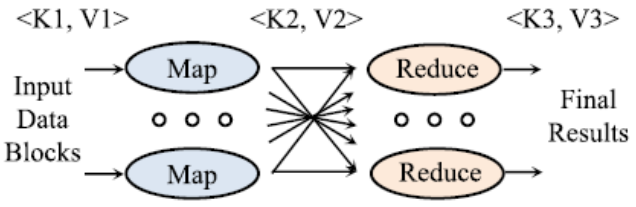


**Fig -1:** MapReduce Computation

**Step 4:** Improvement in reduction technique using Top K Rules

Improvement is done in reduction techniques using Top K Rules algorithm which is the modifies version of a-priori algorithm. The Top K Rules algorithm works as follows:

- The TopKRules algorithm takes as input a transaction database, a number *k* of rules that the user wants to discover, and the *minconf* threshold.
- The algorithm main idea is the following. TopKRules first sets an internal *minsup* variable to 0. Then, the algorithm starts searching for rules. As soon as a rule is found, it is added to a list of rules L ordered by the support. The list is used to maintain the top-k rules found until now. Once *k* valid rules are found, the internal *minsup* variable is raised to the support of the rule with the lowest support in L. Raising the *minsup* value is used to prune the search space when searching for more rules. Thereafter, each time a valid rule is found, the rule is inserted in L, the rules in L not respecting *minsup* anymore are removed from L, and *minsup* is raised to the value of the least interesting rule in L. The algorithm continues searching for more rules until no rule are found, which means that it has found the top-k rules.
- To search for rules, TopKRules does not rely on the classical two steps approach to generate rules because it would not be efficient as a top-k algorithm (as explained in the introduction). The strategy used by TopKRules instead consists of generating rules containing a single item in the antecedent and a single item in the consequent. Then, each rule is recursively grown by adding items to the antecedent or consequent. To select the items that are added to a rule to grow it, TopKRules scans the transactions containing the rule to find single items that could expand its left or right part. Two processes for expanding rules in TopKRules are *left expansion* and *right expansion*. These processes are applied recursively to explore the search space of association rules.
- Another idea incorporated in TopKRules is to try to generate the most promising rules first. This is because if rules with high support are found earlier, TopKRules can raise its internal *minsup* variable faster to prune the search space. To perform this, TopKRules uses an internal variable R to store all the rules that can be expanded to have a chance of finding more valid rules. TopKRules uses this set to determine the rules that are the most likely to produce valid rules with a high support to raise *minsup* more quickly and prune a larger part of the search space.

**Step 5:** Result Analysis and Comparison

- The result of algorithm will be analyzed and will be compared to existing results.

## 4. CONCLUSIONS

The first model uses i²MapReduce, which combines a fine-grain incremental engine, a general-purpose iterative model, and a set of effective techniques for incremental iterative computation. The new model uses a modified version of the A-priori algorithm, named as Top K rules, which finds and recommends only the best K rules of the system. Compared with the first model, the new model is much more efficient and achieved the satisfactory performance as well. The main objective of this paper was to throw some light on the proposed work. It provides a promising methodology to improve the overall speed and accuracy of the rule mining process and make the entire Map Reduce structure perform in real time with highest level of accuracy by using Top K Rules.

## REFERENCES

[1] Yanfeng Zhang, Shimin Chen, Qiang Wang, and Ge Yu, "i²MapReduce: Incremental MapReduce for Mining Evolving Big Data", IEEE Transactions On Knowledge And Data Engineering, Vol. 27, No. 7, July 2015.

[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] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

## BIOGRAPHIES

**Vishakha B. Dalvi** received B.E. degree in Computer Science and Engineering from H.V.P.M's college of Engineering and Technology, Amravati in 2014. She is currently pursuing M.E. degree in Computer Science and Information Technology from H.V.P.M's college of Engineering and Technology, Amravati.

**Prof. Ranjit R. Keole** received the B.E. and M.E. degree in Computer Science from Prof. Ram Meghe Institute of Technology, Badnera in 1992 and 2008, respectively. His field of specialization is web Mining. He is currently working as Associate Professor at H.V.P.M's college of Engineering and Technology, Amravati.