Available Online at www.ijcsmc.com

International Journal of Computer Science and Mobile Computing

A Monthly Journal of Computer Science and Information Technology

ISSN 2320–088X

IJCSMC, Vol. 5, Issue. 2, February 2016, pg.154 – 158

Effective Handling of Data Using Parallelized Incremental Approach

Ms. G.Sowmya, Mr. M.Karthikeyan

Department of Computer Science and Engineering, Sri Krishna Collage of Engineering and Technology, Coimbatore, Tamil Nadu, India
sowmyagurusamy17@gmail.com, karthikeyanm@skcet.ac.in

Abstract— Map reduce is one among the necessary functionality of Hadoop tool kit which enables the users to evaluate the large volume of data at run time. Handling large volume of data with less computational cost is the important features of map reduce frame work. In previous system, incremental map reduce was introduced which focus to eliminate the computation cost of users by using already processed input data for the new computation instead of re-computing from scratch. KV pair level is used, in which the input data that might affects the output would be processed. However, this works lacks in reducing the computational cost in the data set which is collected from various sources with different characteristics. To address this limitation, Parallelized incremental clustering is introduced which can process the data that are gathered from multiple sources in run time by using online aggregation. The proposed system is used to support large volume of data in run time further reducing time complexity.

Keywords— iterative processing, incremental processing

I. INTRODUCTION

Now a days a huge quantity of digital knowledge is being accumulated in several necessary areas, as well as e-commerce, social network, finance, health care, education, and setting. It has become more and more widespread to mine such massive knowledge in order to achieve insights to assist business choices or to provide higher quality services. In recent years, an oversized range of computing frameworks [1], [2], [3],[4], [5], [6], are developed for giant knowledge analysis. Among these frameworks, Map Reduce [1] is that the most widely employed in production because of its simplicity, generality, and maturity. This paper focuses on improving map reduce.

Big data is continually evolving. As new information and updates are being collected, the data will gradually change and also the total results can become hard over time. In many things, it's fascinating to periodically refresh the mining computation so as to stay the mining results in Up-to-date. As an example, the Page Rank algorithmic computes ranking much web content supported the online graph structure for supporting net search. However, the web graph structure is continually unfolded; web pages and hyper-links are created, deleted, and updated. As the underlying net graph evolving, the Page Rank ranking results become stale, probably lowering the quality of net search. Therefore, it's fascinating to refresh the Page Rank computation often.

Incremental process could be a promising approach to refreshing mining results. Given the scale of the input massive data, it's typically terribly expensive to rerun the whole computation from scratch. The input data of two subsequent computations A and B are similar. Only an small fraction of data is changed. The concept is to avoid wasting states in computation A, re-use A's states in computation B and perform re-computation only for states that area affected by the modified input data.

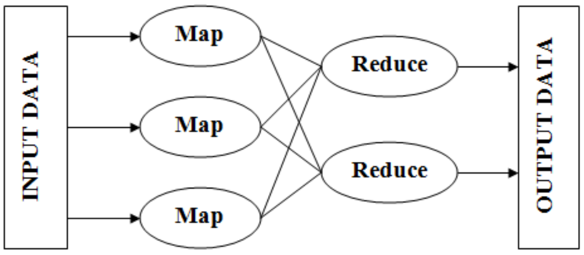

Fig 1: Map Reduce

II. MRBG STORE

The MRBG-Store supports the states for incremental process. First, the MRBG-Store should incrementally store for a modified 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 new computed edge changes. MRBGraph file stores fine-grain intermediate states for a reduce task. In Fig2, we tend to see that the (K2; MK; V2) with the same K2 are kept contiguously in chunk. Since a chunk corresponds to the input of the reduce instance. It treats a chunk as the basic unit and always reads, writes, and operates on entire chunks. The contents of a delta MRBGraph file square measure shown on the bottom left of Fig.2, each record represents a modified state within the original MRBGraph. There are two forms of records, an edge insertion record and deletion record. The merging of the delta MRBGraph with the MRBGraph file in the MRBG-Store is actually a part of operation victimization K2 as a join key. Since the scale of the delta MRBGraph is typically smaller than the MRBGraph file, it's wasteful to scan the whole MRBGraph file. Therefore, we tend to construct an index for selective access to the MRBGraph file. Given a K2, the index returns the chunk position within the MRBGraph file. As only point is needed, we employ a hash-based implementation for the index. The index is kept in associate degree index file and is preloaded into memory before reduce computation. We tend to apply the index nested loop as part of merging operation. MRBGraph and MRBGraph file are sorted in K2 order. There are two ways to read the record K2, first an individual I/O operation on each chunk and second large I/O operation that cover all chunks. Second choice will result in reading lot of useless data, because the files are sorted in K2 order. The read window size can be estimated based on cost estimation.

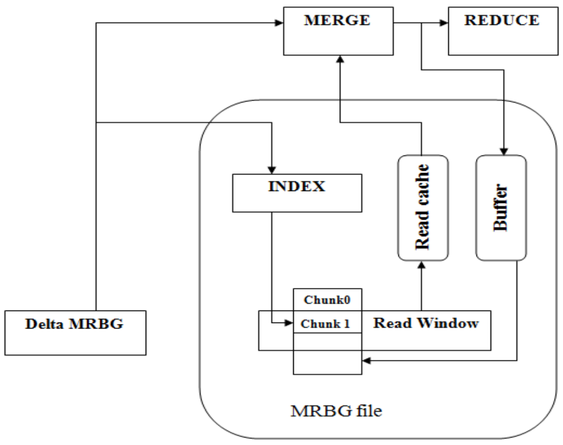

Fig 2: MRBG store

III. RELATED WORKS

J.Dean and S.Ghemawat [1], deals with key value pairs in map reduce. MapReduce[1] could be a programming model and an associated implementation for process and generating massive information sets. One of the common causes that the total time taken for a MapReduce operation is a straggler [1]. The computation takes a set of input key/value pairs, and produces a set of output key/value pairs. The user of the MapReduce library expresses the computation as two functions: Map and Reduce. Map, written by the user, takes an input pair and produces a set of intermediate key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key (I) and passes them to the Reduce function. The Reduce function, also written by the user, accepts an intermediate key (I) and a set of values for that key. It merges these values to form a possibly smaller set of values. Typically just zero or one output value is produced per Reduce invocation. The intermediate values are supplied to the user's reduce function via an iteration. This allows us to handle lists of values that are too large to fit in memory.

The MapReduce programming model has been successfully used in different purposes due to: (i) the model is easy to use, even for programmers without experience with parallel and distributed systems, since it hides the details of parallelization, fault-tolerance, locality optimization, and load balancing. (ii) A large variety of problems are easily expressible as MapReduce computations.(iii) it have developed an implementation of MapReduce that scales to large clusters of machines comprising thousands of machines. Though it has many advantages it fails if there is a single master. This map reduce process is used in this MRBG store in order to partition the large amount data.

Y.Bu,B.Howe,M.Balazinska and M.D.Ernst [3], deals with the extremely climbable parallel processing platforms. MapReduce[1] may be a well-known framework for programming clusters to perform large-scale processing during a single pass. There are 2 key issues with manually orchestrating iterative program [2] in MapReduce. The primary downside is that although the information is unchanged from iteration to iteration, the information should be re-loaded and re-processed at each iteration wasting in I/O, network information measure, and mainframe resources. The second downside is that the termination condition could involve detection once a fix purpose has been reached, i.e., when the application's output doesn't change for two consecutive iterations. This condition itself need an additional Map Reduce job on each iteration overhead in terms of programming extra tasks, reading further knowledge from disk, and moving knowledge across the network. HaLoop[2] inherits the essential distributed computing model and design of Hadoop[1]. HaLoop depends on a distributed filing system (HDFS) that stores every job's input and output knowledge. The system is split into 2 parts: one master node and lots of slave nodes. Client submits jobs to the master node. For every submitted job, the master node schedules variety of parallel tasks to run on slave nodes. Each slave node features a task track process to communicate with master node and manage every task's execution. Every task is either a map task or reduce task. The main disadvantage is that, they are sensitive to failures and have not been shown to scale of thousands of nodes. Even though iterative computation is used in MRBG store to reduce unwanted data fetching from DFS.

y.Zhang,Q.Gao,L.Gao,andC.Wang[5],deals With the iterative computation. These relative information usually Contain millions of records. iMapReduce[3]supports the iterative processing of large relational data and addresses all the issues in the MapReduce[1]. Each MapReduce job has to load data from Distributed File System [2] before the map operation. After the map operation operates to derive intermediate key value pairs, the reduce function operates on the intermediate data, and derives the output of the iteration, which will be written in DFS. In iteration, map tasks load data from DFS and repeat the process. Finally, the iteration terminates when the termination condition is satisfied. Here a termination condition depends on the data size which is stored in MRBG file. iMapReduce does the termination check after each iteration. If the termination condition is satisfied, the master will notify the entire map and reduce tasks to terminate the execution.

Some of the advantages by using this iterative process are (i) it provides a framework for programmers to explicitly model iterative algorithms.(ii) it proposes the concept of persistent tasks to perform the iterative computation to avoid repeatedly creating, destroying, and scheduling tasks. (iii) The input data are loaded to the persistent tasks once and do not need to be shuffled between map and reduce. This can significantly reduce the I/O and the network communication overhead and the processing time. (iv) It facilitates asynchronous execution of tasks within the same iteration, to accelerate the processing speed. But this iterative process requires an additional MapReduce task in each iteration and programmers have to explicitly specify the join operation. Additionally, the repeated DFS loading/dumping are expensive.

D.Peng and F.Dabek[6], deals with the task of building an index of the web that can be used to answer search queries. The indexing system starts by crawling every page on the web and processing them while maintaining a set of invariants on the index. A Percolator system [4] consists of three binaries that run on every machine in the cluster: a Percolator worker, a big table tablet server, and a Google File System [4] chunk server. All observers are linked into the Percolator worker, which scans the big table for changed columns and invokes the corresponding observers as a function call in the worker process. The observers perform transactions by sending read/write Remote Procedure Call [4] to big table tablet servers, which in turn send read/write RPCs to GFS chunk servers. The system also depends on two small services: the timestamp oracle and the lightweight lock service. A Percolator repository consists of a small number of tables; each table is a collection of "cells" indexed by row and column. Each cell contains a value.

The design of Percolator was influenced by the requirement to run at massive scales and lack of a requirement for extremely low latency. Percolator has no central location for transaction management; in particular, it lacks a global deadlock detector. This increases the latency of conflicting transactions but allows the system to scale thousands of machines.

IV. IMPLEMENTATION

Iterative and incremental computation together called as i^2mapreduce.This i^2mapreduce significantly reduce the run time for refreshing big data results.

A. MRBGraph Abstraction

Map reduce bipartite graph is employed that produce the distinctive key for every map instance such as <K1, V1> as a map function call and <K2, V2> as a reduce function call. An edge from a Map instance to a Reduce instance means that the Map instance generates a <K2, V2> that is shuffled to become part of the input to the Reduce instance. An edge contain three piece 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, i2MapReduce generates a globally unique Map key for each Map instance. Therefore, i2MapReduce will preserve (K2, MK, V2) for each MRBGraph edge.

B. Fine Grain incremental Processing

Fine grain incremental process invokes the Map operates for each record within the delta input. For an insertion with '+', its intermediate results <K2, MK, V2> represent recently inserted edges within the MRBGraph. For a deletion with '-', its intermediate results indicate that the corresponding edges are deleted from the MRBGraph. The delta MRBGraph will contain the changes to the MRBGraph and sorted by K2 order. For each affected K2, the merge list of V2 will be used as input to invoke the reduce function to generate the updated final results.

C. MRBG Store

The MRBG-Store supports the preservation and retrieval of fine-grain MRBGraph states for incremental processing. There are two main requirements on the MRBG-Store. (i) 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. (ii) the MRGB-Store must support efficient retrieval of preserved states of given Reduce instances. For incremental Reduce computation, i2MapReduce re-computes the Reduce instance associated with each changed MRBGraph edge. 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. Fine-grain state retrieval and merging work together to achieve the above two requirements. An MRBGraph file stores fine-grain intermediate states for a Reduce task 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 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 wasteful 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, hash-based implementation is used for the index. The index is stored in an index file and is preloaded into memory before Reduce computation. The index nested loop used for the merging operation.

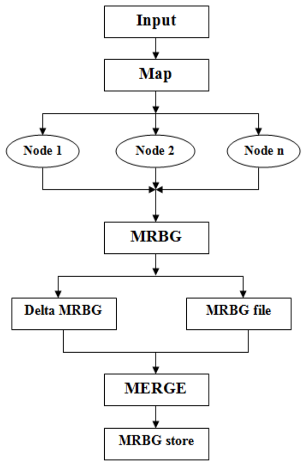

Fig 3: i^2 map reduce

V. CONCLUSION

Map reduce is most featured component in the Hadoop simulation toolkit which is used to handle the large volume of data in an efficient manner. The performance of the map reduce toolkit might get reduced in case of processing the same kinds tasks again and again. This is due to immediate deletion of stored data in catch once it forwarded to the reduced phase. This is resolved in this work by introducing the incremental and iterative algorithm which buffers the intermediate generated results for the particular period. The buffer would be refreshed in the periodic manner to support the large volume of data. There are different iterative processing is used and some of them are very sensitive to failures. Runtime iterative is one process where it update the output at runtime and it act as centralized approach, but it lacking in finding errors. Another iterative processing, where it reduce computation but increase communication. Finally a distributed iterative and incremental approach is combined to increase the performance.

VI. FUTURE WORK

In future work, the data's gathered from the multiple nodes can be supported, by clustering them based on the k-means algorithm and map reduce bipartite graph algorithm. Hence the system performance can be improved significantly.

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] 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.

[3] 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.

[4] 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.

[5] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "imapreduce: A distributed computing framework for iterative computation," Oper. Syst. Des. Implementation, 2010, pp. 1–15.J. Grid Comput., vol. 10, no. 1, pp. 47–68, 2012.

[6] D. Peng and F. Dabek, "Large-scale incremental processing using distributed transactions and notifications," in Proc. 9th USENIX Conf.Oper.Sysr.Des.Implementation,2010,pp.1-15.