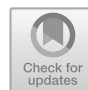
Efficient MapReduce Framework Using Summation



Sahiba Suryawanshi and Praveen Kaushik

1 Introduction

BigData can be defined as huge quantity of data, in which data is beyond the normal database software system tool to capture, analyze, and manage. The data is within the limits of three dimensions which are data volume, data variety, and data velocity [1]. Primary analysis data contains surveys, observations, and experiences; and secondary analysis data contains client information, business information reports, competitive and marketplace information, business information, and location information that contains mobile device information. Geospatial information and image information contains a video and satellite image and provides chain information containing rating and vendor catalogs, to store and process this information that is done by BigData. To process this variety of data, the velocity is incredibly necessary.

The major challenge is not to store the big datasets in our systems, but, to retrieve and analyze the large data within the organizations, that too, for information stored in various machines at completely different locations [1]. Hadoop comes in a picture in these situations. Hadoop has been adopted by many people leading companies, for example, Yahoo!, Google, and Facebook along with various BigData programs, for example, machine learning, bioinformatics, and cybersecurity. Hadoop has the power to analyze the info very quickly and effectively. Hadoop works best on semi-structured and unstructured data. Hadoop has MR and Hadoop distributed file system [2]. The HDFS can provide a storage for clusters, and once the info is stored within the HDFS then it breaks into number of small pieces and distributes those small items into number of servers that are present within the clusters, wherever each server stores

S. Suryawanshi (✉) · P. Kaushik
Department of Computer Science and Engineering, Maulana Azad National Institute of Technology (MANIT), Bhopal, India
e-mail: sahiba686@gmail.com

P. Kaushik
e-mail: kaushikp@manit.ac.in

© Springer Nature Singapore Pte Ltd. 2019

R. K. Shukla et al. (eds.), *Data, Engineering and Applications*,
https://doi.org/10.1007/978-981-13-6351-1_1

these small pieces of whole information set and then for each piece of information a copy stored on more than one server, this copied information set will be retrieved once the MR is process and within which one or a lot of *Mapper* or *Reducer* fails to process [3].

MR appeared as the preferred computing framework for large processing because of its uncomplicated programming model and the execution is done in parallel automatically. MR has two computational phases, particularly *mapping* and *reducing*, that is successively carried through many *maps* and *reduce* tasks unalike. The *map* reads input data and manages to create <*key, value*> pairs depending on the input data. This <*key, value*> pairs are the intermediary outputs within the native machine. Within the *map* phase, the tasks begin in parallel which generates <*key, value*> pairs of intermediate data by the input splits. The <*key, value*> pairs are kept on the native (local) machine and well ordered into various data partitions as one for each *reducer* phase. *Reducer* is liable for processing the intermediary outcomes which receive from various *mappers* and generating ultimate outputs within the *reducer* phase. Every *reducer* takes their part of information from data partitioning coming from all the *mapper* phases to get the ultimate outcome. In between the *mapper* phase and the *reducer* phase, there is a phase, i.e., *shuffle phase* [4]. During this, the info created at the *mapper* phase is ordered, divided, and shifted to the suitable machine execute the *reducer* phases. The MR is performed over a distributed system composed of the master and a group of workers. The input is split into chunks that are allocated to the *mapper* phases [5]. The *map* tasks are scheduled by a master to the workers, which consider the data locality. The *map* tasks provide an output which will be divided into a number of pieces according to the number of *reducers* for the job. Record with the similar intermediary *key* should go to the same partition so that it will guarantee the correctness for the execution. All the intermediary <*key, value*> pairs' partitions are arranged and delivered with the task of *reducer* that needs to be executed. By default, the constraint of data locality does not take into consideration while doing the scheduling tasks for the *reducer*. As the result, the quality of data at the *shuffle* phase, which needs to transfer via a network, is significant. Using tradition, a hash-based function which is used for partitioning the intermediary data in *reducer* tasks is not traffic-efficient because topologies of network and size of data corresponding to each key are not considering it. Thus, this paper proposes a scheme which sums up the intermediate data. The proposed scheme will reduce the data size that has to be sent to the *reducer*. By reducing the size of data, the network traffic at reduce function will minimize. Even though the combiner also performs the same function, the combiner operates on the generated data by *map* task individually which thus fails to operate between the multiple tasks [4]. For summing up the data, it put summation function. Summation function can be put in both either within the single machine or among different machines. For finding the best suitable place for summation function, it uses distributed election algorithm. At the *shuffle* phase, the summation function will work simultaneously, and then it removes the user-irrelevant data. In this, the data of volume and traffic is reduced up to 55%, and then it sends to the *Reducer*. It is more efficient way to process the data, for those jobs which have hundreds and thousands of key ends, and each of the keys is associated with number of values.

The rest of the paper is organized as follows. In Sect. 2, we review recent related work. Section 3 provides the proposed model. Section 4 analyzes the result. Finally, Sect. 5 concludes the paper.

2 Literature Survey

In this section, different techniques for optimization generally applied in the *MapReduce* framework and BigData are discussed. The paper also discussed the attributes of various techniques for optimization and how BigData processing is improved by these techniques.

In [6], the author examined that whether the network optimizing can make a better performance of the system and realize that utilizing the high network and low congestion in a network; good performance can also be achieved parallel with a job in the optimizing network system. In [7], the author gives purlieus, a system which allocates the resources in MR, which will increase the MR Job's performance in the cloud, via positioning intermediary data to the native machines or nearer to the physical machines. This reduces the traffic of data within the *shuffle* part produced in the cloud data center. In [8], paper designs a good key partition approach, in which the distribution of all frequencies of intermediate keys is watched, and it will guarantee that the fair distribution in *reduce* tasks, in which it inspects the partitioning of intermediary key and distribution of data with the key and its respective value among all the machines of *map* to *reducer*, for the data correctness, is also examined. In [9], the author gives two effective approaches (load balancing) to skew the data handling for MR-based entity resolution. In [10], the author proposes MRCGSO mode; it adjusts very well with enlarging data set sizes and the optimization for speedup is very close. In [11], the author relates the parallel and distributed optimization algorithm established on alternating direction method of a multiplier for settling optimization problem in adapting communication network. It has instigated the authorized framework of the extensive optimization problem and explains the normal type of ADMM and centers on various direct additions and worldly modifications of ADMM to tackle the optimization problem. In [12], the author pays attention to accuracy using cross-validation; the paper gives sequential optimize parameters for plotting a conspiracy of accuracy. In [13], the author gives a method to initiate Locality-Sensitive Bloom Filter (LSBF) technique in BigData and also discusses how LSBF can be used for query optimization in BigData. In [14], the author initiates the optimization algorithms using these rules and models can deliver a moderate increase to the highest productivity on inter-cloud and intra-cloud transfers. Application-level transfer adapts parameters just like analogousness pipelining, and concurrency is very needful mechanisms for happening across data transfer bottlenecks for scientific cloud applications, although their optimal values juncture on the environment on which basis the transfers are generally done. By using actual models and algorithms, these can spontaneously be optimized to achieve maximum transfer rate. In [15], the author offers an algorithm for cache consistency bandwidth

optimization. In this perspective, the user data shift is optimized without consideration of the user expectation rate. The debated algorithm differentiates with trending data transmission techniques. In [16], the author recommends improved computing operators focused on smart grids optimization for BigData through online. This settles a problem of generic-constrained optimization by utilizing a module based on the MR framework, which is a trending computing platform for BigData management commenced at Google. Online answer to urged optimization problems is a necessary requirement for a secure, reliable smart grid operation [17]. Many authors proposed methods for optimizing MR, but very less work is done for optimizing MR by reducing the traffic generated, while the data is sent to *reducer* phase. So we proposed a method which is based on distributed summation.

3 Proposed Method

In Hadoop MR, normally the *reduce* task resides in different machines/racks. As the massive intermediate data goes to *reduce* task, it will create heavy traffic in the network. By analyzing the intermediate output, we saw that there are redundant *<key, value>* pairs. So we can sum up all the similar *<key, value>* pairs before sending it to the *reducer*, which will minimize the quantity of intermediate data. The *summation* can be done in both either within the machine or among different machines.

Summation at single machine: The *summation* function can put at each machine in which it will sum up the similar *<key, value>* pairs within the same machine before it is sent to the *reducer.* As a result, the data from each machine will minimize by *summation* function that will minimize the traffic too.

Summation among different machines: When the *summation* function put at each machine, it will reduce the size of data. But for massive data, it needs many *mappers* and *reducers*. There may chances that at *reducer* there might be number of inputs even though the inputs are already minimized by *summation*. But due to number of *map* functions, it will also create traffic at *reducer*. For that, it will sum up data among different machines.

In Fig. 1a, it needs to send three rows of data, wherein Fig. 1b it needs to send two rows of data to the *summation* function reside in different machines. As a result, if the position of node where the *summation* is done will change, the traffic cost will also change; it is the extra challenge to handle.

Architecture: Hadoop has a master node (Job Tracker), and number of slave nodes (Task Trackers) located on remaining nodes. The job tracker handles all the submitted jobs and takes the decision for scheduling and parallelizing the work across the Hadoop cluster. And the task trackers do the work in parallel, allotted by the job tracker. For summating the intermediate data, it needs two things; one piece of code for summation, i.e., *summation phase,* and a manager who will handle the location of that code. The manager resides in job tracker, which has the information where the *summation code* will place for more efficient processing. This architecture will minimize the network traffic in *shuffle* phase.

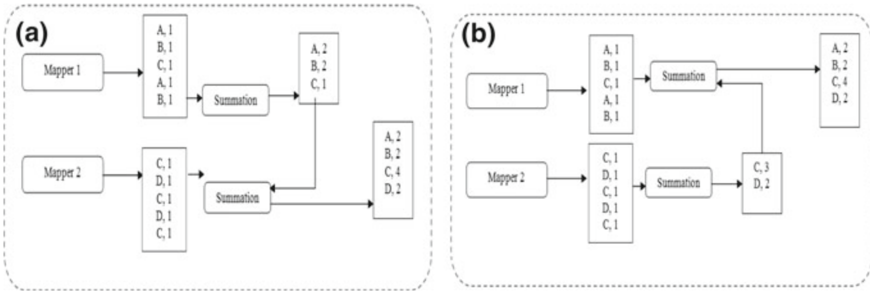


Fig. 1 MapReduce using summation among different machines

Summation phase: In the framework, the *summation phases are* located between *shuffle* phases and *reducer* phases. All the intermediate data acts as input to this and generated output is sent to the *reducer*. It performs the summation of similar <*key, value*> pairs in such a way that each key has single summated pair value. After that, output of *summation phase* along with the similar key has to be delivered to a single *reducer.* In the architecture, the execution of summation is managed by the task tracker at each node. When the manager who is placed in job tracker sends the request to generate the *summation code* at task tracker, task tracker will initialize the instance and specify the tasks attached to the request. Finally, when the task is completed, the *summation code* is removed by task tracker and conveys the message to the manager.

Manager: The manager has two main issues—where the *summation code* resides and routes so that the summated intermediated data will generate less amount of traffic.

Summation code placement—Number of *summation codes* will be generated to reduce the traffic along with the path; the path will define from where the intermediate data will go among different machines. To do this, manager has two main questions; by answering that, it will minimize the traffic during *shuffle* phase:

- On which machine the *summation code* will generate for minimum traffic?
- To which machine the intermediate data come from different machines, i.e., what is the route?

For answering these questions, manager needs the whole information about the *map* and *reduce* function along with the positions and the frequency of intermediate data (volume). Furthermore, manager also requires the information about the resources of slave nodes, i.e., the availability of memory and CPU for processing of summation. All these information will be sent by the task tracker to job tracker with the heartbeat. It is sent by task tracker to job tracker so that job tracker has knowledge of whether the task tracker alive or not; here, alive means its working condition. According to that information, the manager will send the info about the *summation code* and the route. The manager has information about the positioning of all the task trackers, so it will send the request to find the central node for summing

up the data in different machines by creating small clusters. As the slave nodes get the request, one of them elects itself as the central node and finds whether any of other is interested, finds the central node, and informs the manager. Algorithms 1 and 2 are as follows:

Assume the clusters are connected to each other in a ring form; each node can send information to its next node only so that all nodes have the information and it will create less traffic. But if there is any node failure, then it will bypass it.

Distributed algorithm does not assume the existence of the previous central node; every time according to the requirement, it will change which depends on the frequencies of <*key, value*> pairs and the availability of resources. It will choose a node among a group of different nodes in different machines as a central node.

Assume each node has their own IDs, and the priority of node *Ni* is *i*, which is calculated according to the availability of resources and the frequency of <*key, value*> pairs, i.e., volume.

Background: Any node *Ni*(among the nodes to which manager sends the function for processing data) tries to find any other active node which is more suitable, by sending a message; if no response comes in *T* time units, *Ni* tries to elect itself. Details are as follows:

Algorithm 1 for sender node *Ni* that select suitable node

1. *Ni* sends an "*Elect Ni*" with *Pi*
2. *Ni* waits for time *T*
    a. If *Pi* < *Pj* it receives "*elect Nj*"
        i. update central node as *Nj*
    b. If no response comes then *Ni* will be selected as the central node.

Algorithm 2 for node receiver *Nj*

1. *Nj* receive "*elect Ni*"
2. If *Pj* > *Pi*
    a. Send "*elect Nj*"
3. Else forward "*elect Ni*"

4 Implementation

The performance baseline is provided by the original scheme (i.e., no summation is provided) and by the proposed method. We create an Oracle VM virtual machine; we configure it with the required parameters (here we use two processors, 3 GB RAM, 20 GB memory) and settings to act as a cluster node (especially the network settings). This referenced virtual machine is then cloned as many times as there will be nodes in the Hadoop cluster. Only a limited set of changes are then needed to finalize the node to be operational (only the hostname and IP address need to be defined). We have created pseudo-cluster. Our prototype has been implemented on Hadoop 2.6.0.0.

In Fig. 2, it gives output sizes of *mappers* on nodes for respective actual sized. It executes the proposed algorithm using the same data source for comparison in Hadoop environment. It shows the data size for different files of size 250 MB, 500 MB, 750 MB, and 1 GB are minimized after applying the *summation* and the file size reduced to 180 MB, 220 MB, 335 MB, and 450 MB, respectively. As a result, the reduction ratios are 28%, 32%, 48%, and 55% for file size of 250 MB, 500 MB, 750 MB, and 1 GB, respectively. The proposed method will work more efficiently as the size increases, and thus it will work well for BigData.

To compute, the capability of the proposed algorithm by comparing traditional hash-based function is shown here. Hash-based partition without *summation*, as default method in Hadoop, makes the traditional hash partitioning for the intermediate data, which are sent to *reducers* without summing up intermediate data. And our proposed method is *summation* within same machine and *summation* among different machines, in which before sending to *reduce* function the intermediate data will be summed up so that it will minimize the size (traffic); sometimes, after summing up, intermediate data at each machine data at *reducer* will be huge because of many *mappers*, so it can be minimized if the summing up will be done among different machines also as per requirements. In Fig. 3, the performance is shown which takes the same file and performs the traditional method, summation on single machine and *summation* among different machines. Here, if the number of keys is increased, the traffic in *shuffle* phase also increases. For example, for 20 keys, traditional, *summa-*

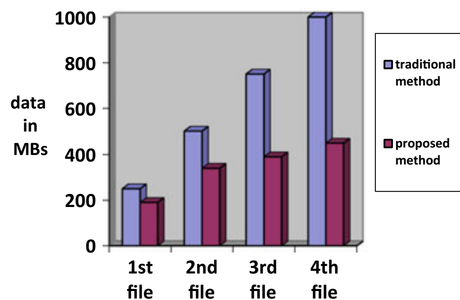


Fig. 2 Data size at reducer

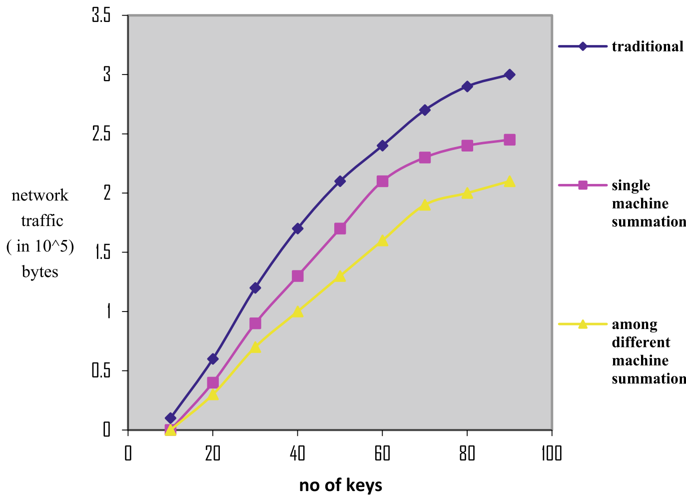


Fig. 3 Traffic cost

tion in single, and *summation* in different machines generate 0.6 * 10ˆ5, 0.4 * 10^5, and 0.3 * 10^5 bytes, respectively.

5 Conclusion

The significance of proposed scheme is discussed, i.e., summation in Hadoop MR to process the BigData that minimizes the network traffic produced by intermediate data: intermediate data is output of *map* function. For verification, we have given an architecture where summation functions can be easily attached to the existing MR framework. How the positioning of summation code among various machines will affect the size is also shown, for which we give a distributed election algorithm that fills to find the best suitable positions for the central summation function among various machines. By applying the proposed method, it will reduce the size of data up to 55%.

The implantation for the proposed scheme is in a pseudo-cluster for computing the behavior; it can compute on heterogeneous distributed clusters.

References

1. Philip Chen, C.L., Zhang, C.-Y.: Data-intensive applications, challenges, techniques and technologies: a survey on big data (2014). Science Direct
2. Apache Haddop HDFS Homepage. http://HADOOP.apache.org/hdfs

3. White, T.: Hadoop: The Definitive Guide, 1st edn. O'Reilly Media (2009)
4. Patel, A.B., Birla, M., Nair, U.: Addressing big data problem using Hadoop and map reduce. IEEE (2013)
5. Ke, H., Li, P., Guo, S., Guo, M.: On traffic-aware partition and aggregation in MapReduce for big data applications. IEEE Trans. Parallel Distrib. Syst. (2015)
6. Blanca, A., Shin, S.W.: Optimizing network usage in MapReduce scheduling (2013)
7. Palanisamy, B., Singh, A., Liu, L., Jain, B.: Purlieus: locality-aware resource allocation for MapReduce in a cloud. ACM (2011)
8. Ibrahim, S., Jin, H., Lu, L., Wu, S., He, B., Qi, L.: Leen: locality/fairness-aware key partitioning for MapReduce in the cloudm. IEEE (2011)
9. Hsueh, S.-C., Lin, M.-Y., Chiu, Y.-C.: A load-balanced MapReduce algorithm for blocking-based entity-resolution with multiple keys (2014)
10. Al-Madi, N., Aljarah, I., Ludwig, S.A.: Parallel glowworm swarm optimization clustering algorithm based on MapReduce. In: 2014 IEEE Symposium on Swarm Intelligence (2014)
11. Liu, L., Han, Z.: Multi-block ADMM for Bigdata optimization in smart grid. IEEE (2015)
12. Liu, Y., Du, J.: Parameter optimization of the SVM for Bigdata. In: 2015 8th International Symposium on Computational Intelligence and Design (ISCID) (2015)
13. Bhushan, M., Singh, M., Yadav, S.K.: Bigdata query optimization by using locality sensitive bloom filter. IJCT (2015)
14. Ramaprasath, A., Srinivasan, A., Lung, C.-H.: Performance optimization of Bigdata in mobile networks. In: 2015 IEEE 28th Canadian Conference on Electrical and Computer Engineering (CCECE) (2015)
15. Ramprasath, A., Hariharan, K., Srinivasan, A.: Cache coherency algorithm to optimize bandwidth in mobile networks. Lecture Notes in Electrical Engineering, Networks and Communications. Springer Verlag (2014)
16. Yildirim, E., Arslan, E., Kim, J., Kosar, T.: Application-level optimization of Bigdata transfers through pipelining, parallelism and concurrency. In: IEEE Transactions on Cloud Computing (2016)
17. Jena, B., Gourisaria, M.K., Rautaray, S.S., Pandey, M.: A survey work on optimization techniques utilizing map reduce framework in Hadoop cluster. Int. J. Intell. Syst. Appl. (2017)