Vijay Dahiphale

1 Binghamton University

June 07, 2024
Dynamic Split Size: A Novel Approach for Bolstering Map Side Aggregation in MapReduce and Future Ideas

Vijay Dahiphale*

Binghamton University, USA

Email: vdahiphale@binghamton.edu*

Abstract—MapReduce efficiently processes large data sets in a Petabyte/Exabyte scale distributed system. Applications such as reverse indexing, image and pattern recognition, and analytics rely extensively on this programming model for large-scale distributed data processing. MapReduce works by splitting the processing into two distinct phases: a Map phase, which distributes data processing, and a Reduce phase, which aggregates results. While open-source applications such as Apache Hadoop and its query engine "Hive" implement MapReduce, some challenges persist. Most crucially, high network I/O is incurred when Mappers write many rows. One solution (attempted by Apache Hive) is to pre-aggregate data at the Mapper and send this aggregated data to the Reducer. This attempts to reduce network I/O. However, this is insufficient for two reasons: first, Reducers are starved while Mappers aggregate data at Map-Side, and second, Mappers can run into memory overflow and need a re-compute if pre-aggregated results exceed their buffer capacity. We propose a design to solve these problems using an adaptive split size for Map-side pre-aggregation. By dynamically adjusting the size of the input data split for a Mapper, we ensure that (1) Reducers get a continuous stream of data without starving and (2) Mapper memory overflows are avoided, which also avoids the need to "flush and re-compute" earlier results.

Index Terms—Big Data, MapReduce, Hadoop, Apache Hive, Mappers, Reducers, Distributed, Streaming, Aggregation, Network, Dynamic, Computing, Feedback, Optimization.

I. INTRODUCTION

In today’s data-driven world, algorithms, and applications are collecting data about almost everything: people, processes, systems, organizations, and their myriad digital interactions. As a result, we generate a huge amount of data daily. However, data is only meaningful to the extent that it can be analyzed, and insights can be derived from it. This is where the MapReduce framework comes in.

MapReduce is a programming model and an associated implementation for processing big data sets with a parallel, distributed algorithm on a cluster [4]. MapReduce was initially developed and used by Google for analyzing its search results. However, it eventually gained massive popularity due to its ability to split and process terabytes of data in parallel. As a result, it achieved quicker results in data analysis, indexing, and querying applications. MapReduce facilitates concurrent processing by breaking petabytes of data into smaller chunks and processing them in parallel on Hadoop commodity servers. Ultimately, it aggregates all the data from multiple servers to return a consolidated output to the application.

There are many implementations of the MapReduce model, such as Associated MapReduce [1], MapReduce Online [8], Apache Spark [9], Cloud MapReduce [10], Hadoop [6], Disco [11], MapReduce-MPI [12], MARIANE [13], MARISSA [14] and Phoenix [15]. The most common and widely used is Hadoop. Hadoop Online Prototype (HOP) [8] is a modification to the traditional Hadoop implementation of MapReduce. In conventional Hadoop, Map, and Reduce phases run sequentially - The map phase finishes before the Reduce phase can begin. In other implementations (e.g., Hadoop Online Prototype), the Map and the Reduce phases run simultaneously, speeding up processing.

In MapReduce, Map workers are computing nodes that do the task of filtering and sorting records, and Reduce workers aggregate records generated by Mappers. Therefore, the record aggregation only happens on the Reducers’ side. To make aggregation efficient, Apache Hive proposes the Map-Side aggregation approach (in-Map aggregation with Hashmap or Hashmap-like structure) for avoiding or alleviating a very high network I/O at the Reducers side. Without Map-Side aggregation, all rows/records (Mappers’ output) must be sent to Reducers, causing high network I/Os. However, for large data sets, Map-side aggregation often results in memory shortages due to in-memory Hashmaps used by Mappers.

We propose a novel solution for solving two main problems 1. Map workers are running out of memory due to Map-Side aggregation. 2. Starvation of Reduce workers. We further propose a mechanism to describe how the Primary node can use this approach [1] [6] for improving the scheduling of Mappers, Reducers, and all other workers required by MapReduce. In addition, we identify and highlight the advantages of Map-Side aggregation in general for all implementations of MapReduce. The paper also introduces ideas for future areas of research.

The paper is organized as follows: Section II begins with a background survey of technologies and services related to our work. Section III is dedicated to our contribution, which primarily includes 1. Spotting current problems, 2. The proposed idea, 3. The architecture of our proposal, and 4. Advantages of Map-Side aggregation in general. Section IV briefly touches
that the Reduce tasks always follow the map tasks. This is to reduce charges, and the data is shuffled and reduced. Note conclusions, and acknowledgments, respectively.

and VII present future research opportunities & enhancements, on the applications of the proposed design. Sections V, VI, and VII present future research opportunities & enhancements, conclusions, and acknowledgments, respectively.

II. BACKGROUND SURVEY

A. Hadoop Distributed File System

Hadoop Distributed File System (HDFS) [6] [17] is a distributed file system that provides reliable and scalable data storage. It is designed explicitly for spanning large clusters on the commodity hardware. The Hadoop uses HDFS for data storage in the backend [8]. The input data, intermediate data (output of Mappers), and final results - are all kept in the HDFS for all the MapReduce jobs.

The figure 1 shows a fundamental architecture of HDFS. HDFS has a Primary/Secondary architecture. An HDFS cluster consists of a single NameNode. The NameNode is a Primary server that manages the file system namespace and regulates access to files by clients. The DataNodes manage storage attached to the nodes that they run on. HDFS exposes a file system namespace and allows user data to be stored in files. The DataNodes are responsible for serving read and write requests from the file system’s clients, and also they perform block creation, deletion, and replication upon instruction from the NameNode.

B. MapReduce

MapReduce [1] is a programming model developed by Google for processing big data sets in a distributed fashion. It operates in two phases:

- Map Phase: In this phase, input data is split into multiple chunks/splits. A Mapper applies a user-defined function (Map function) on these chunks/splits.
- Reducer Phase: This phase aggregates the data produced in the Map phase to generate the final output.

In other words, the output of a map task is used as an input to reduce charges, and the data is shuffled and reduced. Note that the Reduce tasks always follow the map tasks. This is because all the Mappers run in parallel with the other Mappers, and all the Reducers also run in parallel with other Reducers.

For example, the often cited word count example elegantly illustrates the computing phases of MapReduce. A large document is to be scanned, and the numbers of occurrences of each word are to be determined. For example, consider the to be processed contains the text as "Fear leads to anger; anger leads to hatred; hatred leads to conflict; conflict leads to suffering.". The solution for counting the words in the document using MapReduce goes as follows (please refer to relevant figure 2):

1) Input Data and Chunks/Splits: MapReduce splits the input into smaller blocks called input chunks or splits. In MapReduce context, it represents a block of work with a single Mapper task. The information is kept on Hadoop Distributed File System (HDFS) [6]. This, by default, happens in the MapReduce job setup phase.

2) Mappers: The input data is processed and divided into smaller segments in the Map phase, where the number of Mappers equals the number of input chunks for a simple implementation of MapReduce. RecordReader produces a key-value pair (a pair also called a "record") of the input splits using TextFormat, which Reducer later uses as input. The Mapper then processes these key-value pairs using coding logic to produce an output of the same form.

3) Shuffling: In the shuffling phase, the output of the Mapper phase is passed to the Reducer phase by removing duplicate values and grouping the values. The output remains as keys and values in the Mapper phase. Since shuffling can begin even before the Mapper phase is complete, it saves time. This component is not shown in figure 2 for keeping the word count example simple to understand.

4) Sorting: Sorting is performed simultaneously with shuffling. The Sorting phase involves merging and sorting the output generated by the Mapper. The intermediate key-value pairs are sorted by key before starting the Reducer phase, and the values can take any order (however, the applications which require the records to be sorted by values rather than keys eventually reverse the records - key becomes a value and vice versa - but sorting is always done on keys). Sorting by value is done by secondary sorting if required. This component is not shown in figure 2 for keeping the word count example simple to understand.

5) Reducers: In the Reducer phase, the intermediate values from the shuffling phase are reduced/aggregated based on the keys to producing a single output value (or records with unique keys and aggregated value for all the records with the same key) that summarizes the entire dataset. HDFS is then used to store the final output.

Here’s an example of using MapReduce to count the frequency of each word in input data. The data is, "This is the input data content for word count example using MapReduce. Hadoop first divides input data and then the word count."

1) The input data is divided into multiple segments and then processed in parallel to reduce processing time. In this case, the input data will be divided into four input splits to distribute work over all the map nodes (also four in our case).
2) A Mapper counts the number of times each word occurs from the assigned input chunk in key-value pairs where the key is the word, and the value is the frequency. Initially, the Mappers map each word to value 1.

3) For the first input split, it generates the following key-value pairs:

<table>
<thead>
<tr>
<th>Word String</th>
<th>Frequency (Mapped to 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fear</td>
<td>1</td>
</tr>
<tr>
<td>leads</td>
<td>1</td>
</tr>
<tr>
<td>to</td>
<td>1</td>
</tr>
<tr>
<td>anger</td>
<td>1</td>
</tr>
</tbody>
</table>

TABLE I: Word Count Example: Mapper Thread/Worker 1 Output

4) It is followed by the shuffle phase, in which the values are grouped by keys in the form of key-value pairs at the Mapper side (also called combiners work before sending the records over the network).

5) The same Reducer is used for all key-value pairs with the same key, so it becomes practical to aggregate all the records with the same key. All the words in the data are combined into a single output in the Reducer phase. The output shows the frequency of each term.

6) Here in the example, we get the final output of key-value pairs as follows:

<table>
<thead>
<tr>
<th>Word String</th>
<th>Frequency (Mapped to 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>anger</td>
<td>2</td>
</tr>
<tr>
<td>conflict</td>
<td>2</td>
</tr>
<tr>
<td>Fear</td>
<td>1</td>
</tr>
<tr>
<td>hatred</td>
<td>2</td>
</tr>
<tr>
<td>leads</td>
<td>4</td>
</tr>
<tr>
<td>suffering</td>
<td>1</td>
</tr>
<tr>
<td>to</td>
<td>4</td>
</tr>
</tbody>
</table>

TABLE II: Word Count Example: Final Output from the Reducer into the output files, and the final output data is, by default, stored on HDFS or fed back to the application which invoked this MapReduce job.

C. Apache Hadoop

The Apache Hadoop [6] [5] project develops open-source software for reliable, scalable, distributed computing. The Apache Hadoop software library is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models. It is designed to scale from single servers to thousands of machines, each offering local computation and storage.

D. Apache Hive

The Apache Hive [16] gives an SQL-like interface to query data stored in various databases and file systems that integrate with Hadoop. The Apache Hive data warehouse software facilitates reading, writing, and managing large datasets in distributed storage and queried using SQL syntax.
Map-Side aggregation is Apache Hive’s one essential feature that is not there in traditional MapReduce or any of its other implementations. The idea behind map-side aggregations is the same as Hadoop combiners [2], [8]. If a single mapper can yield multiple values for the same key, you can reduce shuffling by reducing values locally. The Map-Side optimization partially aggregates inside the Mapper, which results in the Mapper outputting significantly fewer rows. This reduces the amount of information Hadoop needs to shuffle, sort, and distribute over the network to the Reducers.

Considering the famous word count example for Map-Side aggregation, the Mappers would instead tokenize each row and store partial counts in an in-memory Hashmap at the Mappers side (in other words, the Mappers are holding each key with the corresponding partial aggregation for the token appeared in the input chunk so far, which is just a count of words in this case). Periodically, the Mappers will output the pairs <"word", "word_count"> when the memory buffer limit is reached. The Hadoop framework again sorts these pairs, and the Reducers sum the values to produce the total counts for each word. In this case, each Mapper will output one row for each word every time the partial map output is flushed instead of one row for each occurrence of each term. The tradeoff is that the Mappers must keep a map of all words (tokens) in memory.

E. Online MapReduce

This is famously called Hadoop Online Prototype (HOP) [8]. There are many MapReduce implementations [8] [18] [10], for example Hadoop [6], Apache Spark [9], Disco [11], MapReduce-MPI [12], MARIANE [13], MARISSA [14] and Phoenix [15]. The most common and widely used is Hadoop. Hadoop Online Prototype [8] [2] is a modification to the traditional Hadoop implementation. In traditional Hadoop, Map and Reduce phases run sequentially - first Map phase finishes, and then Reduce phase starts. However, the Map and Reduce phases run simultaneously in Hadoop Online Prototype. As HOP is well-proven, the most popular, and efficient in processing Big Data sets, we decided to use it to process stock market data.

F. Network I/O in MapReduce

In MapReduce, the communication between different workers (Mappers, Reducers, Combiners, Master, task trackers, etc.) [1] [2] [8] [3] happens over the network because these
workers could be running on any machine in a cluster [4]. For optimizing MapReduce and making it more efficient, the optimal use of the network in a cluster and network bandwidth aware scheduling [19] of different workers have become very important and well-researched.

III. PROBLEMS, PROPOSAL, ARCHITECTURE, AND ADVANTAGES

This section first discusses the general advantages of using Map-side aggregation. It then lists the current problems in processing big data using MapReduce. Finally, it proposes our "Dynamic Split Size" solution and discusses the new advantages of using this approach to solve the current problems.

A. Advantages of Map-side Aggregation in General

1) Reduce Network I/O: When a large number of rows are aggregated using Reducers, it can cause significant network traffic. This is because the Reducers need to receive all the data from the mappers before they can begin aggregating it. Map-Side aggregation is a technique that can help reduce network traffic by performing some aggregation work on the mappers. This can significantly improve performance, especially for large datasets.

2) Early Feedback: Recent researchers are working on ways to optimize the MapReduce framework. One approach is to use feedback mechanisms to speculate the output of a job before it finishes. This can be done using Map-Side aggregation, which allows the mappers to aggregate data locally before sending it to the reducers. This can significantly reduce the amount of time it takes to process a job, and it can also help to identify errors early on.

3) Avoid Idle Reducers: This paper’s proposed approach uses Dynamic Split Size modification to tune the chunk size. This ensures that reducers receive a continuous stream of data records for aggregation, which can improve performance.

4) Less Data Transfer over Network: Map-Side aggregation merges multiple records (pairs of keys and values). This dramatically reduces the number of rows or records transferred over the network, helping to save a lot of network bandwidth. This is a handy feature when a cluster has bandwidth constraints.

5) Automatic Mapper Status: In cases where a Mapper only flushes the result at the end of processing assigned data, Reducers can automatically infer the status of the respective Mapper. In some implementations, getting these Mappers’ status is extra overhead, which can be somewhat avoided. However, if a Mapper fails, the status of that Mapper must be explicitly needed.

6) Load Balancing in Aggregation Heavy Computations: If a MapReduce job is aggregation/Reducer-intensive, the mappers finish quickly, but the reducers will take too long. By enabling Map-Side aggregation, the mappers can assist the reducers to some extent, balancing[20] the workload of aggregation between mappers and reducers during processing.

7) Glimpse into the Future Output: By allowing early aggregation at mappers with mapper-level granularity, a user can predict how the output will look to some extent. Moreover, issues in the overall MapReduce jobs can be detected much earlier in the processing, saving a lot of time in cases of experimentation by developers or even naive users.

B. Current Problems

1) Aborting Map-Side Aggregation: The Mappers must abort the Map-Side aggregation when they run out of the allocated memory. The abort implies the flushing of partial in-memory results to the local storage.

Generally, MapReduce suggests aggregating the intermediate results by Reducers. However, considering the advantages (as listed in the previous subsection III-A) of adding Mappers side aggregation, some of the MapReduce implementations like "Apache Hive" incorporate this "Reducers friendly Mappers" approach.

2) Idle Reducers: For the Map-Side aggregation, even if Mappers do not run out of the allocated memory, the delay in map task completion could lead to Reducers being idle and thus wasting processing resources (specifically, in the case of Online or Streaming MapReduce [8] [18], Advanced MapReduce [2] and any other implementation of Mapreduce where Mappers and Reducers run in parallel). This mainly happens because of three main reasons 1) The input chunk size for a Mapper is huge. 2) The Mapper task produces too many rows, even for a small chunk of input data. 3) Network limitation in a target cluster. 4) Processing power constraints on the nodes where the Mappers and Shufflers work. We are mainly focusing on the first two issues from the list. However, the solution can be easily extended to tackle the other two issues.

3) High Bandwidth Usage: In traditional MapReduce and most implementations, aggregation is proposed only at the Reducers. As a result, all the Mappers produce many rows (key-value pairs/records) that must be sent over the network. This could affect network I/O bandwidth and latency, and therefore it can adversely affect the MapReduce job performance. Furthermore, this problem can adversely affect the other jobs or processes (other MapReduce jobs or non-MapReduce jobs) running in the same cluster and network I/O dependent.

4) High Aggregation Overhead: Suppose a MapReduce job is aggregation/Reducer intensive or focused. The Mappers would be finished quickly. However, the Reducers from the job would take too much time to touch a finish line. By enabling Map-Side aggregation, Mappers would assist the Reducers to some extent, leading to more concurrency and balance [20] in the workload.

5) Sluggish Mappers: Suppose a mapper or a few mappers are slow in the current MapReduce infrastructure. As a result, it delays the overall job completion considerably. Currently, many of the solutions do not do anything to tackle slow mappers issues actively. However, with feedback from mapper
tasks to the Application Controller task, the problem of sluggish mappers in a MapReduce job can be proactively tackled. Therefore, our solution can also help optimize MapReduce.

6) A Big Record Size: Hive [16] tries to optimize the query execution by performing map-side aggregations when certain conditions are met in the query execution. We know the trade-off with Map-side aggregations is the hash Map needs to be stored in memory. As a preventive measure to avoid "OutOfMemoryException" in the Mapper by the size explosion in the aggregation hash map, hive.map.aggr.hash.percentmemory can be used to control the flush of the hash map to the reducer wherever hash map size exceeds the percentage specified in the parameter compared to the total memory available to the Mapper. However, this is an estimate based on the number of rows and the expected size of each row, so if the memory usage per row is very high, the mappers may run out of memory before the hash map is flushed to the reducers.

C. Proposed Solution for the Above Listed Problems

We propose a novel approach for dynamically adjusting a Mapper’s input data chunk/split size. Figure 3 gives a gist of our solution. To achieve this, we introduce a feedback [21] loop from Mappers (indirectly the nodes where Mappers are running) to the Primary node (or the scheduler node - a node responsible for scheduling different workers under a MapReduce job). The feedback will be mainly in two forms:
1. The point in time and percentage of input chunk processed by a Mapper before aborting the Map-Side aggregation. 2. How long did a Mapper process an allocated chunk of the input data (this can be calculated at the Job tracker machine node)?

Furthermore, we propose leveraging the communication mechanism between Mappers and the Primary node. In the existing architecture (before our modification), the Mappers send heartbeats [1] (a type of message in the Hadoop framework) to the Primary node to inform that they are alive and still working on their allocated chunk of the data. On a very high level, once a map task gets completed, it notifies the Application Controller (primary node) through heartbeat; the application Controller keeps track of the mapping between map output and hosts. In cases of MapReduce implementation on the top cloud, this communication can be done using a shared database (indirectly). For example, in Cloud MapReduce [10], the Mappers report their status to SimpleDB (Amazon Web Service).

A feedback message from Mappers to the Application controller can be different for different implementations of the MapReduce framework. A unique MapId identifies a Mapper. An Application Controller node differentiates a mapper from others using their IDs. A typical message that should append the current Heartbeat message should look like \(\langle\text{MapId, %Progress, CurrentTime}\rangle\). Moreover, the last message would contain the total time required to finish the assigned chunk of data by the Mapper.

Figure 4 shows an existing architecture for MapReduce primarily with 1. Application Manager 2. Node Managers 3. Primary Node 4. Secondary Nodes, and 5. Important communication links between them. The Heatbeat message link from node managers to the application manager in a MapReduce job is proposed to be leveraged to send the above-specified additional messages. The application managers consult the resource manager before making any decisions. The secondary nodes hold the containers created for running Map and Reduce tasks.

Interpretation of a feedback message at the Application Controller is critical in dynamically deciding the input chunk size and the number of mappers in a MapReduce job. The first form of feedback tells the Primary server to gradually decrease the chunk/split size because the existing chunk size is causing to produce too many rows/records that need to be kept in memory for Map-Side aggregation. Too many rows or records cause the memory overflow at the Mappers. Therefore, Mappers must “flush and re-compute” multiple times before they finish. In other words, by decreasing the chunk size for a Mapper, we can reduce the memory requirement of a Mapper for Map-Side aggregation. The second feedback form tells the Primary server to either gradually increase the chunk size or decrease it depending on the time a Mapper takes to finish. In other words, in cases where the memory at the Mappers side is not a problem, the chunk size will be tuned such that a Mapper finishes in an optimal time (not too quickly or not taking huge time). This will avoid the problem of Reducers being idle.
D. Architecture

The figures 3 and 5 show MapReduce architecture with the proposed design. We primarily introduced the feedback loop between Mappers and the Primary server of a target cluster or a system (the rest of the work remains the same as suggested in the primary MapReduce papers [1] [6]). The Primary server then (as indicated in the "Proposed Solution" subsection - the previous section) uses the information pushed by Mappers to decide input data chunk size for a Mapper to be an optimal one, making the overall system most efficient - and also enabling more applications using the same framework.

Note that we show direct communication arrows between the Mappers and the Master in the architecture shown in figure 3 and 5. However, this communication can be direct or indirect depending on which implementations of MapReduce the user is dealing with. Indirect communication is communication between Mappers and the Master via some shared database. For example, in the case of the Cloud Mapreduce [10] [2] (A Mapreduce implementation on top of Amazon Web Services (AWS) [22]), the Mappers write their status to Simple Database (SimpleDB) [23] (one of the Amazon Web Services). The architectural diagram 3 is labeled with number and explained below:

1) User Program/Application: This application invokes a MapReduce job. Usually, the final output is also expected by the same application that launches the MapReduce job.

2) Primary Node: This is a Primary server - a node responsible for the complete execution of a submitted job. It decides the chunk size (based on user input parameters), spawns different workers for the position, and tracks their progress. Moreover, it makes appropriate decisions in cases of failure.

3) Input Data: The input data usually resides in HDFS. However, it depends on the type of MapReduce implementation one is going to use; for example, in the case of Cloud MapReduce [10], the input data is kept in Simple Storage System (S3) [22] - one of the Amazon Web Services for storing user data.

4) Map Phase: In the Map phase, the worker/threads apply a user-defined Map function on the allocated chunk of data. The generated intermediate output by the Mappers is written to the local disk (HDFS in the case of Apache Hadoop). We mainly incorporate our proposed idea in this phase of the system. The Map function is modified to report a piece of information to the Primary server (this is along with the status Heartbeat [8]). The reported information (along with the Heartbeat of a map worker) will be used by the Primary server to make decisions about deciding the chunk size for the remaining data and schedule the workers who will be launched later in the processing. Map level aggregation also happens in this phase.

5) Intermediate Staging Area: This component represents the storage where the Mappers’ output is stored before it is transferred over the network for further processing by the Reducers.

6) Reducers: The Reducers apply the user-defined Reduce Function on the output of the Mappers. The Reducers aggregate the records in the Reduce phase; the aggregation happens based on the user-defined Reduce Function - which usually aggregates or collapses the records with the same key.

7) Final Output: The final output is again written to HDFS (in the case of Hadoop) or S3 (in the case of Cloud MapReduce). In the case of batch processing, the output is pulled on-demand; however, in the case of streaming analytics, the result is continuously read by the application that launched the MapReduce job.

E. Tuning the Number of Mappers in Hadoop

According to the current Hadoop APIs, the number of Mappers is configured using \texttt{JobConf.setNumMapTasks} API is just a hint to the Hadoop run-time. Configuring the number of map workers per node should be possible by using the \texttt{mapred.tasktracker.map.tasks.maximum} and the \texttt{mapred.tasktracker.reduce.tasks.maximum} in the \texttt{mapred-site.xml} file. This way, it’s possible to configure the total number of mappers executing in parallel across the entire cluster. The input data chunk size indirectly guides the number of Map workers in a MapReduce job. Adding feedback from the Mappers to the primary server would help dynamically adjust the input chunk size and thus guide the number of Map workers.

IV. Applications

This design applies to any problem that fits into the MapReduce paradigm. This will optimize any MapReduce job in general. However, this solution can be more advantageous for a subset of problems where the early results are more crucial to act quickly, for example, in the Distributed Denial of Service (DDoS) [24] attack, the feedback loop from Mappers and Map-Side aggregation can be crucial in early detecting DDoS attack threats.
V. CONCLUSION AND FUTURE WORK

In this paper, first, we identified problems in Map-side aggregation in MapReduce jobs. Next, we proposed a novel “Dynamic Split Size” design for avoiding Map-Side (in-Map) aggregation problems and alleviating the idling of Reducers. We designed the approach and explained the idea in detail. We also identified the general advantages of the Map-Side aggregation for any MapReduce application. Furthermore, we proposed that our system can be extended to improve the performance of the MapReduce framework. Finally, the paper states multiple exciting ideas on a similar line for readers to explore and venture into as a future work.

The authors are currently working on implementing this paper’s proposed design. Adding the feedback loop from Mappers to the Master can be very useful for applying MapReduce such as Distributed Denial of Service Attack (DDoS) [24] - that is, in early threats detection of DDoS. We encourage the readers to explore more such applications of MapReduce and use the proposed design in those applications.

We would love to explore in the future if adding a feedback loop from the Reducers to the Master also helps improve MapReduce performance. The communication between Reducers and the Master would have been similar to that between Mappers and the Master.

VI. ACKNOWLEDGEMENT

We are grateful to the inventors of MapReduce and the Volunteers of Apache Hadoop & Hive implementations who fueled our research into this area. We also thank Dreamz Voluntees of Apache Hadoop & Hive implementations who made the approach and the idea in detail. We also identified the general advantages of the Map-Side aggregation for any MapReduce application. Furthermore, we proposed that our system can be extended to improve the performance of the MapReduce framework. Finally, the paper states multiple exciting ideas on a similar line for readers to explore and venture into as a future work.

The authors are currently working on implementing this paper’s proposed design. Adding the feedback loop from Mappers to the Master can be very useful for applying MapReduce such as Distributed Denial of Service Attack (DDoS) [24] - that is, in early threats detection of DDoS. We encourage the readers to explore more such applications of MapReduce and use the proposed design in those applications.

We would love to explore in the future if adding a feedback loop from the Reducers to the Master also helps improve MapReduce performance. The communication between Reducers and the Master would have been similar to that between Mappers and the Master.

REFERENCES