Co-located Compute and Binary File Storage in Data-intensive Computing

Qiangju Xiao, Pengju Shang, Jun Wang
Department of Electrical Engineering & Computer Science
University of Central Florida
Orlando, United States
Email: qxiao, shang, jwang@eecs.ucf.edu

Abstract—With the rapid development of computation capability, the massive increase in data volume has outmoded compute-intensive clusters for HPC analysis of large-scale data sets due to a huge amount of data transfer over network. Co-located compute and storage has been introduced in data-intensive clusters to avoid network bottleneck by launching the computation on nodes in which most of the input data reside. Chunk-based storage systems are typical examples, splitting data into blocks and randomly storing them across nodes. Records as the input data for the analysis are read from blocks. This method implicitly assumes that a single record resides on a single node and then data transfer can be avoided.

However, this assumption does not always hold because there is a gap between records and blocks. The current solution overlooks the relationship between the computation unit as a record and the storage unit as a block. For situations with records belonging to one block, there would be no data transfer. But in practice, one record could consist of several blocks. This is especially true for binary files, which introduce extra data transfer due to preparing the input data before conducting the analysis. Blocks belonging to a single record are scattered randomly across the data nodes regardless of the semantics of the records.

To address these problems, we develop two solutions in this paper. One is to develop a Record-Based Block Distribution (RBBD) framework and the other is a data-centric scheduling using a Weighted Set Cover Scheduling (WSCS) to schedule the tasks. The Record-Based Block Distribution (RBBD) framework for data-intensive analytics aims to eliminate the gap between records and blocks and accomplishes zero data transfer among nodes. The Weighted Set Cover Scheduling (WSCS) is proposed to further improve the performance by optimizing the combination of nodes.

Our experiments show that overlooking the record and block relationship can cause severe performance problems when a record is comprised of several blocks scattered in different nodes. Our proposed novel data storage strategy, Record-Based Block Distribution (RBBD), optimizes the block distribution according to the record and block relationship. By being combined with our novel scheduling Weighted Set Cover Scheduling (WSCS), we efficiently reduces extra data transfers, and eventually improves the performance of the chunk-based storage system. Using our RBBD framework and WSCS in chunk-based storage system, our extensive experiments show that the data transfer decreases by 36.4% (average) and the scheduling algorithm outperforms the random algorithm by 51%-62%; the deviation from the ideal solutions is no more than 6.8%.

Index Terms—HPC Analytics Applications; Hadoop; Data-intensive; MapReduce; HDFS

I. INTRODUCTION

The further development of computation capability and large-scale storage systems facilitates research interweave among different scientific fields. Scientific applications running on High End Computing (HEC) platforms such as virtualization and simulation can generate large volumes of output to peta-scale [1]. Since I/O bandwidth is the bottleneck for evaluated applications [2], it imposes great challenges for the computing science researchers to resolve this issue.

Hadoop system [3], a data-intensive system based on HDFS, is designed to solve these challenges. HDFS is a chunk-based storage system, splitting data into blocks and storing them randomly across the data nodes. Hadoop achieves the co-located compute and storage by dividing the input data into splits according to the user parameters and distributing the tasks to the nodes storing most part of the splits. And then the split is divided into records which are the actual input for the Map phase. Although it works well for situations where a single record is within one block, it would not be sufficient for situations when a single record consists of several blocks which are distributed across the nodes as shown in Figure 1.

Assume the default block size of HDFS is M bytes, the file is divided into N blocks, of which N-1 blocks are M bytes and the block N is equal or less than M bytes. The input data as a record could be of several blocks especially for the binary files which are unsplittable as a whole file. For current storage mechanisms, the component blocks are randomly scattered among nodes without consideration of the record and block relationship. When a computation is requested by a record, blocks consisting of the record need to be transferred to the node to perform the computation.
To address this problem, we propose a Record-Based Block Distribution (RBBD) strategy and a Weighted Set Cover Scheduling (WSCS) strategy. For applications where the prior knowledge of the record and block relationship is retrieved, we design a Record-Based Block Distribution (RBBD) strategy, which stores a record with blocks to a single node according to the record and block relationship, not only dramatically decreases the possibility of network congestion by avoiding huge data transfer, but also reduces the whole analysis time.

On the other hand, for applications where the prior knowledge is unclear, the proposed WSCS utilizes the information of 1) available slots for scheduling, 2) locations of file chunks and 3) the network transfer latency of multiple chunks to estimate the cost of each candidate node for the map task and determines the best candidate for scheduling. Figure 2 shows the high-level system view with RBBD and WSCS.

Fig. 2. High-Level System View with RBBD and WSCS

The rest of this paper is organized as follows: Section II discusses the background and motivation. Section III details the design and implementation of RBBD. Section IV demonstrates WSCS strategy. Section V presents algorithms, results, and analysis respectively. Section VI states the related research in the data intensive analysis. Finally, Section VII concludes the paper and introduces our plan for the future work.

II. BACKGROUND AND MOTIVATION

A. Background

HDFS, a chunk-based storage system, is the primary distributed storage used by applications on Hadoop[1]. Files are divided into blocks of a predefined size, 64 MB by default, and blocks are stored as the basic storage unit. There are one Name Node and several Data Nodes in every Hadoop system. The Name Node manages relationship between files and blocks, also performing as Job Tracker to track task status on Task Trackers. While Data Nodes store blocks, also performing as Task Tracker[3] to execute tasks assigned by Job Tracker. The distributed storage enables computations on nodes independently and the replica attribution, 3 replicas by default, makes it reliable and easy for computation to be restarted in case of any failure.

MapReduce, the most popular framework adopted on Hadoop system, is consisted of two stages: Map and Reduce, which should be performed sequentially. Files as the input of the MapReduce are divided into splits according to the map task number specified by the user, then Task Trackers are scheduled by the Job Tracker to execute the map tasks. To reduce the data transfer, co-located computation is introduced to Hadoop system by launching the Map computation on nodes with most part of the input data. Figure 3 demonstrates the work flow of MapReduce over HDFS. Step (1) in Figure 3 distributes the Map tasks to Task Trackers according to the block locations, and in Step (2), Task Trackers read records from file splits, if the records are on the same node, then the data transfer can be avoided, otherwise, the remaining data should be read from other nodes.

B. Motivation

As illustrated above, blocks are not equal to records. For example, a file A is specified as the input of a MapReduce job:

1) The default block size is 64 MB and the replication is 3;
2) The File A, size 910 MB, is divided into 15 blocks, of which 14 blocks are 64 MB, the remain one is 14 MB;
3) One replica is used for the record distribution, other two replicas are stored randomly across the Data Nodes;
4) The record and block relationship is shown in Figure 4(a);
5) The random block distribution on the HDFS system is as shown in Figure 4(b);
6) The optimized distribution by RBBD on the HDFS system is as shown in Figure 4(c).

We learn that the transfer rate of blocks for original random solution is 10/15 = 66.7%; however, for our optimized solution RBBD the rate is 0/15 = 0%. So the performance enhancement for data transfer would be 66.7%. For scientists who are
encountering the bottleneck of network transfer, reducing the transferred data volume is essential to continue the research not only because it reduces the preparing time for data transfer and also avoids the possibility of network congestion caused by huge data transfer.

III. RECORD-BASED BLOCK DISTRIBUTION (RBBD)

A. Overview

There are two access patterns for binary files: with or without prior knowledge of the record and block relationship. In this section, we design RBBD which applies to the access pattern with prior knowledge such as files of Gadget [4] and HDF5 [5]. RBBD optimizes the block distribution according to the record and block relationship. The other pattern will be detailed in Section IV for binary files without prior knowledge, such as media files.

We use the uploading middleware (UPMIDD) to perform the block optimization at the time the data is uploaded to the storage system. As compared to the existing HDFS mechanism, our RBBD eliminates the gap between the records and blocks, and makes no data transfer at the time to perform the Map task. To make it simple, we use 1-replica to demonstrate how we improve the HDFS block distribution as shown in Figure 5.

Suppose there are N data nodes, the default block size is B bytes, the file to be uploaded is comprised of M blocks, the size of the Mth block is \( \leq B \) bytes, and we assume that every block belongs to only one record, no overlap among blocks of different records. Then we can analyze the possibility of optimized distribution as below:

1) We learned from the metadata that the number of blocks for the R records is an array as below:

\[
R[R] = R_0, R_1, R_2, \ldots, R_{R-1}
\]  

2) For the random placement, the probability of \( x \) blocks are placed on the same node is:

\[
p = \left( \frac{1}{N} \right)^x
\]  

3) Through random placement, the chance that the blocks are placed as optimized layout is:

\[
p_{opti} = \prod_{j=0}^{R-1} \left( \frac{1}{N} \right)^{R[j]} = \frac{1}{N} \sum_{j=0}^{R[j]} (1/N)^{R[j]}
\]  

4) Since we assume there is no overlap, then we can simplify the Equation 3 to:

\[
p_{opti} = \left( \frac{1}{N} \right)^M
\]  

Hence, Equation 4 defines the possibility of the random placement to achieve the optimal data placement according to the record, which is determined by the block number and the data node number. We can see the possibility is
Fig. 5. A detailed View of comparing HDFS default block distribution and our RBBD

decompressing along with the increasing number of data nodes in the Hadoop cluster; given a specific number of nodes, the increasing number of data blocks also leads to the decreasing possibility as well. Based on our analysis, let’s take our test bed as an example of a small-scale cluster, which contains over 40 blocks and 15 data nodes, the possibility using random data placement to achieve optimal data layout is $10^{-47}$. As a matter of fact, in large-scale clusters, the number of blocks and the number of data nodes are much larger than our example, which would dramatically decrease the possibility.

B. Uploading Middleware

Uploading Middleware (UPMIDD) takes charge of block optimization according to the metadata provided by the user. The Name Node maintains the record and block relationship in HDFS. What UPMIDD does is to replace the default random block distribution mechanism of HDFS with RBBD. The purpose of this middleware is to reduce the later data transfer caused by the random placement of blocks.

The work flow is shown in Figure 6.

When a client is uploading their data to the HDFS, the name node determines whether to use UPMIDD to do the record-based storage by the input parameter isRecordBased. The UPMIDD reads the user provided metadata with record delimitation information and then determines the record and block relationship together with the default block size and distributes the blocks of the same record on to the same node. isRecordBased is also used to determine whether this file needs to be distributed according to the metadata when the user requested to rebalance on HDFS. Assume the block size in the HDFS is B and the user provided metadata is listed as:

#recordId startoffset endoffset
1    N1   N2
2    N3   N4
3    N5   N6
...   ...   ...
M    N(2M-1)  N(2M)

Then UPMIDD would distribute the blocks as follows:

Algorithm 1 Pseudocode for UPMIDD using metadata

Input: Metadata, provided by the user, specifies the record delimitation and the block size in the HDFS configuration file.

Output: Block Number X and optimized distribution according to the record and block relationship.

Steps:
1) Get the block list of data: B1[0, B-1], B2[B, 2B-1], ..., BN[XB-1, N(2M)-1];
   - The block number X is:
   - if(N(2M)%B == 0), then X = N(2M)/B;
   - else X = N(2M)/B + 1;
2) Retrieve the block information and set the node IP address into the data structure RecordBlock.nodeIp;
3) Distribute the blocks to the node RecordBlock.nodeIp.
IV. WEIGHTED SET COVER SCHEDULING (WSCS)

This section illustrates the access pattern for binary files without prior knowledge of the record and block relationship by means of WSCS.

Suppose MapReduce tasks comprise of M map tasks and present as a set $S = \{s_1, s_2, ..., s_M\}$, such that $s_i$ stands for a data split consisting of multiple blocks, also multiple records, and will be processed by the ith map task. Once the Name Node receives these tasks, it returns all the nodes containing that data blocks (including the replica nodes) to the job scheduler. For a three replicas case, three nodes will be returned for a specific block. We call this set $K$, and $K_i$ will correspond to all the nodes that contain blocks from the input files.

We want to find a set $A (A \subseteq K)$ of nodes, where one of the nodes is the primary node to host the map task, and others are served as the secondary nodes and will provide the missing data to the primary node. The cost of data transfer from the secondary nodes to the primary node should be minimum. This nodes selection problem is similar to the weighted set cover problem, which models many resource selection problems. A split and a node in scheduling problem exactly corresponds to an element and a set respectively in the weighted set cover problem. The only difference is that “a split also represents a set of records, not a single element”. Each node contains at least one of the records. We need to find the set of nodes for each split. The weighted set cover problem has been proven to be NP-hard so that a heuristic and iterative algorithm is generally used to solve it.

The algorithm starts by retrieving all the nodes containing the blocks from a record and then starts the iteration with an empty set, $A_i$, which denotes a collection of the nodes selected until the last iteration. At each iteration, the algorithm selects a node $n_i$, adds it to $A_i$. Nodes that are added to the set $A_i$ are considered to be covering a part of split $s_i$. The algorithm finishes iteration when all the blocks in the $s_i$ are covered, and then $A_i$ gives a set of nodes, where the first one is used as a primary node and others as secondary nodes.

Algorithm 2 Pseudocode for scheduling using Weighted Set Cover Problem

Input: A set $U$ which consists of splits required by map tasks in a MapReduce Application, $S = \{s_1, s_2, ..., s_k\}$; A set C of nodes with the information that each node $n_i$ contains data blocks, $C = \{n_1, n_2, ..., n_N\}$.

Output: Find A for all splits in U such that $A_i \subseteq C$, and $A_i$ has nodes that completely cover all the blocks in split $s_i$.

Steps: $A = \{A_1, A_2, ..., A_M\}$, and all $A_i$ are empty, we will start computing each $A_i$ and add it to the set $A_i$. For $i$ is 1 to M, [Iterate through all the elements in set S] $A_j$ is empty.

1) Compute $K_i$. This will be a list of nodes with required data blocks. Data blocks will be located on multiple nodes.
2) Compute $w$ for all the nodes in $K_i$. $w$ is the weight assigned to each block as explain earlier. Also, mark the nodes which will be used for the data transfer.
3) Find the Block contribution of each node in $K_i$.
4) Compute the price as $\text{price}=\text{block contribution} \times w$ for each node in $K_i$.
5) Sort the values in ascending order, the first value will correspond to the primary node. Add the node to $A_i$.

This algorithm is used for all map tasks, and determines the optimal nodes for all map tasks belonging to one application.

V. EXPERIMENTAL RESULTS AND ANALYSIS

In our experiments, we demonstrate how RBBD framework performs for the two access patterns of binary files as described in Section III. The first access pattern of the binary files performs weighted set cover scheduling over the existing binary files with no prior metadata available on HDFS, whereas the second access pattern deals with the binary files with prior knowledge of the metadata about the record and block relationship. We also show the data transfer improvement due to block optimized distribution for the second access pattern. The most challenging part of this work is to fairly evaluate RBBD framework using various data access patterns against the same patterns used in the existing access pattern used in the existing HDFS storing implementations. However, it turned out unfortunately most binary analytics programs need to be developed. Also, there are no matured benchmarks for us to test our design. For the first access pattern, we simulate the scheduling algorithm with comprehensive experiments. We have used the Halo Finding application from the astrophysics domain, Gadget II [6] binary format to evaluate the RBBD framework. In the next subsection, we will describe the test bed and benchmark setup used to generate results.

A. Test bed

Our test bed is consisted of totally 15 heterogenous nodes with Hadoop 0.20.1 installed. All these nodes are in the same rack and the configurations for the Hadoop cluster is shown in Table I. One node is configured as the Name Node and Job Tracker, and other 14 nodes as Data Nodes and Task Trackers.

B. Simulation of WSCS

We simulate the scheduling algorithm with comprehensive experiments. There are two steps in our simulation: 1) set up environments with different configurations; 2) test the performance of random algorithm and our scheduling algorithm, and compare them with that of the ideal solution.

In the first step, we simulate a large scale cluster consisted of 100 racks, 5000 nodes in total. We also create a global routing table followed the instructions on ["TCP/IP network administration, p146-148"] and the metrics distribution observed in the real routing table for our campus network. In the widely used Routing Information Protocol (RIP) and other dynamic routing protocols (such as OSPF, OSI, and RIPv2), the metric in a routing table denotes the cost of the path through which the packet is to be sent. In order to focus on the efficiency of our scheduling algorithm, we don’t
consider the local transfer latency caused by disk I/O or cache miss; the only transfer latency for a chunk is the network latency denoted by the metric in the routing table. We assign number of different types of routine cost followed by uniform distribution. The cost starts from 0ms (in the same rack), increasing step is 10ms (based on the observations from part of the routing tables in our campus network). For simplicity, we don’t simulate the dynamic updates in the routing table. The number of replica for each chunk is set to be 3 as default in HDFS. Every rack contains the same number of nodes. All chunks are distributed onto the nodes based on the following rules: any two of the three replica should not be in the same node; each replica is randomly assigned to a rack.

In the second step, we implement random scheduling algorithm and our weighted set covering based scheduling algorithm. In both algorithms, we randomly pick one rack as the primary rack on which the read request is issued. Then we check the routing table to obtain the network information (the transfer latency starting from primary rack). For random algorithm, we randomly pick the racks from those who have the required chunks until the all the chunks are read. The algorithm is iterated 40 times and we get the average transfer latency. For weighted set covering algorithm, we use greedy algorithm to find the optimal or quasi-optimal solution which gives the combination of racks providing all the required chunks with smallest transfer latency. In the first 3 experiments, we use 4 types of network latencies (0ms to 30ms, 10ms for one step) but different number of required chunks. The purpose is to show how different the request sizes affect the performance of different algorithms; in the last 2 experiments, we use the same number of required chunks but vary the number of network latencies (8 types, 16 types) to explore the impact of different sizes of network. The results show that increasing the size of request (the number of requested chunks) does not significantly increase the transfer latency. The reason is: since the transfer latency between every two nodes in the same rack is negligible, and also the transfer latency from a node in one rack to any of the nodes in another rack is all the same, we can say that transfer latency between every two nodes is same as the transfer latency between the two racks. As a result, the data transfers in our experiment actually are incurred among racks. Since there are only 100 racks while much more requested blocks, every rack will maintain some of the requested blocks. In other words, the possible candidate set for solution always has all the racks (100 in our experiments). Our weighted set covering algorithm always choose the optimal or quasi-optimal combination of racks from the same candidate set, as a result, the overall performances are similar. The little difference is due to the different primary rack we randomly picked. The results in Figure 7 show that along with the increasing size of network, the average transfer latency will be obviously increased. This is caused by more higher network latencies we add (50ms, 60ms and so on). But no matter what kind of configuration we set, our WSCS outperforms the random algorithm 51%-62%; the deviation from the ideal solutions is no more than 6.8%.

In the second step, we implement random scheduling algorithm and our weighted set covering based scheduling algorithm. In both algorithms, we randomly pick one rack as the primary rack on which the read request is issued. Then we check the routing table to obtain the network information (the transfer latency starting from primary rack). For random algorithm, we randomly pick the racks from those who have the required chunks until the all the chunks are read. The algorithm is iterated 40 times and we get the average transfer latency. For weighted set covering algorithm, we use greedy algorithm to find the optimal or quasi-optimal solution which gives the combination of racks providing all the required chunks with smallest transfer latency. In the first 3 experiments, we use 4 types of network latencies (0ms to 30ms, 10ms for one step) but different number of required chunks. The purpose is to show how different the request sizes affect the performance of different algorithms; in the last 2 experiments, we use the same number of required chunks but vary the number of network latencies (8 types, 16 types) to explore the impact of different sizes of network. The results show that increasing the size of request (the number of requested chunks) does not significantly increase the transfer latency. The reason is: since the transfer latency between every two nodes in the same rack is negligible, and also the transfer latency from a node in one rack to any of the nodes in another rack is all the same, we can say that transfer latency between every two nodes is same as the transfer latency between the two racks. As a result, the data transfers in our experiment actually are incurred among racks. Since there are only 100 racks while much more requested blocks, every rack will maintain some of the requested blocks. In other words, the possible candidate set for solution always has all the racks (100 in our experiments). Our weighted set covering algorithm always choose the optimal or quasi-optimal combination of racks from the same candidate set, as a result, the overall performances are similar. The little difference is due to the different primary rack we randomly picked. The results in Figure 7 show that along with the increasing size of network, the average transfer latency will be obviously increased. This is caused by more higher network latencies we add (50ms, 60ms and so on). But no matter what kind of configuration we set, our WSCS outperforms the random algorithm 51%-62%; the deviation from the ideal solutions is no more than 6.8%.

C. Applications

We use a mass analyzer working with astrophysics data sets [6] for Halo Finding to do the experiments. There are positions and velocities of particles in these data sets. Each particle has the following attributes: (x, y, z) for the position coordinate and (v_x, v_y, v_z) for the speed velocity, and particle mass M, particle tag T. The mass analyzer reads in these data and calculates the average mass in each area with a predefined area size. Since the particle information as a whole input is much larger to fit in one analysis program, the particles are divided into several records and computed respectively. The records are of random size and stored across data nodes. We first execute the analysis using MapReduce with blocks randomly stored by default on HDFS, and then re-execute the same program again after the blocks are distributed across the nodes using RBBD and compare the two sets of results.

D. Results of RBBD

In this section, we present three sets of results for RBBD:

1) The block distribution: The random block distribution of the HDFS is adopted to balance the work load of data nodes. However, it also causes problems for cases we mentioned above where the record is comprise of several blocks. We did experiments for data with different record and block relationship to see how the random block distribution would turn out. We did 5 groups of experiments: one file with records of which the average number of blocks of a record is 2, 4, 8, 10 and 16. And we get the block distribution as Figure 8. Firstly, we get the average number of blocks to be transferred as shown in Figure 8(a) for different size of data. In order to show the actual data transfer rate for every node, Figure 8(b) shows the average number of blocks to be transferred on data node performing the actual map task for different size of data and average number of blocks of one record is 4. As we can see from Figure 8(a), for the data of the same size, if the average number of blocks of the record is equal or larger than the total number of data nodes, then there is a sharp reduce
for the average block number to be transferred because at that time, the possibility of the 16 blocks of the same record to be on the same node (across 14 data nodes) increases.

2) Overhead of RBBD: The RBBD is implemented as a Middleware in HDFS. The Hadoop administrator may manually enable/disable RBBD middleware in the conf file by setting isRBBDEnabled to true/false at any time and restart the Hadoop service. In this section, we quantify the overhead of RBBD, of which it includes two parts:

- Storage space for metadata;
- Retrieval of record and block relationship.

We made experiments to see the storage and calculation overhead of RBBD for files with 6 different groups of files as shown in Figure 9. The overhead includes the time to retrieve the record and block relationship from the metadata file and get the block list. The retrieval time is shown in Figure 9 for 6 groups of files. For situation with the average number of blocks of one record is 4, then the storage overhead per 64 MB is 5.44 bytes, the storage overhead rate is 0.0083% which is very little. And for the same situation, the average calculation overhead per 64 MB is only 0.43 ms which is worthy to the 62 ms for 64 MB to be transferred across 1 GB/s network.

3) Comparison of network transfer: By default, the HDFS stores the blocks randomly across the data nodes, but RBBD would optimize the blocks according to the record and block relationship to keep one replica of all blocks to the same record on a same node. We made experiments on data sets of size 28.8 GB, 57.6 GB, 115.2 GB, 172.8 GB, 230.4 GB, 288 GB and 576 GB, uploading them to the HDFS and get the block distribution diagram as shown in Figure 10.
influencing the performance. For example, if we have a data set of size 1TB, then by optimizing the block distribution, we can get an amount of 363.7GB reduce in data transfer. And for clusters which charge users for data transfer even inside the cloud, for example Azure, it means more than the reducing of time for transfer, but also the cost.

VI. RELATED WORK

There have been a lot of large-scale data processing frameworks in the scientific research area. MapReduce [7], Bigtable [8], Dryad [9] and many more storage systems are contributing to the storage and analysis of the large-scale data. Dryad is a research project at Microsoft Research for the general purpose runtime for execution of data parallel applications. Some approaches contribute to the scheduling improvement of MapReduce performance such as the performance improve research in Heterogeneous Environment [10] and Mantri [11]. Mantri proposes a solution to improve the MapReduce performance by improving the outliers, such as using network-aware placement of tasks, smart restart of outliers and protecting outputs of valuable tasks. In Mantri, they focus on the task monitoring and management.

Binary files are widely used in scientific applications especially as the simulation output. Structured format such as NetCDF [12], HDFS [5] and Gadget [4] applies to a lot of scientific data with binary format. There is a project, SciHadoop explores the porting of NetCDF on MapReduce framework [13]. However, The characteristic of the binary file processing is that the whole record should be input as a whole, otherwise the data does not mean anything. Our RBBD framework distribute the blocks according to the predefined metadata and processing them as a whole, which makes the analysis proceed correctly.

There are also researches on the access patterns research for MapReduce framework such as MRAP [14]. The author made a design change for the MapReduce by allowing the Map function to access two inputs and they also improved the data layout according to the data semantics to erase the gap between scientific data and HDFS storage format. However, our work is to erase the gap between the block and the actual map input as a record and to optimize the block distribution from the view of the record and it is designed to overcome the unsplittable feature of the binary file.

VII. CONCLUSION

We have developed an extended HDFS uploading framework RBBD to allow users to specify data semantics for Data-intensive analysis applications. Our approaches reduces the overhead of data transfer caused by the ignore of record and block relationship for the default random block placement. We provide functions and metadata templates to specify the record delimitations. We also studied the unsplittable feature of binary files, after specifying the delimitations through the metadata file, the program is easy to retrieve the correct input of the Map function. For experimentation, we ran a real application from astrophysics. Our results show an average data volume reduce rate of 36.4%.

These tasks which access records of multiple blocks also map to the data nodes as many as possible with the help of WSCS by selecting the optimal nodes for scheduling map tasks on the basis of block locations retrieved from the Name Node. In the future, we would implement the dynamic block balancing and scheduling schemes on a running Hadoop cluster. And we would like to take into account of the Intermediate data at Map phase to reduce data volume to be transferred and achieve performance improvement.

VIII. ACKNOWLEDGEMENTS

This work is supported in part by the US National Science Foundation Grant CCF-0811413, CNS-1115665, and National Science Foundation Early Career Award 0953946.

REFERENCES


