Optimizing Geo-Distributed Data Processing with Resource Heterogeneity over the Internet

SAEED MIRPOUR MARZUNI, Department of Electrical and Computer Engineering, University of Science and Technology of Mazandaran, Iran

ADEL N. TOOSI*, The University of Melbourne, Australia

ABDORREZA SAVADI†, Ferdowsi University of Mashhad, Iran

MAHMOUD NAGHIBZADEH, Ferdowsi University of Mashhad, Iran

DAVID TANIAR, Monash University, Australia

The traditional MapReduce frameworks were originally designed for processing data within a single cluster and are not suitable for handling geo-distributed data. Consequently, alternative approaches such as Hierarchical and Geo-Hadoop have been proposed to address this limitation. However, these approaches still face challenges in efficiently managing inter-cluster data transfer, particularly considering the heterogeneity of clusters and varying bandwidth among them. Moreover, the need to transmit results to a central global reducer for geo-distributed MapReduce operations adds unnecessary complexity. To tackle these issues, we introduce Extended Cross-MapReduce (ECMR), a framework that integrates resource heterogeneity and network links in geo-distributed MapReduce workflows. ECMR optimizes data management and determines the necessary data volume for generating final results. To enhance performance, ECMR leverages the overlap between data transfer and execution time by utilizing multiple global reducers and grouping temporary results that require data transfer over the Internet. In ECMR, we propose a bipartite graph and extend the Gale-Shapley algorithm to determine the optimal number of clusters and select the most suitable locations for global reducers. Through extensive experimental evaluations conducted on a real testbed, we demonstrate the effectiveness of our proposed ECMR method. The results exhibit significant improvements over traditional Hierarchical and Geo-Hadoop approaches, achieving reductions of up to 81% and 85% in overall makespan, respectively.

CCS Concepts: • **Computing methodologies** → **MapReduce algorithms**; • **Computer systems organization** → **Distributed architectures**; • **Information systems** → *Data management systems.*

Additional Key Words and Phrases: MapReduce, Geo-distributed Data, Data Centers, Big Data, Multi-cluster

ACM Reference Format:

Saeed Mirpour Marzuni, Adel N. Toosi, Abdorreza Savadi, Mahmoud Naghibzadeh, and David Taniar. 2023. Optimizing Geo-Distributed Data Processing with Resource Heterogeneity over the Internet. 1, 1 (November 2023), 28 pages. https://doi.org/XXXXXXX.XXXXXXX

*Corresponding author: adel.toosi@unimelb.edu.au

†Corresponding author: savadi@ferdowsi.um.ac.ir

Authors' addresses: Saeed Mirpour Marzuni, mirpour@mail.um.ac.ir, Department of Electrical and Computer Engineering, University of Science and Technology of Mazandaran, , Behshahr, Mazandaran, Iran, 48518-78195; Adel N. Toosi, The University of Melbourne, VIC 3052, Parkville, Australia, adel.toosi@unimelb.edu.au; Abdorreza Savadi, Ferdowsi University of Mashhad, Mashhad, Iran; Mahmoud Naghibzadeh, Ferdowsi University of Mashhad, , Mashhad, Khorasan Razavi, Iran; David Taniar, Monash University, VIC 3800, Clayton, , Australia.

1 INTRODUCTION

Every day, a massive amount of data is produced in the online world. It is believed that 90% of the world's data has been generated in the last two years, totaling more than 2.5 quintillion bytes per day. The need to analyze such data is increasingly essential across various fields of science and business. Biologists rely on high-throughput microscopes, while synchrotron beam lines employ high-speed cameras, resulting in large datasets that require thorough analysis for meaningful insights. Furthermore, modern Internet-based applications like the Internet of Things (IoT), smart cities, and social networks generate substantial amounts of big data, necessitating frequent processing and analysis. The input data for many applications is scattered across multiple locations worldwide, requiring the processing and analysis of vast amounts of geo-distributed data [7]. For example, Telegram servers are distributed globally, and Facebook operates a growing number of data centers spread across the world. Additionally, in many cases, the generation of geo-distributed data occurs at a significantly higher speed than the network data transfer rate [19, 23], e.g., modern satellites [10].

There are three primary reasons for the existence of geo-distributed data: (i) Many organizations operate in various countries, generating local data in different regions worldwide; (ii) Organizations may opt to utilize multiple clouds to improve their reliability, availability, and security [1, 14]. For example, in social networks, e-commerce, and content delivery networks, large volumes of data are constantly generated across geographically dispersed sites. Similarly, bioinformatic applications analyze genome datasets sourced from multiple laboratories, while monitoring systems examine log files from distributed servers. (iii) Data is frequently stored in close proximity to its source and needs to be processed in different locations. For instance, in wireless sensor networks (WSN), data is stored near the sensors themselves and must be processed together with data from other sensors.

In the modern era of computing, big data processing occurs across heterogeneous clusters within cloud data centers. These data centers are distributed globally and interconnected through the Internet, which often has a lower speed compared to the switching fabric of a single cluster. As a result, the processing of geo-distributed data becomes a significant and challenging task across various domains.

There are several frameworks available for big data processing within a single cluster, including Hadoop [1], Spark [26], Storm [2], Flink [3] and Heron [4]. Among these frameworks, the MapReduce programming model is widely adopted and recognized in the big data community. It is used in various applications such as inverted indexing in search engines like Google and Bing, as well as spam detection in Yahoo. However, popular frameworks that support MapReduce, such as Hadoop and Spark, are not specifically designed for processing geo-distributed data.

To utilize these frameworks for processing geo-distributed data, users are typically required to collect all the raw data in a central location. This approach is undesirable, especially when the output results of the computation at a single site are smaller than the input data [3, 11, 13]. In the literature, two state-of-the-art approaches have been proposed for processing geo-distributed data within the MapReduce model: the *Hierarchical* approach [4, 17] and the *Geo-Hadoop* approach [5, 24]. These approaches aim to address the challenges of processing data spread across different locations and provide solutions for efficient geo-distributed data processing. Even though the *Hierarchical* and *Geo-Hadoop* approaches are designed to execute MapReduce tasks on geo-distributed data, they still face challenges in transferring a significant amount of data between clusters over the network, particularly when clusters are connected through the Internet.

[1] http://hadoop.apache.org
[2] http://storm.apache.org
[3] https://flink.apache.org
[4] https://heron.apache.org

In the *Hierarchical* approach, each cluster initially processes its local data, generating temporary results (referred to as "temp results") at each cluster. These temp results are then transferred to a single cluster, known as the *global reducer*, where they are processed to produce the final results. However, the *Hierarchical* approach has a weakness: it only selects a single global reducer and transfers all the temp results to that particular cluster, even though not all the temp results are often required for generating the final results. This results in increased data transfer over the Internet and leads to reduced performance.

The *Geo-Hadoop* approach, on the other hand, resembles Hadoop MapReduce (Vanilla Hadoop) running on a single data center, allowing slave nodes from one cluster to communicate with the master node on another cluster. This enables the MapReduce process to proceed as usual, including the shuffle phase that can occur between nodes located in different clusters. However, *Geo-Hadoop* faces the significant challenge in the shuffle phase due to large inter-cluster data transfer over the Internet. For example, in an Adjacency-list application within the same cluster, the exported data from one cluster can be up to twice the size of the raw data [18]. Similarly, the *Invertedindex* application, with an input data size of 1.4 GB, generates 4.5 GB of intermediate data, as in [11]. Due to slower data transfer between clusters over the Internet compared to within a cluster, it leads to prolonged processing times, particularly for applications with intermediate results larger than the final results.

In this paper, we introduce Extended Cross-MapReduce (ECMR), a solution for processing geo-distributed data across multiple heterogeneous cloud clusters. Our approach focuses on effectively managing the data, providing a platform-independent framework that enhances the performance of geo-distributed MapReduce. By selecting multiple global reducers and transferring only necessary temporary results, we minimize unnecessary data transfer and optimize the overall processing time. Additionally, our Data Balancing algorithm ensures a balanced distribution of data volume, leading to equalized makespan across clusters. The estimation algorithm for concurrent data transfer time further enhances the efficiency of data processing and overlapping the data transfer time with processing time. Our **key contributions** are as follows:

- **Data Balancing**: We propose Data Balancing algorithm to determine how much data should be transferred and to which cluster in a way that local MapReduce jobs are finished simultaneously. This approach optimizes performance by transferring raw data from clusters that finish their local MapReduce job processing later to clusters that finish it earlier.
- **Data Transfer Time Estimation**: We present algorithms to estimate the data transfer time between clusters, taking into account the effect of concurrent data transfers on bandwidth used by Data Balancing algorithm.
- **Global Reducer Selection**: We propose the innovative concept of multiple global reducers, eliminating the requirement to transfer the entire temp results to a single global reducer. ECMR intelligently manages the data by selectively shuffling the necessary portion of results, which are crucial for generating the final results. Furthermore, we address the challenge of global reducer selection by modeling it as a Complete Bipartite Graph and extending the Gale-Shapley algorithm to find the best global reducers.
- **Coordinator Selection**: In order to efficiently group keys and execute the global reducer selection algorithm, the presence of a capable coordinator is essential. To address this need, we propose a dedicated algorithm designed to identify the optimal coordinator for performing these critical tasks with utmost efficiency and effectiveness.
- **Real Testbed Evaluation:** We evaluate our proposed framework and algorithms on real-world YARN clusters, comparing them against state-of-the-art solutions for geo-distributed MapReduce.

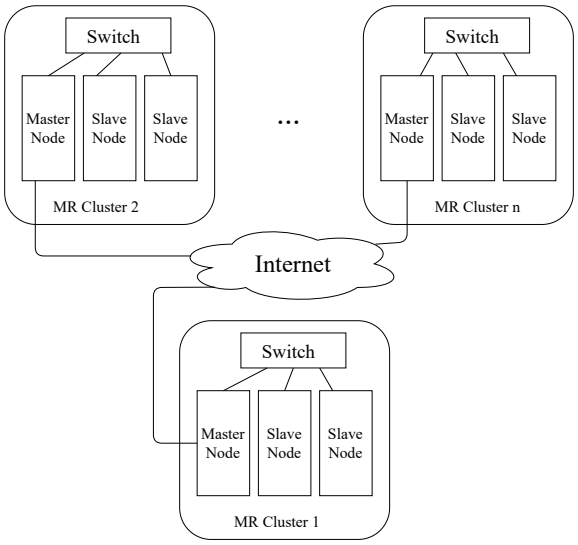


Fig. 1. System Overview

The rest of the paper is organized as follows: Section 2 describes the problem addressed in this research. The ECMR framework is proposed in Section 3. Section 4 presents the experimental results. In Section 5, we navigate through the diverse challenges present in this domain with precision and clarity. Section 6 discusses existing methods and related work, and the final section concludes the work and suggests directions for future research.

2 PROBLEM STATEMENT

The system model consists of a set of MapReduce clusters, each comprising several nodes. These clusters are interconnected through the Internet. Figure 1 illustrates the high-level system architecture.

Suppose a framework supporting the MapReduce model, such as Hadoop or Spark, is set up on all clusters. We make the following three general assumptions:

- Clusters are heterogeneous, meaning that each cluster can have processors and computing devices with different computational and networking capabilities compared to other clusters. Consequently, the processing time for a specific job may vary across clusters, even when the conditions such as data volume and application type are the same.
- Inter-cluster bandwidths are heterogeneous. Each cluster is assumed to have both upstream and downstream links for data upload and download, respectively, which can differ and vary across clusters. In other words, when data is sent from one cluster to another, the data transfer time can vary based on the destination. For example, if Cluster A has an upstream link of 1000 Mbps and Cluster B has a downstream link of 500 Mbps, the available bandwidth from Cluster A to Cluster B, assuming no other network traffic, would be limited to 500 Mbps.
- We assume that the data is unevenly distributed among the clusters before the job execution begins.

The computational power of some clusters is higher than that of others, and the bandwidth between clusters is limited over the Internet, which can become a bottleneck for the system. The main research problem addressed in this

paper is "how can we minimize the MapReduce processing time for big data scattered over multiple geographically distributed and heterogeneous clusters?"

Since clusters differ in terms of their computational power and the size of their data volumes, when a job is executed on local data, there may be clusters that complete their tasks earlier than others. This creates a waiting time for the remaining clusters to finish their jobs before shuffling the results. This waiting time leads to reduced performance and inefficient resource utilization. To address this problem, we propose a data balancing algorithm.

Data balance refers to a state in which all clusters complete their local processing simultaneously. To achieve data balance, the following questions arise: "Which clusters should transfer data to which clusters, and how much data should be transferred?" To answer these questions, it is necessary to consider the inter-cluster bandwidth and computational power of the clusters.

We differentiate between *inter-cluster* and *intra-cluster* data transfer. Intra-cluster data transfer can be freely performed within a cluster, while inter-cluster data transfer occurs over limited bandwidth between clusters and is considered costly and a primary bottleneck of the system. Therefore, we aim to minimize inter-cluster data transfer as much as possible. We introduce "Gshuffling" as a solution, which defers inter-cluster data transfer until local MapReduce jobs are completed in each cluster. Instead of shuffling data over the Internet, the transfer between clusters takes place after all reduce tasks are finished within each cluster. By implementing Gshuffling, we expect a significant reduction in the volume of inter-cluster transfers, as the number of records transmitted between clusters is decreased.

As stated earlier, instead of sending the entire temporary results to a single cluster, we only send keys to a selected cluster referred to as the "coordinator". Thus, one of the challenges lies in determining which cluster should serve as the coordinator. Various factors can be considered in making this selection, such as the cluster with the highest volume of keys, the highest computational power, or the highest downstream bandwidth. With multiple potential candidates for the coordinator role, it becomes crucial to devise an algorithm that can identify the most suitable cluster to enhance performance. To address this, we introduce the "coordinator selection" algorithm, which aims to identify the optimal cluster to serve as the coordinator.

The coordinator plays a crucial role in our problem by selecting specific global reducers from the clusters and determining the direction of necessary data transfer. In the subsequent phase, the coordinator is responsible for selecting multiple global reducers. This selection process raises several important questions: 1) Which clusters should be chosen as global reducers? 2) What portion of the results needs to be processed by the global reducers? 3) Where should the selected portion data be transferred for processing by the global reducers? Addressing these inquiries necessitates considering factors such as inter-cluster bandwidth and the computational power of the clusters. Consequently, the "global reducer selection" algorithm is devised in the final phase to provide answers to these questions.

2.1 Motivational examples

In Figure 2, we explain a very simple example to demonstrate how the use of data management can increase efficiency with ECMR. Consider three clusters, each of which has completed its job in the first phase. Suppose the processing power of *Clusters 1* and *3* is the same, and the processing power of *Cluster 2* is less than theirs. For the sake of simplicity, we assume that the data transfer rate between all three clusters is identical. If we use a hierarchical method, we should choose one of the *Clusters 3* or *1* as a global reducer. Suppose *Cluster 3* is selected. Then all the key-value pairs available in *Clusters 1* and *2* must be transferred to *Cluster 3*, which includes a total of 20 records.

However, in ECMR, efficient data management is utilized to transfer only the required key-value pairs to global reducers. This is achieved by grouping keys based on the number of clusters and creating groups that represent common

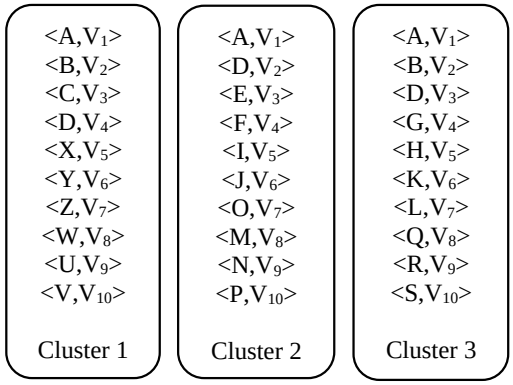


Fig. 2. An example of temp results in each cluster

keys among them. The group names are determined by the cluster numbers involved. For instance, if we have two clusters with numbers 1 and 3, we create a group titled 1-3, containing keys common to both clusters. This results in four groups for three clusters: 1-2, 1-3, 2-3, and 1-2-3.

To illustrate, suppose *Key A* is common in all three clusters. In this case, it is placed in *Group 1-2-3*. *Key B* is common in *Clusters 1* and *3*, so it is put in *Group 1-3*, and the remaining keys are grouped in the same way. This process results in the following: (1-2-3: A, D), (1-2: NULL), (1-3: B), and (2-3: NULL).

Once the groups are created, ECMR sends them to the related clusters, and the corresponding key-value pairs are determined. By utilizing this method, only three records (A, B, and D) need to be transferred. The global reducers are selected based on the size of the key-value pairs of the grouped data and the bandwidth and processing power of the clusters using Algorithm 4.

Considering the transfer speed between clusters is identical, and the three records that need to be transferred, *Clusters 1* and *3* are chosen as the global reducers. This allows the two clusters to process the records simultaneously, resulting in a shorter overall makespan. The key-value pairs of *Group 1-2-3* (A, D) are transferred from *Clusters 1* and *2* to *Cluster 3*, while the key-value pair of 1-3 (B) is transferred from *Cluster 3* to *Cluster 1*, generating the final result.

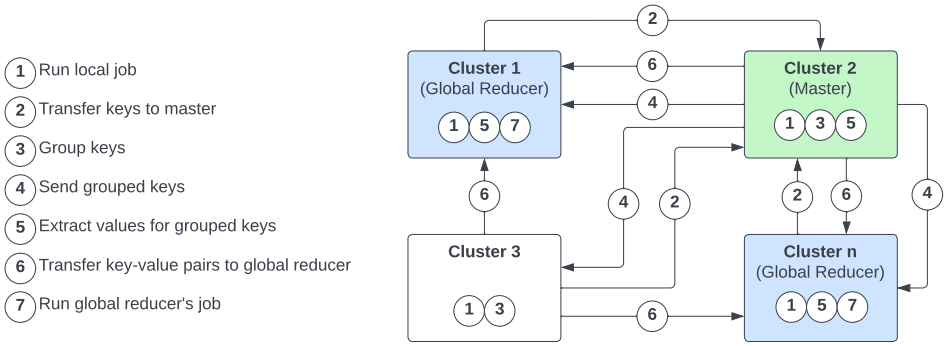


Fig. 3. Extended Cross-MapReduce steps

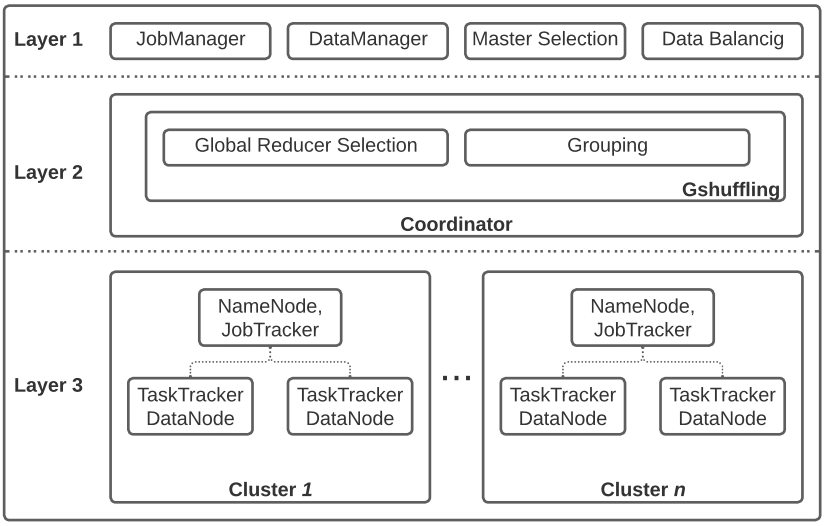


Fig. 4. System Architecture

3 EXTENDED CROSS-MAPREDUCE (ECMR)

Generally, we transmit data in three stages. A key aspect of this process is that at each stage, the amount of data being transmitted is considerably smaller than the total data volume. Although the number of transfers has increased, the overall volume of data transferred has decreased. Notably, only during the first stage (raw data transmission) might the data volume be large, which also enhances efficiency by overlapping with data processing time. Figure 3 illustrates the sequential steps involved in different phases of ECMR, as well as the communication process between clusters. Initially, ECMR operates similarly to the *Hierarchical* approach, where each cluster independently executes its jobs on local data, generating individual results (Step 1). Instead of transmitting the complete temporary results to a single cluster, only the keys are sent to a designated *coordinator* cluster (Step 2). In the *coordinator*, all keys are grouped together (Step 3) and subsequently dispatched to their respective clusters (Step 4). Each cluster then extracts the key-values pertaining to their assigned groups (Step 5). The extracted groups of key-values are subsequently transferred to the designated global reducers (Step 6), ultimately resulting in the production of final results (Step 7).

3.1 System design

Firstly, a user submits the job to the JobManager, which then divides the job into sub-jobs and dispatches each one to the clusters in the third layer. Within the third layer, data is processed locally and independently within each cluster. Before initiating the job in each cluster, the Data Balancing component ensures an equitable distribution of data among the clusters to ensure that all clusters complete their local tasks simultaneously.

The completion of each sub-job is reported to the JobManager, along with the corresponding address of the results stored in the DataManager. Subsequently, the DataManager leverages the “Coordinator Selection" algorithm to choose a cluster known as the *coordinator*. Within Layer 2, the coordinator collects keys from all clusters. In the coordinator cluster, the keys are grouped to identify common keys between clusters, thereby facilitating the Gshuffling process. The

Table 1. symbols frequently used in the paper

Symbol	Definition
n	Number of clusters
$\mathbf{T}$	The vector of makespan of a running job locally
T_i	Makespan of a running job locally at cluster i
$\mathbf{T_w}$	The waiting time vector for clusters to receive data share
T_{w_i}	Time that *Cluster i* waits to receive its data share
T_{p_i}	Processing time of local MapReduce job
$\mathbf{\Delta}$	The matrix of transferred data volume among clusters
δ_{ij}	Signed transferred data volume from *Cluster i* to *Cluster j*
d_{ij}	Transferred data volume from *Cluster i* to *Cluster j* $\lvert\delta_{ij}\rvert$
D_i	Initial data volume size at *Cluster i*
α_i	The computing power of *Cluster i*
β_i	Preprocessing time of local job on *Cluster i*
$\mathbf{BW}$	Matrix of available bandwidth among clusters
bw_{ij}	The available bandwidth between *Cluster i* and *j*
NS_i	Number of clusters to which data should be sent
NR_i	Number of clusters from which data should be received
S_i	Upstream bandwidth of *Cluster i*
R_i	Downstream bandwidth of *Cluster i*
$w_{i,j}$	Cost of processing *Group j* on *Cluster i*

"Global Reducer Selection" algorithm, which is elaborated upon in subsequent sections, determines the necessary global reducers and their optimal placement within Layer 2. The required portion of the results is then transmitted to the selected global reducers.

Finally, the process concludes with the DataManager adding the addresses of the global reducer results. Similar to G-Hadoop, ECMR also distributes the final results across multiple clusters. This approach finds wide applicability in various applications such as GeoSearcher [25].

The details of each phase are described in the subsequent subsections, and Table 1 presents a comprehensive list of the symbols utilized. Matrices and vectors are denoted by bold face upper case letters.

3.2 Phase 1: Data Balance and Running Job Locally

In big data processing, one of the challenges is deciding whether to move computations to the node where the data is located or to transfer the data to the node where the computations are taking place. In Phase 1, each cluster independently runs the job. Due to heterogeneity among clusters and variations in data volume, it is highly probable that some clusters may finish their jobs earlier than others, a scenario we refer to as 'data imbalance'. Consequently, the coordinator must wait for all clusters to complete their jobs in Phase 2. To enhance performance, transferring raw data from clusters that finish later to clusters that finish earlier is a favorable option for achieving better data balance.

To illustrate the application of the data balance, let's consider two clusters as depicted in Figure 5. *Cluster 2* possesses twice the processing power of *Cluster 1* and a data transfer bandwidth of 10 MBps is available between them. Suppose *Cluster 1* processes 10GB of data in 60 minutes, while *Cluster 2* completes its job with 2GB in 6 minutes. Without using the data balancing algorithm, the system would need to remain in Phase 1 for the entire 60 minutes to complete the processing. However, by transferring a portion of the data from *Cluster 1* to *2* before initiating processing in *Cluster 2*, the processing time for Phase 1 can be reduced.

We take into account the data transfer bandwidth and the processing power of the clusters. It transfers data from the cluster with a longer processing time to the cluster with a shorter processing time, enabling all clusters to complete their work nearly simultaneously in Phase 1. In the example, assuming a linear correlation between processing time and data size, if we transfer 5 GB of data from *Cluster 1* to *Cluster 2* in 128MB blocks, the processing time in *Cluster 1* is reduced by approximately 30 minutes. The transfer time for 5 GB of data to *Cluster 2* is 8.5 minutes. The processing time for the remaining 7 GB of data in *Cluster 2* is 21 minutes. Considering the addition of transfer time, the total processing time becomes 29.5 minutes. Therefore, by utilizing the data balancing algorithm, the processing time in the first phase can be significantly decreased.

Although precise synchronization of processing time across all clusters is not feasible in the first phase due to block partitioning in MapReduce (where the minimum amount of data that can be transferred is one block), efforts are made to ensure that all clusters finish their jobs nearly simultaneously.

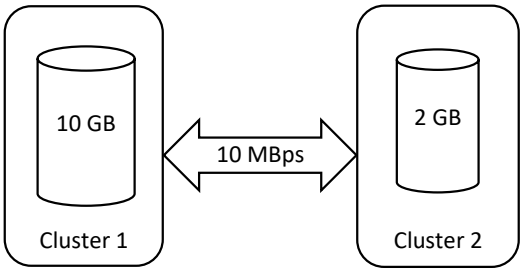


Fig. 5. Example of data balancing algorithm

For this purpose, we present two approaches: a “linear programming optimization" model and a heuristic known as the “data balancing" algorithm.

3.2.1 Linear programming optimization model. The linear programming model is presented below:

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} |T_i - T_j| \\ \text{subject to} \quad & T_i = T_{w_i} + T_{p_i}, \\ & 0 < d_{ij} \leq D_i, \\ & 0 < d_{ji} \leq D_j, \\ & 0 < i, j \leq n. \end{aligned} \tag{1}$$

The objective function aims to minimize the disparities in the makespans of running the job locally across all clusters. Here, n denotes the number of clusters. T_i represents the makespan of a job running locally on *Cluster i*. T_i is the sum of T_{w_i}, which is the time *Cluster i* waits for the transfer of its data share (data transfer time), and T_{p_i}, representing the processing time of the local MapReduce job. T_{w_i} is a function of d_{ji}, which indicates the volume of data intended to be transferred from *Cluster j* to *Cluster i*.

Table 2. Estimated data transfer time versus actual data transfer time

Transferred data	BW_{ij}	Estimated time	Actual time
Monash to London	43.2Mbits/sec	3:16 min	3:29 min
Monash to Ohio	57.2Mbits/sec	2:38 min	2:32 min

Before delving into the intricacies of the linear programming optimization model, it is crucial to comprehend that the data transfer time between two specific clusters can be intuitively calculated as follows:

$$\frac{d_{ij}}{\min(S_i, R_j)}, \tag{2}$$

where d_{ij} represents the intended data volume to be transferred from *Cluster i* to *Cluster j*, while S_i denotes the upstream bandwidth of *Cluster i*, and R_j represents the downstream bandwidth of *Cluster j*. For the sake of brevity, from this point onwards, we will use the notation bw_{ij} to represent $\min(S_i, R_j)$. To validate this model, we conducted several experiments using multiple virtual machines (VMs) running in various data centers around the world. We transfer 1GB of data between these VMs and compared the estimated transfer time with the actual transfer time. The actual transfer time was measured to reflect the true duration of the data transfer. Table 2 presents the results for two sample pairs. In these experiments, the available bandwidth was estimated using the "ipref3" tool, and data was transferred using the Linux "scp" command. As shown in Table 2, the estimated time closely matches the actual data transfer time. Note that the difference is even smaller for a larger size files (> 1GB).

Now, suppose D_i represents the initial data volume size at *Cluster i*. In order to achieve a better data balance, *Cluster i* can receive data from multiple clusters. Therefore, the data transfer time T_{w_i} is computed as the sum of all data transfers to *Cluster i*. The data transfer time from *Cluster j* to *Cluster i* is calculated as the ratio of the data transferred to the bandwidth between *Cluster j* and *Cluster i*. It is important to note that d_{ij} must be lower than D_i for valid transfers to occur. As a result, T_{w_i} is calculated as follows:

$$T_{w_i} = \sum_{j=1, j\neq i}^{n} \frac{d_{ji}}{bw_{ji}}. \tag{3}$$

For the purpose of estimating processing time (T_{p_i}), we utilize the model presented in [22]. In each job, the processing time is determined based on two variables: 1) data volume and 2) the number of resources. In this paper, the cluster sizes remain constant, and there are no variations in terms of resources, while the data volume is variable. Thus, we assume that the processing time of a local job on a single cluster is linearly correlated with the data volume. This correlation is empirically demonstrated in Section 4.1 and is presented as the following lemma.

Lemma 1: The processing time of a job on a specific cluster is linearly correlated with the data volume size at the cluster.

Lemma 1's proof is available in the supplementary material.

Therefore, we calculate T_{p_i} as follows:

$$T_{p_i} = \alpha_i \times D_i + \beta_i, \tag{4}$$

where α_i is the regression slope that shows the computing power of *Cluster i* and β_i is the preprocessing time of local job on *Cluster i*.

Although linear programming optimization is designed to provide an optimal solution, it has two fundamental weaknesses. Firstly, as the number of clusters increases, its complexity grows exponentially due to the pairwise

comparison of makespans. Secondly, to convert this problem into a linear optimization, the data transfer time is modeled to be sequential (serial), meaning that data transfer time is computed under the assumption that each cluster sends its data to a specific destination one after the other.

For instance, if data transfer is required from *Cluster j* and *k* to *Cluster i*, the transfer time is calculated as *Cluster j* transferring its data first, utilizing the entire bandwidth, and then *Cluster k* transferring its data. In practice, data transfer can occur concurrently, where the bandwidth is shared. However, the precise calculation of such concurrent data transfer is not a trivial task and involves non-linearities. Hence, we present the data balancing algorithm, which assumes concurrent data transfer, to address this limitation.

To demonstrate the efficiency of the data balancing algorithm proposed in the next section, we compare it against the optimal method as proposed in Section 4.

3.2.2 Data balancing algorithm. The data balancing algorithm begins by estimating the makespan of local execution of job on each cluster in Phase 1. It then calculates the average makespan of running the job locally across all clusters. The algorithm proceeds to determine the difference between the makespan of each cluster and the average makespan. Based on this difference, data is transferred from clusters with higher makespans to clusters with lower makespans. This process is repeated iteratively until the desired makespan difference is achieved. Essentially, the algorithm calculates the amount of data that needs to be transferred among the clusters to ensure an equivalent makespan across all clusters.

As mentioned, the algorithm aims to converge the makespan of all clusters to the average point. To achieve this, we define an $n \times n$ matrix, denoted as $\mathbf{\Delta}$, where n represents the number of clusters. Within this matrix, each element δ_{ij} denotes a real number, either positive or negative. All initial values of the matrix are set to zero. Suppose that $\delta_{ij} = d$, if d is a positive number, d units of data should be transferred from *Cluster i* to *Cluster j*, and if d is negative, d units of data should be transferred from *Cluster j* to *Cluster i*. Algorithm 1 outlines the steps of the data balancing algorithm.

In Line 1, the makespan of each cluster is computed as the sum of the processing time (Equation 4) and the estimated data transfer time (discussed in Subsection 3.2.3). In Line 2, a condition is checked to determine whether the difference between the maximum and minimum makespan of clusters is lower than the specified threshold. In Line 3, the balancing function computes how much data should be transferred among clusters. In Line 4, the algorithm checks whether the $\mathbf{\Delta}$ is normal. Here, a normal matrix is defined as a matrix in which each row includes only positive or negative numbers. If there exist both positive and negative numbers in a row, data transfer between clusters can be reduced.

Let's explain this with an example: Suppose that $\delta_{2,1} = 6$ and $\delta_{2,3} = -10$, indicating that *Cluster 1* should transfer 6 units of data to *Cluster 2*, and *Cluster 2* should transfer 10 units of data to *Cluster 3* ($1 \xrightarrow{6} 2 \xrightarrow{-10} 3$). In this case, we can reduce the total amount of data transfer. *Cluster 2* can transfer only 4 units of data to *Cluster 3*, and *Cluster 1* should transfer 6 units of data to *Cluster 3* ($1 \xrightarrow{6} 3, 2 \xrightarrow{4} 3$). We refer to this data transfer as normalized transfer, and we update the matrix accordingly for any possible cases like this (line 5). In Line 6, the size of data that exists in each cluster and the data transfer time between clusters should be updated based on the normalized $\mathbf{\Delta}$ matrix.

The balancing function (Lines 8-23) divides the clusters into two groups: 1) clusters whose makespan is less than the average time, and 2) clusters whose makespan is more than the average time (Lines 13 and 17). The function calculates the amount of data that needs to be transferred from clusters in the latter group to clusters in the former group. This is determined using the following equation:

$$T' = T_p + T_w. \tag{5}$$

Algorithm 1 Data Balancing

```
1: T = GetMakespan( );
2: while Max(T)-Min(T)> threshold do
3:     Balancing( );
4:     while IsNormal(Δ) == False do
5:         Normalize(Δ);
6:     UpdateDataAndTransferTime( );
7:     T = GetMakeSpan( );
8: function Balancing( )
9:     makespanAvg = GetAverage(T);
10:    k = 0, j = 0
11:    for i = 0; i < n; i++ do
12:        if T_i > makespanAvg then
13:            CLA_j = i
14:            j = j + 1
15:        else
16:            if T_i < makespanAvg then
17:                CMA_k = i;
18:                k = k + 1
19:    for i = 0; i < CMA.Size(); i++ do
20:        for j = 0; j < CLA.Size(); j++ do
21:            Temp = DataForBalancing( );
22:        data = GetMin(Temp);
23:        Δ is updated by data;
```

$T^{'}$ represents the updated makespan after the data transfer. The data transfer time T_w is obtained by (6):

$$T_w = \frac{d}{BW}, \tag{6}$$

where d represents the portion of data that should be transferred, and BW denotes the bandwidth between two specific clusters. Now, we rewrite Equation (5) by incorporating (4) and (6):

$$T_w = \alpha \times (D + d) + \beta + \frac{d}{BW}, \tag{7}$$

where D represents the initial existing data volume in the cluster. Therefore, the portion of data that should be transferred is determined as follows:

$$\begin{gathered} \Rightarrow T_w - \alpha \times D - \beta = \alpha \times d + \frac{d}{BW} \\ \Rightarrow d = \frac{T_w - \alpha \times D - \beta}{\alpha + \frac{1}{BW}} \Rightarrow d = \frac{T_w - T_p}{\alpha + \frac{1}{BW}} \end{gathered} \tag{8}$$

In Line 9, the Balancing() function obtains the average makespan of all clusters. From Lines 19 to 23, for each cluster that has a makespan less than the average, it identifies the portion of data it can receive from clusters with a makespan greater than the average. There may be multiple candidate clusters for data transfer, and the cluster with the minimum data transfer requirement is selected. The determined portion of data to be transferred is then stored in the **Δ** matrix. The DataForBalancing() function is used to calculate the portion of data to be transferred using Equation (8).

The time complexity of the Balancing function is $O(n^2)$, where n is the number of clusters ($CLA.Size \leq n$). The Balancing function is called at most n times by the algorithm in the worst case. Therefore, the overall time complexity of the algorithm is $O(n^3)$.

3.2.3 Data transfer time estimation in concurrent mode. In our problem, one cluster can receive data from other clusters concurrently. When data is being sent and received between multiple clusters concurrently, the estimation of data transfer time becomes more complex due to bandwidth sharing among multiple transfers. Therefore, we propose Algorithm 2 to estimate the concurrent data transfer time between clusters using the model in Equation 2. The underlying idea behind this algorithm is that when there are multiple active data transfers over an immediate upstream/downstream link, the available bandwidth is evenly shared among those transfers, with the share limited to the available bandwidth of the sources/destinations, respectively.

Algorithm 2 Data Transfer Time Estimation

Input: $\mathbf{\Delta}$, $\mathbf{S}$, $\mathbf{R}$
Output: $\mathbf{T_w}$

1: **BW** = SetBw($\mathbf{\Delta}, \mathbf{S}, \mathbf{R}$);
2: **DTTime** = GetDTTime(Δ, **BW**);
3: **flag** = $\{True\}$;
4: **while** True **do**
5: $minTime$ = GetMin(**DTTime**);
6: **if** $\mathbf{\Delta}$ is null (zero) matrix **then**
7: $break$;
8: **for** $i = 0; i < n; i++$ **do**
9: **for** $j = 0; j < n; j++$ **do**
10: **if** δ_{ij}==0 **then**
11: $continue$;
12: $tData = minTime \times bw_{ij}$;
13: **if** $flag_j$==True **then**
14: $T_{w_j} = T_{w_j} + minTime$;
15: $flag_j = False$;
16: $\delta_{ij} = \delta_{ij} - tData$;
17: **BW** = SetBw($\mathbf{\Delta}, \mathbf{S}, \mathbf{R}$);
18: **DTTime** = GetDTTime(Δ, **BW**);
19: $flag$ = Set($True$)
20: Return($\mathbf{T_w}$);

We define four arrays: **S**, **R**, **NS**, and **NR**, where S_i, R_i, NS_i, and NR_i represent cluster i's upstream bandwidth, downstream bandwidth, the number of clusters to which data should be sent, and the number of clusters from which data should be received, respectively. The algorithm outputs $\mathbf{T_w}$, which represents a vector of the total time that each cluster should wait for its data. In the first line, **NS** and **NR** are calculated using the SetBw() function, and then the available bandwidth between clusters involved in data transfer is determined as $\min(\frac{S_i}{NS_i}, \frac{R_j}{NR_j})$. NS_i is calculated as the number of non-zero elements in Row i of the Matrix $\mathbf{\Delta}$ using the Function SetBw(), and NR_j is calculated as the number of non-zero elements in Column j of $\mathbf{\Delta}$ using the function. The data transfer time between each pair of clusters is calculated in Line 2 based on the data transfer volume and the available bandwidth.

A flag is used for each cluster in Line 3, with its initial value set to *True*. This flag is used to avoid repeated addition of concurrent data transfer time to $\mathbf{T_w}$. In Line 5, the minimum data transfer time (*minTime*) is obtained. It represents the minimum time during which no other new events (start or end of a data transfer) occur that would affect the current data transfer. Next, in Line 6, the algorithm checks if all elements of Matrix d are zero, and if so, it terminates the process. In Lines 8 to 16, the algorithm computes the total time that each cluster should wait to receive its data. During the period of *minTime*, the data volume that can be transferred between two specific clusters is calculated in Line 12. Before adding *minTime* to $\mathbf{T_w}$ in Line 14, the algorithm checks the flag of each cluster to ensure that *minTime* is only added if it has not been added in the current period (Lines 13 to 15).

At the end of the current period, the data transfer volume is subtracted from δ_{ij}, updating the data Matrix $\mathbf{\Delta}$ in Line 16. In Lines 17 and 18, **BW** and **DDTime** are updated accordingly for use in the next period. Finally, all flags are set to *True* again in Line 19, and the algorithm loops back to the beginning to find the next *minTime* (period) until the condition in Line 7 is met.

The time complexity of the algorithm is $O(n^2)$, where n is the number of clusters, as it iterates over every cluster and checks the data transfer to all others.

3.3 Phase 2: Coordinator Selection and Grouping

ECMR selects a cluster as a *coordinator* in Phase 2 after all clusters have completed their local data processing. Subsequently, each cluster sends the keys of its temporary results to the *coordinator*. However, the question remains: which cluster is the best choice for serving as the *coordinator*? To answer this question, ECMR utilizes the Coordinator Selection algorithm (Algorithm 3), which selects the best cluster as a coordinator based on the available inter-cluster bandwidth and the size of the key set in each cluster.

Algorithm 3 Coordinator Selection

Inputs: keysVolume ▷ The array of volume of keys in each cluster
Output: Coordinator

1: $TTime = 0$; ▷ Transfer time
2: $Vol = \textsc{TotalVolume}(keysVolume)$; ▷ Summing the elements in **keysVolume** and assigning it to 'Vol' as the total volume of keys
3: $coordinator = 0$; $minTime = +\infty$;
4: **for** $i = 0; i < n; i++$ **do**
5: **for** $j = 0; j < n; j++$ **do**
6: $TTime = TTime + \frac{keysVolume_j}{bw_{ji}}$
7: $T_i = TTime + \textsc{ProcessingTime}(Vol)$
8: **if** $T_i < minTime$ **then**
9: $minTime = T_i$;
10: $coordinator = i$;
11: $\textsc{Return}(coordinator)$;

In the second line of Algorithm 3, the sum of the all of the keys volume is computed. From Lines 4 to 6, the algorithm calculates the data transfer time for the key set to each cluster. Subsequently, the makespan is computed for each cluster in Line 7. The ProcessingTime() function obtains the processing time of the job based on data volume using the Equation 4. In Line 8 to 10, the cluster with the minimum makespan is selected as the *coordinator*. The selected

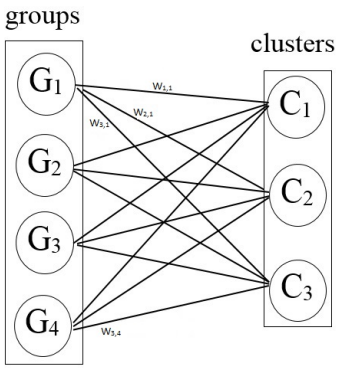


Fig. 6. A complete bipartite graph

coordinator is then announced to all clusters. Subsequently, all other clusters transfer the keys associated with their temporary results to the *coordinator*.

After receiving all the keys, the *coordinator* organizes them into groups. Each key is assigned to a single group, which contains a set of common keys among the largest set of clusters. For example, if there are three clusters in the system, four different groups are created. The first group contains keys that are common between *Clusters 1* and *2*, the second group contains keys that are common between *Clusters 1* and *3*, the third group contains keys that are common between *Clusters 2* and *3*, and the last group contains keys that are common among all three clusters. The maximum number of groups is computed based on all possible combinations of clusters and is given by $\sum_{i=1}^{n-1}(2^i - 1) = 2^n - n - 1$, where n is the number of clusters. Finally, the *coordinator* transfers the groups (containing group keys) to the matching clusters to select global reducers.

The algorithm calculates the makespan for all clusters. Therefore, the time complexity of the algorithm is $O(n^2)$, where n is the number of clusters.

3.4 Phase 3: Global Reducer Selection

The main purpose of Phase 3 is the selection of global reducers among clusters. In this phase, clusters process the transferred groups, split the corresponding key-values for each group, and assign each group to the best cluster. ECMR defines an n-by-m matrix called $requiredData$, where n and m represent the number of clusters and groups, respectively. To select global reducers, the control layer of ECMR collects the volume of each set of key-values corresponding to the groups and stores it in the $requiredData$ matrix.

The problem of global reducer selection is modeled using a *Complete Bipartite Graph*. We define a bipartite graph (C, G, E), where C and G represent the sets of clusters and groups, respectively, and E is a set of edges connecting groups to clusters. The weight of an edge represents the sum of data transfer and processing time of a group on a cluster. In the case of a group of three clusters with four possible groups, the bipartite graph is shown in Figure 6.

Considering Figure 6, if G_1 is processed by the first, second, or third cluster, the associated weights are $w_{1,1}$, $w_{2,1}$, or $w_{3,1}$, respectively. These weights are represented in a matrix called $weightMatrix$, which has dimensions n-by-m. Each

element of the *weightMatrix* is obtained using Equation (9):

$$w_{i,j} = \sum_{k=1}^{n} \frac{requiredData[k][j]}{bw_{ki}} + \textsc{ProcessingTime}(\sum_{k=1}^{n} requiredData[k][j]), \tag{9}$$

where $w_{i,j}$ represents the cost of processing *Group j* on *Cluster i* and BW_{ki} is the bandwidth between *Clusters k* and *i*. The cluster cost is calculated as the sum of the weights of selected edges for a cluster. The objective is to find a mapping from groups to clusters such that the maximum cluster cost is minimized. It should be noted that each group is assigned to only one cluster, and all clusters with at least one assigned group are considered as *global reducers*.

We extend the Gale-Shapley algorithm [9], originally proposed for the Stable Marriage Problem (SMP), to find the optimal mapping. The SMP is a problem that involves finding a stable matching between two sets of equal size, given the preferences of each element. The problem can be stated as follows: "Given a set of n men and n women, where each person ranks all members of the opposite sex in order of preference, form couples between the men and women in such a way that there are no two individuals of opposite sexes who both prefer each other over their current partners. A set of marriages is considered stable when no such pairs exist." In other words, a stable marriage is a matching where no couple would prefer to be paired with someone else instead of their current match. It has been proven that the output of the Gale-Shapley algorithm is free of instabilities, providing the best possible matching results.

In our problem, we encounter two sets with different sizes: the group set and the cluster set. Specifically, the size of the group set is always greater than the cluster set. To handle this situation, we extend the Gale-Shapley algorithm to accommodate the different sizes of the sets. Our extension allows a single cluster to accommodate multiple groups, provided that it has the capacity to process them. Furthermore, in our extended algorithm, it is possible for a cluster to remain unselected for any group, meaning that it may not be assigned to process any group.

The processing cost in the *weightMatrix* serves as the preferences in the extended Gale-Shapley algorithm. The weights in each row of the matrix are sorted in ascending order, representing the preferences of the corresponding cluster. Similarly, the weights in each column are sorted in ascending order, representing the preferences of the corresponding group. The details of the extended Gale-Shapley algorithm are outlined in Algorithm 4.

The algorithm begins with a cluster that has a preference that has not been checked yet (Line 1). It continues until all preferences in all clusters have been checked (Line 2). In Line 3, a cluster c_i is chosen that has at least one preference that has not been checked so far. In Line 4, g_j represents group j that cluster i has not proposed to yet. *Cluster i* proposes to *Group j* in Line 5. In Lines 6 and 7, if *Group j* is not assigned to any cluster, then *Group j* is assigned to *Cluster i*. Otherwise, if *Group j* is already assigned to another cluster, then *Group j* compares its preference for *Cluster i* with its preference for the previous cluster. If *Group j* prefers *Cluster i* over its previous cluster, then *Group j* is assigned to *Cluster i* in Line 9. In Lines 10 to 13, if *Cluster i* is already assigned to some other groups, then the cost of *Group j* is added to the cost of *Cluster i*. As a result, the preferences of *Group j* need to be updated based on the new cost in Line 12. Then, in Line 13, *Group j* checks whether the new cost of *Cluster i* is lower than the cost of its previous assignment. If the new cost is lower, then *Cluster i* is selected to be assigned to *Group j*.

The time complexity of the algorithm is $O(n \times m)$. The while loop in Line 2 runs $n \times m$ times because there are n clusters and each cluster has m preferences for all groups.

Algorithm 4 Global Reducer Selection

Input:
A set of n clusters: $C = \{c_1, c_2, \ldots, c_n\}$
A set of m groups: $G = \{g_1, g_2, \ldots, g_m\}$
A preference list for each cluster and group: for example $P_{c_i} = (g_3, g_1, ..., g_4)$ ▷ Note that a smaller index is a stronger preference.
Output: Assignment of all data groups to clusters

begin with cluster having a preference not checked
while there exists a preference not checked in *Cluster i* **do**
    Choose a *Cluster i* (c_i)
    Let g_j be *group j* that *Cluster i* has not yet proposed to
    Cluster i is proposed to *Group j*
    if *Group j* is not assigned to any cluster **then**
        Group j is assigned to *Cluster i*
    else if *Cluster i* does not accommodate any group **then**
        Group j checks the *Cluster i*'s preference and if $P_{g_j}(c_i) < P_{g_j}(c_{previous})$, then *group j* is assigned to *luster i*
    else
        group j calculates the sum of its own cost associated with *cluster i* along with the cost of *cluster i*
        Group j updates its preference with new cost for *cluster i*
        Group j checks the *Cluster i*'s preference and if $P_{g_j}(c_i) < P_{g_j}(c_{previous})$, then *Group j* is assigned to *Cluster i*

4 PERFORMANCE EVALUATION

Our experiments to evaluate ECMR are divided into four parts. First, we evaluate the processing time estimation using Equation 4. Then, we examine the efficiency of the Data Transfer Time Estimation algorithm (Algorithm 2). After that, we evaluate the efficiency of the Data Balancing algorithm (Algorithm 1). Finally, we assess the overall performance of ECMR.

We use four applications: *Word-count*, *Invertedindex*, *Adjacency-list*, and *Sql-query*, with different ratios of key to value volume to perform experiments.

Word-count: is a classic MapReduce job that counts the number of occurrences of each word in a document or a set of text documents. In a *Word-count*, the size of the value volume is often less than that of the key volume, and the domain of keys are words.

Invertedindex: is a mapping from content, such as words or numbers, to their locations in a document(s). In this application, filenames are considered as the domain of values, and the domain of keys consists of words, with the key volume often smaller than the value volume.

Adjacency-list [5]: is similar to a search-engine computation that generates adjacency and reverse adjacency lists of vertices of a graph for PageRank algorithms. Nodes Id is make up keys in this application, with the key volume often smaller than the value volume.

Sql-query: The obfuscated log data from an administrative application for a real-world job scheduling mechanism is used. Task IDs are considered as the domain of keys. The log data has eight columns of information:

- *TaskId*: a unique number identifies each task.
- *TaskStatus*: indicates that if a task is successfully completed.

[5]https://engineering.purdue.edu/ puma/

- *TaskExecutionTime*: shows the execution time of the task.
- *TaskDeadline*: shows the deadline of the task.
- *TaskArriveTime*: shows the time a task is submitted.
- *TaskStartTime*: indicates the time a task starts to run.
- *TaskWaitTime*: shows the duration a task waits to be ran.
- *TaskType*: shows the type of a task.

For the experiments, we execute queries such as counting the number of task runs and obtaining the average task execution time and the average TaskWaitTime when the TaskStatus is 'Success' in the log file. We do not make any assumptions regarding the presence of common keys across different clusters. Instead, we operate on the principle that the volume of common keys among data in different clusters is often minimal, as demonstrated by our experiments. Our primary focus in this paper is on the performance of job running in terms of makespan time. For *Word-count*, *Adjacency-list*, and *Invertedindex*, we utilize the PUMA dataset [6].

In applications such as *Invertedindex*, *Sql-query*, and *Adjacency-list*, the volume of key sets is significantly lower than the volume of output results. However, in the case of *Word-count*, the volume of key sets is relatively large compared to the volume of output results, with approximately 90% of the results consisting of keys.

We consider a real testbed consisting of four clusters for our experiments. Selecting a small number of clusters for the experiments can be justified by the fact that many applications' servers are located in a few specific places. For example, Telegram servers are distributed worldwide, with five data centers in different regions.

Table 3 displays the data volume used for each application in every cluster. In our experiments, we utilize Apache Hadoop 2.6.5 (Yarn) for MapReduce processing and deploy a copy on each of the four clusters. Each cluster is created over a physical host with a group of VMs. Table 4 presents the specifications of the clusters, including the number of VMs. Our clusters are point-to-point interconnected. We use the Linux application *Wondershaper*[7] to limit the bandwidth between the clusters, providing heterogeneous upstream and downstream bandwidth for each cluster. The upstream and downstream bandwidth of each cluster is also shown in Table 4.

Table 3. The clusters data volume for each application

	Word-count	*Invertedindex*	*Adjacency-list*	*Sql-query*
1	3 GB	3 GB	3.7 GB	4 GB
2	4 GB	4 GB	2.6 GB	3 GB
3	3 GB	3 GB	2.7 GB	2 GB
4	512 MB	512 MB	512 MB	1 GB

4.1 Evaluation of Processing Time Estimation

The processing time estimation is calculated using Equation (4). To obtain α and β, we run each application twice with data sizes of 512 MB and 750 MB. Each run is repeated five times, and the average runtime is recorded for both data volumes. The system of linear equations is then used to calculate α and β.

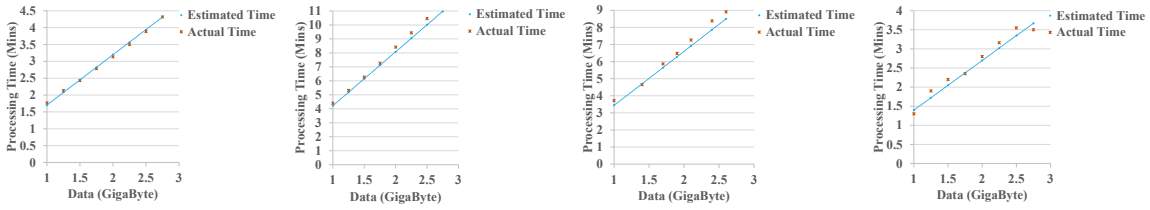
Figures 7a to 7d depict the comparison between the estimated processing time and the actual processing time for different data sizes. As shown in these figures, the estimated processing time is significantly close to the actual time for

[6] https://engineering.purdue.edu/
[7] https://github.com/magnific0/wondershaper

Table 4. clusters details

	Cluster 1	Cluster 2	Cluster 3	Cluster 4
Number of VMs	6	4	4	3
Number of CPU Cores	12	5	5	3
Memory (GB)	12	8	8	4
Upstream (MBps)	1.5	2	1	0.5
Downstream (MBps)	2	2.5	1.5	1

all applications. The *Mean Squared Errors (MSE)* are 0.02, 0.06, 0.1, and 0.02 for *Word-count*, *Invertedindex*, *Adjacency-list*, and *Sql-query*, respectively.



(a) *Word-count* (b) *Invertedindex* (c) *Adjacency-list* (d) *Sql-query*

Fig. 7. Comparison of the estimated time and actual processing time for all applications

4.2 Evaluation of Data Transfer Time Estimation

To evaluate the efficiency of the Data Transfer Time Estimation algorithm, we compare the actual data transfer time among clusters with the estimations made by Algorithm 2 and the sequential data transfer method considered by the linear programming optimization model. For brevity, the Data Transfer Time Estimation and the sequential data transfer time used by the linear programming optimization model will be referred to as the *parallel* (concurrent) and *serial* methods, respectively.

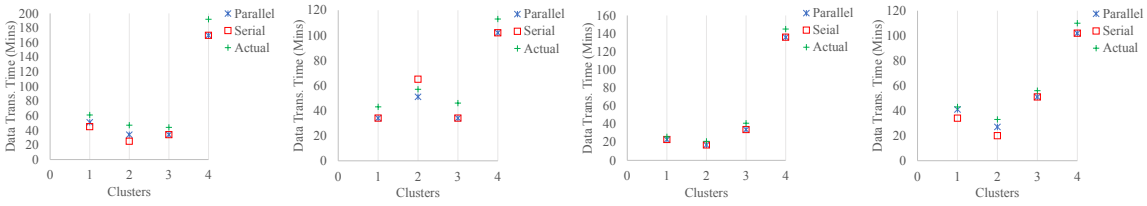
For this purpose, we assume that all clusters contain 2GB of data, and we consider four random data transfer scenarios among the clusters. Each data transfer scenario is represented by Matrix $S_{n\times n}$, where $S_{ij} = d$ indicates that d units of data should be transferred from *Cluster* i to *Cluster* j. Figure 8 displays the four randomly chosen scenarios. We estimate the data transfer time using the *parallel* and *serial* methods and compare them against the actual data transfer for each scenario.

$$S_1 = \begin{pmatrix} 0 & 1 & 1 & 2 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 2 \\ 2 & 1 & 0 & 0 \end{pmatrix} \quad S_2 = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 2 & 0 & 1 & 1 \\ 1 & 2 & 0 & 1 \\ 0 & 2 & 1 & 0 \end{pmatrix} \quad S_3 = \begin{pmatrix} 0 & 0 & 0 & 2 \\ 2 & 0 & 1 & 0 \\ 0 & 0 & 1 & 2 \\ 0 & 1 & 0 & 0 \end{pmatrix} \quad S_4 = \begin{pmatrix} 0 & 1 & 2 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 2 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Fig. 8. Four different data transfer scenarios used in the evaluation of data transfer time estimation

Figures 9a to 9d depict the data transfer time among clusters in Scenarios S_1, S_2, S_3, and S_4. As observed in these figures, both the serial and parallel algorithms generally provide estimations that are close to the actual data transfer

time. However, our proposed parallel approach offers significantly improved scalability compared to the serial approach while also providing more accurate estimations in most cases. This superiority in accuracy can be attributed to the parallel approach's consideration of concurrent data transfer from multiple sources to a specific destination.



(a) scenario one (S_1) (b) scenario two (S_2) (c) scenario three (S_3) (d) scenario four (S_4)

Fig. 9. Comparison of actual, serial and parallel data transfer time for four scenarios

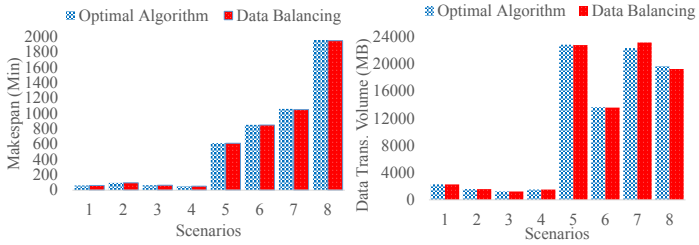
Table 5. The used scenarios for comparing data balancing algorithm

	Clusters	Data	Capacity(C_i)		Clusters	Data	Capacity(C_i)		Clusters	Data	Capacity(C_i)		Clusters	Data	Capacity(C_i)
S1	1	3 GB	0.5	S2	1	3 GB	0.5	S3	1	2 GB	0.5	S4	1	2 GB	1
	2	4 GB	1		2	4 GB	0.5		2	1 GB	0.5		2	1 GB	0.66
	3	3 GB	0.66		3	3 GB	0.5		3	3 GB	0.5		3	3 GB	0.5
	4	0.5 GB	2		4	0.5 GB	0.5		4	1 GB	0.5		4	1 GB	0.5
S5	1	30 GB	0.5	S6	1	50 GB	1	S7	1	80 GB	1	S8	1	100 GB	1
	2	40 GB	1		2	40 GB	0.66		2	25 GB	0.66		2	95 GB	0.66
	3	30 GB	0.66		3	30 GB	0.5		3	35 GB	0.5		3	105 GB	1
	4	5 GB	2		4	5 GB	0.5		4	15 GB	0.5		4	70 GB	0.66

4.3 Evaluation of Data Balancing Algorithm

In this experiment, we compare the performance of the data balancing algorithm (Algorithm 1) with the linear programming optimization model or optimal solution (Equation 1). To conduct the comparison, we randomly generated eight scenarios with varying data volumes and capacities in each cluster. The settings for all scenarios are presented in Table 5. Scenarios S1 to S4 involve small-scale data volumes, while S5 to S8 encompass larger-scale data volumes. For simplicity, we assume $\beta_i = 0$ in both algorithms and $\alpha = \frac{1}{c_i}$ in the data balancing algorithm. It's worth noting that β represents the preprocessing time, which remains constant for both algorithms. Therefore, this assumption does not affect the validity of the results.

To evaluate the algorithms, we examine two metrics: the makespan and the volume of data transfer. Figures 10a and 10b illustrate the makespan and data transfer volume in all scenarios for both the optimal and data balancing algorithms, respectively. As depicted in Figure 10a, the makespans for Scenarios S1 to S8 exhibit a close resemblance in both algorithms, with the data balancing algorithm slightly outperforming the optimal solution in the majority of cases. This can be attributed to the data balancing algorithm's consideration of concurrent data transfer, allowing for more efficient estimation of data transfer times. Similarly, the data transfer volumes show minor differences, mainly noticeable in Scenarios S7 and S8. Overall, the discrepancies are negligible, confirming that the data balancing algorithm performs slightly better with significantly lower computational complexity compared to the optimal solution.



(a) Makespan (b) Data transfer volume

Fig. 10. Comparing (a) makespan and (b) transferred data volume for data balancing and optimal algorithms

Table 6. The volume of key set and output results

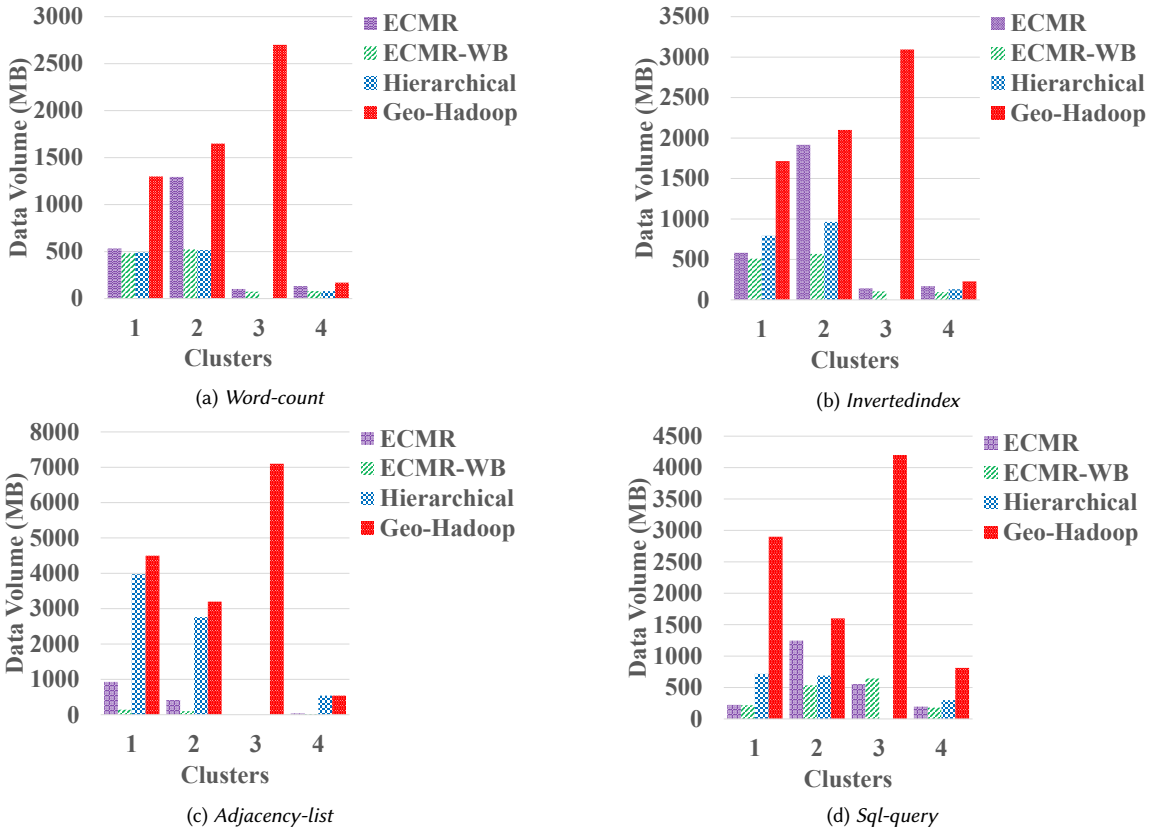
		Cluster 1	Cluster 2	Cluster 3	Cluster 4
Volume of results (MB)	*Word-count*	491	516	413	80
	Invertedindex	792	964	700	131
	Adjacency-list	3880	2700	1810	540
	Sql-query	717	690	450	300
Volume of keys (MB)	*Word-count*	475	494	398	77
	Invertedindex	473	486	396	77
	Adjacency-list	141	98	102	19
	Sql-query	189	200	128	94

4.4 Evaluation of ECMR

In this section, we conduct a series of experiments to assess the effectiveness of ECMR in enhancing the performance of Map-Reduce jobs across multiple geographically distributed clusters. We compare the approaches based on inter-cluster data transfer and makespan. The performance of ECMR with two state-of-the-art approaches: *Hierarchical* [17] and *Geo-Hadoop* [24] is compared. Additionally, we examine ECMR-WB (ECMR without data balancing), a variant of ECMR that lacks the data balancing algorithm, in order to highlight the impact of data balancing algorithm. ECMR-WB is similar to ECMR, with the exception that it does not perform the data balancing algorithm in the initial phase. We apply ECMR-WB to demonstrate the impact of a load balancing algorithm on our method. Through this analysis, we find that the load balancing algorithm improves the efficiency of ECMR.

Since the volume of the key set affects the efficiency of ECMR, we first examine the volume of the key set and the final output results for all applications. Table 6 shows the volume of results and their keys in each application. In the *Word-count* application, keys make up more than 90 percent of the results volume. In the *SQL-query* and *Adjacency-List* applications, keys make up less than 30% and 10% of the results volume, respectively. In the *Inverted-index* application, keys make up about 55% of the results volume. Based on these findings, we anticipate that the algorithm will demonstrate the best performance in processing Adjacency-Lists compared to other applications.

We compare the exported data in all approaches. Before the total inter-cluster data transfer is examined, the amount of exported data from each cluster is depicted in Figures 11a to 11d for each application. Based on the experiments, it can be observed that *Geo-Hadoop* exhibits the highest data transfer among clusters. This can be attributed to the substantial size of intermediate data (output of the map task) in the applications, which is significantly larger compared to the raw data.



(a) *Word-count* (b) *Invertedindex*

(c) *Adjacency-list* (d) *Sql-query*

Fig. 11. The amount of exported data from clusters

Figure 11a illustrates the volume of exported data for the *Word-count* application. As observed in Figure 11a, the *Hierarchical* approach exhibits the minimum exported data across all clusters. This occurs due to the high ratio of the volume of the key set to the volume of results in *Word-count*. In the *Hierarchical* approach, all clusters send their results to *Cluster 3*, which explains the absence of exported data for Cluster 3 in Figure 11a. Following the *Hierarchical* approach, ECMR-WB shows the second minimum exported data. Since *Cluster 3* is selected as the coordinator, its exported data is lower than the other clusters. The relatively low exported data for the fourth cluster is attributed to the small volume of its results. In contrast, ECMR has a higher volume of exported data compared to the *Hierarchical* approach and ECMR-WB. This is justified by the additional data transfer in Phase 1 of ECMR.

Figure 11b shows the volume of exported data for the *Invertedindex* application. In this application, the *Hierarchical* approach shows no data exported by *Cluster 3*, while *Cluster 2* has the highest exported data compared to the other clusters due to its larger size of results. In ECMR, *Cluster 2* exhibits the highest exported data because it transfers raw data in Phase 1 to other clusters. *Clusters 1, 3,* and *4* export slightly more data than the ECMR-WB approach, as a portion of raw data is sent from *Cluster 2* to other clusters. Consequently, the corresponding values in those clusters need to be collected again in a global reducer.

The volume of exported data is depicted in Figure 11c for the *Adjacency-list* application. In this application, only 2 phases are executed, as Phase 3 is not run due to the absence of common keys between clusters. Moreover, the

ratio of the key set volume to the results volume is relatively small, resulting in significantly lower data transfer in ECMR-WB compared to the other approaches. In the ECMR approach, inter-cluster data transfer is slightly higher than ECMR-WB due to the inclusion of the data balancing phase. Specifically, in Phase 1, *Clusters 1* and *2* send a portion of raw data to *Clusters 3* and *4* to enhance performance. In contrast, the *Hierarchical* approach exhibits a significantly higher inter-cluster data transfer due to the large volume of results.

Figure 11d shows the size of exported data for the *Sql-query* application. In ECMR, the exported data in *Cluster 2* exceeds that of the *Hierarchical* and ECMR-WB approaches due to the data balancing process. In the *Hierarchical* approach, the exported data from *Cluster 3* is zero because it is designated as the cluster to which all other clusters send their results.

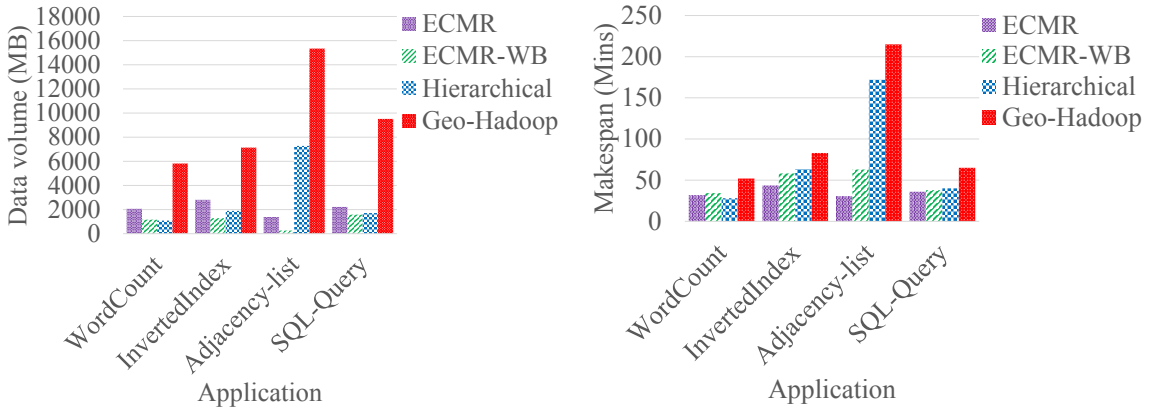
In summary, for all applications except *Word-count*, ECMR-WB exhibits the minimum inter-cluster data transfer. However, in the case of the *Word-count* application, the *Hierarchical* approach demonstrates the minimum data transfer between clusters due to the fact that the size of keys accounts for more than 90% of the size of the results.

Figures 12a and 12b depict the total inter-cluster data transfer and makespan for all applications, respectively. Despite ECMR-WB exhibiting lower inter-cluster data transfer compared to the other approaches, ECMR achieves the lowest makespan. ECMR delivers superior performance in terms of makespan due to the following reasons. Firstly, the data balancing algorithm in Phase 1 significantly enhances overall performance. Secondly, the concept of transmitting keys instead of complete results effectively reduces inter-cluster data transfer, but it does not ensure optimal performance in cases of high data imbalance. Hence, by incorporating data balancing and the transmission of keys instead of complete results, ECMR guarantees the best performance among all the approaches.

In the *Word-count* application, ECMR outperforms *Geo-Hadoop* by 38% and ECMR-WB by 8% in terms of makespan. However, the *Hierarchical* approach achieves a 9% better result than ECMR due to the high ratio of keys to results in this specific application. As shown in Figure 12b, ECMR's makespan surpasses *Geo-Hadoop* by 44%, *Hierarchical* by 10%, and ECMR-WB by 5% in the *Sql-query* application. For the *Adjacency-list* application, ECMR outperforms *Geo-Hadoop* by 85%, *Hierarchical* by 81%, and ECMR-WB by 50%. In the *Invertedindex* application, ECMR surpasses *Geo-Hadoop* by 48%, *Hierarchical* by 31%, and ECMR-WB by 25%. In summary, ECMR attains the best performance for applications with a lower ratio of keys to results, which is the case for many common MapReduce applications.

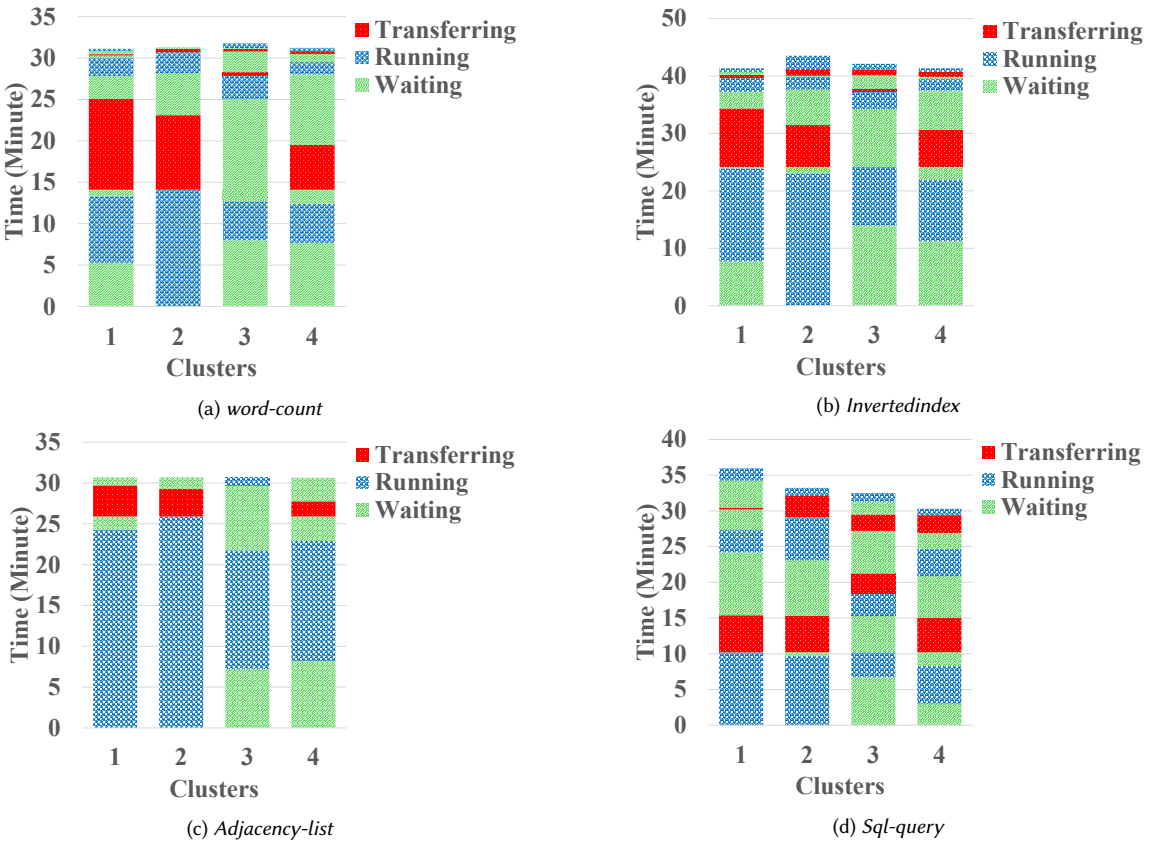
4.4.1 Breakdown of time spent at different states. To gain a better understanding of how ECMR achieves its high performance, Figures 13a to 13d present the state of each cluster (data transferring, running job, waiting for data) along with the corresponding time spent in each state for each application. As depicted in Figures 13a and 13b, *Cluster 2* immediately starts executing the job, while *Clusters 1, 3,* and *4* must wait for data reception. According to the data balancing algorithm, *Cluster 2* transfers the raw data to the other clusters in the *Word-count* and *Invertedindex* applications. In the case of *Adjacency-list* and *Sql-query* applications, as illustrated in Figures 13c and 13d, *Clusters 3* and *4* experience a waiting period to receive data, while *Clusters 1* and *2* can immediately initiate the job. Across all applications, clusters generally complete their tasks around the same time, except in the *Sql-query* application. This discrepancy arises from inaccuracies in the data transfer estimation, resulting in varying network delays.

Cluster 3 is consistently chosen as the *coordinator* through the "Coordinator Selection" algorithm in all applications. Consequently, in all applications, it awaits the receipt of the keys set from other clusters after Phase 1. It is important to note that in the *coordinator*, the data transfer time (Step 4 in Figure 3) and the job processing time (Step 5 in Figure 3) overlap. Therefore, Step 5, which involves extracting values of grouped keys, is not depicted for any of the applications in the *coordinator*. ECMR exhibits additional steps, all of which are showcased in the figures. As demonstrated in Figures



(a) inter-cluster data transfer (b) makespan

Fig. 12. a) makespan and b) total inter-cluster data transfer for all applications



(a) *word-count* (b) *Invertedindex*

(c) *Adjacency-list* (d) *Sql-query*

Fig. 13. Breakdown of time spent at different states by ECMR

13a to 13d, the additional local processing time in clusters compared to the data transfer time is negligible in ECMR. As a result, ECMR completes the job earlier than the other approaches. This is because ECMR significantly reduces the makespan at the expense of increased local computation and data transfer volume.

5 DISCUSSIONS

In some real-world applications, multiple jobs are executed in a chain, where each job may rely on the output of the preceding job as its input and may introduce new datasets within the chain. To address this scenario, ECMR incorporates the storage of output result addresses within the *DataManager* component of the first layer, allowing for their utilization in subsequent jobs. Although ECMR can be applied to such applications, it is not the primary focus of this paper, and as a result, it may not provide an optimal solution in these particular cases. Future work could delve deeper into this aspect.

In practice, the percentage of common keys across different clusters (data centers) is often very low. For example, in social networks, users can be from different languages and countries, and their information is stored regionally, negating the need for storage across all regions. The validity of this assertion has been examined in our previous paper [18]. We examined two files named 'hadoop_tutorial' and 'mapreduce_tutorial,' which are conceptually similar. The 'hadoop_tutorial' file contained 2,899 records, while the 'mapreduce_tutorial' file held 2,821 records. Of these, only 202 records had common keys, indicating that less than 4% of the total records shared common keys. Therefore, even though we employed an arbitrary method for selecting data and sending it to other clusters in the data balancing algorithm, the proposed method still outperforms other methods in terms of efficiency. Additional research and exploration on data selection could further improve the algorithm's effectiveness.

While this paper focuses on Big Data applications, we made a deliberate decision to limit our data volumes to the GB range in order to ensure efficient experimentation in laboratory-scale settings and the ability to conduct multiple iteration of experiments within a reasonable timeframe. However, it is important to emphasize that the size of the data volume does not have a significant impact on the trends and relative performance gains we observed. Our findings reveal a clear and consistent linear correlation between processing time and data volume size, as demonstrated in Figure 7. Therefore, the validity and generalizability of our results remain robust across varying data volume sizes.

6 RELATED WORK

In data-driven applications, efficient communication between nodes plays a crucial role in achieving optimal system performance. Numerous studies have focused on task scheduling and data transfer optimization within a single cluster to minimize runtime and data transfer. However, in scenarios where data is distributed across multiple clusters, inter-cluster data transfer can become a bottleneck, posing new challenges for optimization. Considerable research has been conducted on data distribution across various data centers. For example, Sherma et al. [20] discuss about geo-distributed cloud data center energy-efficient workflow scheduling approaches. They classified the heterogeneous workloads from Google dataset using machine learning techniques for energy-efficient assignment to appropriate data centers. Ullah et al. [21] evaluate the performance of distributed data processing frameworks hosted in private and public clouds. They evaluate the performance of Hadoop, Spark, and Flink in a hybrid cloud in terms of execution time, resource utilization, horizontal scalability, vertical scalability, and cost. Emara et al. [8] develop a geographically distributed data management framework to efficiently handle large-scale data distributed across multiple data centers, with a focus on optimizing data block utilization for diverse analytical tasks. In the following sections, we review related works focusing on the *Hierarchical* and *Geo-Hadoop* approaches.

6.1 Geo-Hadoop approach

Wang et al. [24] introduced the G-Hadoop framework, which enables the processing of geo-distributed data across multiple clusters without requiring changes to the existing cluster architecture. In G-Hadoop, data is stored in a geo-distributed file system called the Gfarm file system. In G-Hadoop clusters, specialized hardware interconnected by high-performance networks, such as Infiniband, is commonly employed. In our system, clusters can communicate with each other through the Internet.

Meta-MapReduce [2] aims to minimize the amount of data that needs to be transferred between different locations by only transferring essential data required to obtain the final result. It takes into consideration the data and task locality, avoiding unnecessary movement of data that does not contribute to the final output. In Meta-MapReduce, users send metadata to the site of mappers instead of sending the original data. Mappers and reducers operate on metadata, and when needed, reducers retrieve the required original data from the users' site to generate the desired result.

Li et al. [16] proposed an algorithm for minimizing inter-DC traffic during the shuffle phase by addressing both data and task allocation problems in the context of MapReduce. The algorithm identifies data centers with high output-to-input ratios and poor network bandwidth, and then migrates their data to more capable data centers.

Jayalath et al. [13] introduced G-MR, a Hadoop-based framework for running MapReduce jobs across multiple data centers. Unlike G-Hadoop, G-MR does not randomly place reducers [27]; instead, it utilizes a single directional weighted graph for data movement, employing the shortest path algorithm. Unlike ECMR, which can be applied to any framework that supports MapReduce, G-MR is specifically designed based on a Hadoop-based framework.

Heintz et al. [11] introduced shuffle-aware data pushing during the map phase. In this method, they identify the mappers that significantly affect job completion within a data center and exclude mappers that would cause delays. Essentially, they select mappers capable of executing a job and shuffling intermediate data within a specified time constraint.

Resilin [12] offers a hybrid cloud-based MapReduce computation framework. Resilin implements the Amazon Elastic MapReduce (EMR) [8] interface and leverages existing Amazon EMR tools for system interactions. It allows users to process data stored in one cloud using resources from other clouds.

Chen et al. [6] propose an efficient data-placement technique that considers both network traffic reduction and Quality of Service (QoS) guarantees for data blocks to optimize communication resources. They formulate the joint optimization of the data-placement problem, present a generic model for minimizing communication costs, and show the NP-hardness of the joint data-placement problem.

6.2 Hierarchical approach

Hierarchical MapReduce (HMR)[17] is a two-level programming model where the upper level consists of a global controller layer, and the lower layer comprises multiple clusters that execute MapReduce jobs. HMR processes data independently in each cluster, and a single global reducer collects the results generated by other clusters. Finally, the global reducer is executed to generate the final result. An extension to HMR is proposed in[4], where the authors suggest considering the amount of data to be transferred and the resources required to produce the final output at the global reducer. However, similar to HMR, this extension does not take into account heterogeneous inter-DC bandwidth and the available resources at the clusters [7]. Another extension is presented in [15], where the authors consider

[8] http://aws.amazon.com/elasticmapreduce

the availability of cluster resources and different network link capacities. In these studies, the raw data is initially centralized, and frameworks distribute the data. In contrast, our system operates based on initially distributed data.

In our previous work [18], we presented a framework to address the limitations of *Hierarchical* and *Geo-Hadoop* solutions in homogeneous settings. In [18], we assume that each cluster has equal data volume and processing power, eliminating the need for a data balancing algorithm. One of the challenges with the *Hierarchical* solution is that all cluster outputs need to be sent to a single cluster, even though a large portion of the transferred data is unnecessary for producing the final results. To tackle this issue, we propose the Global Reduction Graph (GRG) in [18]. In contrast, in this paper, we propose ECMR for heterogeneous environment. In the ECMR framework, we propose the bipartite graph to answer the following three important questions: (i) How to distribute data among the clusters? (ii) What fraction of the results will be sent to the global reducers by considering the heterogeneity of clusters and bandwidth? (iii) What are the best global reducers? To answer these questions, we propose Data Balancing algorithm to determine how much data should be transferred and to which cluster in a way that local MapReduce jobs are finished simultaneously, Data Transfer Time Estimation algorithm to estimate the data transfer time between clusters by considering effect of concurrent data transfer on bandwidth, Global Reducer Selection to select the best cluster for generating the final result and Coordinator Selection algorithm to group keys and specify the required keys for global reducers.

7 CONCLUSIONS AND FUTURE WORK

The transfer of data between cloud data centers poses a major challenge in processing geo-distributed data, significantly impacting the processing time of such applications. In this paper, we proposed ECMR, a solution for geo-distributed data processing that takes into account cluster heterogeneity and varying available bandwidth among them. In the first phase, ECMR redistributes raw data among clusters to achieve better data balance. It then determines the portion of temporary results that needs to be transferred to global reducers by processing the keys. Finally, ECMR groups the keys, extracts values, and presents them using a bipartite graph to generate the final results. The results demonstrated that ECMR significantly reduces inter-cluster data transfers and the overall makespan. It proves highly effective, especially in scenarios where the volume of the key set is smaller compared to the volume of the value set. For future research, we are interested in investigating the impact of data balancing on global reduction and exploring how raw data redistribution can further reduce inter-cluster data transfers in the final phase. Additionally, we aim to adapt our proposed method to applications with deadline constraints. Another area of interest is studying the challenges related to data privacy and the geolocation of sensitive data.

REFERENCES

[1] [n. d.]. 5 tips for building a successful hybrid cloud. https://www.computerworld.com/article/2834193/5-tips-for-building-a-successful-hybrid-cloud.html.

[2] Foto Afrati, Shlomi Dolev, Shantanu Sharma, and Jeffrey D Ullman. 2015. Meta-MapReduce: A technique for reducing communication in MapReduce computations. *arXiv preprint arXiv:1508.01171* (2015).

[3] Michael Cardosa, Chenyu Wang, Anshuman Nangia, Abhishek Chandra, and Jon Weissman. 2011. Exploring MapReduce Efficiency with Highly-Distributed Data *(MapReduce '11)*. Association for Computing Machinery, New York, NY, USA, 27–34. https://doi.org/10.1145/1996092.1996100

[4] Marco Cavallo, Giuseppe Di Modica, Carmelo Polito, and Orazio Tomarchio. 2016. Application profiling in hierarchical Hadoop for geo-distributed computing environments. In *2016 IEEE symposium on computers and communication (ISCC)*. IEEE, 555–560.

[5] Marco Cavallo, Carmelo Polito, Giuseppe Di Modica, and Orazio Tomarchio. 2016. H2F: a Hierarchical Hadoop Framework for big data processing in geo-distributed environments. In *Proceedings of the 3rd IEEE/ACM International Conference on Big Data Computing, Applications and Technologies*. 27–35.

[6] Wuhui Chen, Baichuan Liu, Incheon Paik, Zhenni Li, and Zibin Zheng. 2021. QoS-Aware Data Placement for MapReduce Applications in Geo-Distributed Data Centers. *IEEE Transactions on Engineering Management* 68, 1 (2021), 120–136. https://doi.org/10.1109/TEM.2020.2971717

[7] Shlomi Dolev, Patricia Florissi, Ehud Gudes, Shantanu Sharma, and Ido Singer. 2019. A Survey on Geographically Distributed Big-Data Processing Using MapReduce. *IEEE Transactions on Big Data* 5, 1 (2019), 60–80. https://doi.org/10.1109/TBDATA.2017.2723473

[8] Tamer Z Emara, Thanh Trinh, and Joshua Zhexue Huang. 2023. Geographically distributed data management to support large-scale data analysis. *Scientific Reports* 13, 1 (2023), 17783.

[9] David Gale and Lloyd S Shapley. 1962. College admissions and the stability of marriage. *The American Mathematical Monthly* 69, 1 (1962), 9–15. https://doi.org/10.1080/00029890.1962.11989827

[10] Kenneth A Hawick, Paul D Coddington, and Heath A James. 2003. Distributed frameworks and parallel algorithms for processing large-scale geographic data. *Parallel Comput.* 29, 10 (2003), 1297–1333. https://doi.org/10.1016/j.parco.2003.04.001

[11] Benjamin Heintz, Abhishek Chandra, Ramesh K Sitaraman, and Jon Weissman. 2014. End-to-end optimization for geo-distributed mapreduce. *IEEE Transactions on Cloud Computing* 4, 3 (2014), 293–306. https://doi.org/10.1109/TCC.2014.2355225

[12] Anca Iordache, Christine Morin, Nikos Parlavantzas, Eugen Feller, and Pierre Riteau. 2013. Resilin: Elastic mapreduce over multiple clouds. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*. IEEE, 261–268.

[13] Chamikara Jayalath, Julian Stephen, and Patrick Eugster. 2013. From the cloud to the atmosphere: Running MapReduce across data centers. *IEEE Trans. Comput.* 63, 1 (2013), 74–87. https://doi.org/10.1109/TC.2013.121

[14] Konstantinos Kloudas, Margarida Mamede, Nuno Preguiça, and Rodrigo Rodrigues. 2015. Pixida: optimizing data parallel jobs in wide-area data analytics. *Proceedings of the VLDB Endowment* 9, 2 (2015), 72–83.

[15] Linh Le, Jie Hao, Ying Xie, and Jennifer Priestley. 2016. Deep Kernel: Learning Kernel Function from Data Using Deep Neural Network. In *Proceedings of the 3rd IEEE/ACM International Conference on Big Data Computing, Applications and Technologies* (Shanghai, China) *(BDCAT '16)*. Association for Computing Machinery, New York, NY, USA, 1–7. https://doi.org/10.1145/3006299.3006312

[16] Peng Li, Song Guo, Toshiaki Miyazaki, Xiaofei Liao, Hai Jin, Albert Y Zomaya, and Kun Wang. 2016. Traffic-aware geo-distributed big data analytics with predictable job completion time. *IEEE Transactions on Parallel and Distributed Systems* 28, 6 (2016), 1785–1796. https://doi.org/10.1109/TPDS.2016.2626285

[17] Yuan Luo and Beth Plale. 2012. Hierarchical mapreduce programming model and scheduling algorithms. In *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*. IEEE, 769–774.

[18] Saeed Mirpour Marzuni, Abdorreza Savadi, Adel N. Toosi, and Mahmoud Naghibzadeh. 2021. Cross-MapReduce: Data transfer reduction in geo-distributed MapReduce. *Future Generation Computer Systems* 115 (2021), 188–200. https://doi.org/10.1016/j.future.2020.09.009

[19] Daniel A. Reed and Jack Dongarra. 2015. Exascale Computing and Big Data. *Commun. ACM* 58, 7 (jun 2015), 56–68. https://doi.org/10.1145/2699414

[20] Anu Priya Sharma and Jaspreet Singh. 2024. Energy Efficient Distribution of Heterogeneous Workloads in Cloud Data Center. In *2024 3rd International Conference for Innovation in Technology (INOCON)*. IEEE, 1–6.

[21] Faheem Ullah, Shagun Dhingra, Xiaoyu Xia, and M Ali Babar. 2024. Evaluation of distributed data processing frameworks in hybrid clouds. *Journal of Network and Computer Applications* (2024), 103837.

[22] Abhishek Verma, Ludmila Cherkasova, and Roy H Campbell. 2011. Resource provisioning framework for mapreduce jobs with performance goals. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer, 165–186.

[23] Ashish Vulimiri, Carlo Curino, Philip Brighten Godfrey, Thomas Jungblut, Konstantinos Karanasos, Jitendra Padhye, and George Varghese. 2015. WANalytics: Geo-Distributed Analytics for a Data Intensive World. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1087–1092. https://doi.org/10.1145/2723372.2735365

[24] Lizhe Wang, Jie Tao, Rajiv Ranjan, Holger Marten, Achim Streit, Jingying Chen, and Dan Chen. 2013. G-Hadoop: MapReduce across distributed data centers for data-intensive computing. *Future Generation Computer Systems* 29, 3 (2013), 739–750. https://doi.org/10.1016/j.future.2012.09.001

[25] Carolyn Watters and Ghada Amoudi. 2003. GeoSearcher: Location-based ranking of search engine results. *Journal of the American Society for Information Science and Technology* 54, 2 (2003), 140–151. https://doi.org/10.1002/asi.10191

[26] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica, et al. 2010. Spark: Cluster computing with working sets. *HotCloud* 10, 10-10 (2010), 95.

[27] Jiangtao Zhang, Lingmin Zhang, Hejiao Huang, Zeo L Jiang, and Xuan Wang. 2016. Key based data analytics across data centers considering bi-level resource provision in cloud computing. *Future Generation Computer Systems* 62 (2016), 40–50. https://doi.org/10.1016/j.future.2016.03.008

Received 28 June 2023; revised ?? June 20??; accepted ?? June 20??