A Coflow-based Co-optimization Framework for High-performance Data Analytics

Long Cheng[1], Ying Wang[2], Yulong Pei[1], and Dick Epema[3]

[1] Eindhoven University of Technology, The Netherlands
[2] Institute of Computing Technology, Chinese Academy of Sciences, China
[3] Delft University of Technology, The Netherlands
l.cheng@tue.nl wangying2009@ict.ac.cn y.pei.1@tue.nl d.h.j.epema@tudelft.nl

***Abstract*—Efficient execution of distributed database operators such as joining and aggregating is critical for the performance of big data analytics. With the increase of the compute speedup of modern CPUs, reducing the network communication time of these operators in large systems is becoming increasingly important, and also challenging current techniques. Significant performance improvements have been achieved by using state-of-the-art methods, such as reducing network traffic designed in the data management domain, and data flow scheduling in the data communications domain. However, the proposed techniques in both fields just view each other as a black box, and performance gains from a co-optimization perspective have not yet been explored.**

In this paper, based on current research in coflow scheduling, we propose a novel Coflow-based Co-optimization Framework (CCF), which can co-optimize application-level data movement and network-level data communications for distributed operators, and consequently contribute to their performance in large distributed environments. We present the detailed design and implementation of CCF, and conduct an experimental evaluation of CCF using large-scale simulations on large data joins. Our results demonstrate that CCF can always perform faster than current approaches on network communications in large-scale distributed scenarios.

***Keywords*-big data; coflow scheduling; distributed joins; network communications; data-intensive applications**

I. Introduction

An increasing number of companies rely on the results of massive data analytics to improve their business operations, customer service and risk management. Moreover, scientific researchers such as bioinformaticians are also increasingly looking into their data to find valuable insights about disease and personal health [1], [2]. To cope with the added weight of current Big Data in a timely manner, high-performance analytics over large-scale systems, such as data centers equipped with hundreds or even thousands of servers, is becoming the mainstream.

As one of the key tasks in such scenarios, efficient execution of distributed operators such as joining and aggregating is still challenging current techniques. Specifically, in large-scale distributed environment the core performance challenge is network communications. The main reason is that these operators always bring in expensive data shuffling, which consumes tremendous network resources and thus results in long communication time. In fact, in recent years, as the performance of CPU has grown much faster than network bandwidth, the network has become a performance bottleneck to computation, even in a single data center [3]. Moreover, current work has shown that these operators in data queries could spend more than 65% of their completion time on transferring data even in a small network [4]. Therefore, efficient optimization on the executions of these operators, which can minimize data communication time, becomes increasingly desirable.

Reducing the volume of transferred data over networks is an efficient way to speedup the operators [4]. Based on this, various advanced approaches [5] and strategies [6] have been proposed in the *data management* domain. Their philosophy is generally to move *small* data chunks instead of *large* data chunks during their executions. For example, track-join [4], has adopted a very fine-grained way, which can search all possible opportunities on reducing data movement, and consequently minimize the network traffic in join executions. Although all the approaches are shown to be very efficient, minimizing communication traffic does **not** necessarily lead to minimal communication time. This is because when computing nodes use the network without any coordination, utilization of network bandwidth could be very poor. For instance, for a join implementation, if all computing nodes first send their data to the first node, then to the second node, and so on, then there will be significant network congestion, since the nodes will compete for the bandwidth of a single link while other links are not fully utilized [7].

To improve network communication time of current data applications, scheduling over the abstraction *coflow* [8], which is defined as a group of parallel data flows that are related to each other (e.g., shuffle flows in MapReduce), is being studied in the *data communications* domain. Rather than individual flows, current work focuses on improving the performance of data flows for a job (e.g., a parallel join), such as minimizing the completion time of the slowest flow. To date, several solutions, such as Varys [8], Aalo [9] and RAPIER [10] have been shown to be very efficient on coflow processing, and thus they can be directly applied to current systems to speedup big data analytics. Regardless, all these techniques focus on **network-level optimization**, which is decoupled from application-level optimization/scheduling, and thus they could lead to suboptimal performance (we demonstrate this problem with more details in Section II).

With targets for more efficient big data analytics in large-scale systems, in this work, we present a novel **C**oflow-based **C**o-optimization **F**ramework (CCF). This framework can co-optimize the schedules of application-level data movement and network-level data communications, and thus it can speedup current distributed operators. The main contributions of this paper can be summarized as follows:

- We demonstrate that additional performance gains in distributed operators can be achieved, by co-optimizing current scheduling techniques studied in the data management and data communication domains.
- We present the detailed design and implementation of the proposed CCF, with describing the co-optimization problem by a detailed mathematical model.
- We develop a fast and efficient algorithm to approximately solve our theoretical NP-complete problem, in terms of system implementations.
- Extensive simulations show that CCF can indeed speedup network communications for distributed data operators in the presence of big data.

In the following, we will focus on describing our techniques based on the detailed execution of the joins. The proposed techniques can be similarly applied to other distributed operators, such as aggregation and duplicate elimination.

The remainder of this paper is organized as follows. In Section II, we introduce the background with a motivating example of this work. We present our framework design and its detailed implementation in Section III. We carry out extensive evaluation of our approach in Section IV. We report the related work in Section V and conclude this paper in Section VI.

II. Background and Motivation

In this section, we have an overview of the techniques on distributed joins and the coflow abstraction. Moreover, we also illustrate the advantages of applying the techniques from a co-optimization perspective through a motivating example.

A. Distributed Join Executions

The execution of a distributed join can be broadly decomposed into an initial data redistribution stage followed by a local join process [11], [5]. This latter process has been extensively studied and its cost does not contain any inter-machine communication. For the purpose of this work, we focus on the former phase.

In fact, the process of the data redistribution replies on the scheduling of the data movement of the input data. This schedule process could be very simple. For example, in a hash-based join [11], data tuples are assigned based on the hash values of their join keys. In the meantime, the process could be also very complex, e.g., track-join [4] adopts a four phase schedule in its implementations.

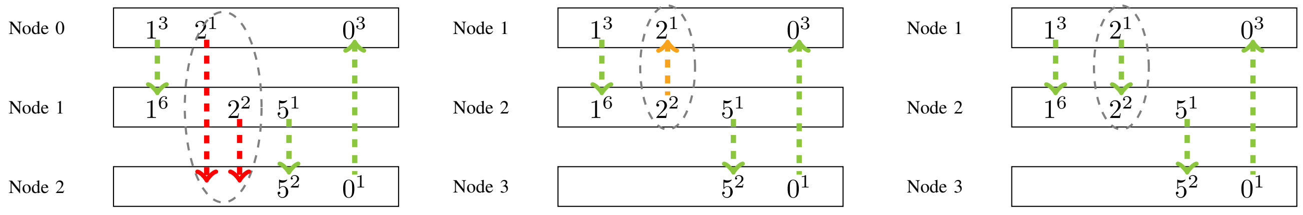
A *join* facilitates the combination of two relations based on a common key, if we want to implement a join in a distributed environment, then all the tuples with the same join key must be co-located on a same computing node. To show more details of current schedule techniques in this aspect, an example of three possible schedule plans of a distributed join over a three-node system is demonstrated as Figure 1. There, each data tuple is presented by its join key and the superscript of each key means the frequency it appears. For instance, 1^3 in means that there are three tuples with the join key 1. Also, the dashed arrows in each subfigure mean the outputs of the scheduling, i.e., the destination node of each tuple, which will be delivered to underlying systems for the final execution. Figure 1(a) shows the details of a hash-based join. There, we use a very simple hash function to assign the destination of input data, i.e., the hash value of a join key is a modulus of the value of key and the number of nodes. For example, tuples with keys 2 and 5 will be scheduled to Node 2 in join executions.

If we quantify the cost of network communication by the number of tuples moved to *remote* nodes, then the cost of three schedules in Figure 1 will be 8 (i.e., 3+1+2+1+1), 7, and 6 respectively. Within the scope of current study in the filed of data management (e.g., [4], [6]), the schedule plan SP_2 will be considered as an optimal solution and chosen by underlying systems, because it transfers **less** data than other two approaches.

B. The Coflow Abstraction

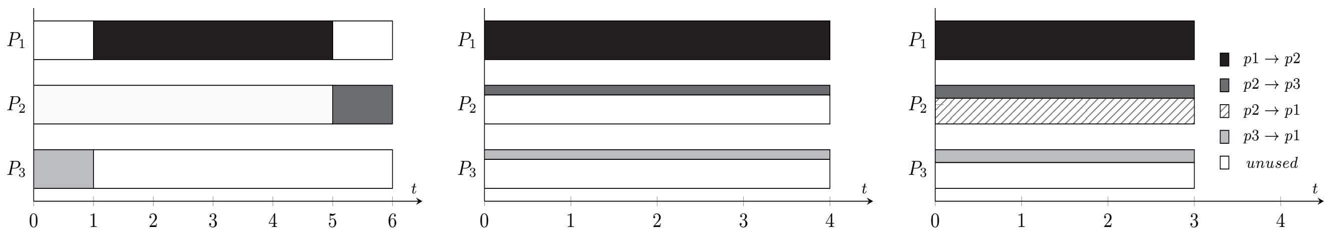
The coflow abstraction was first proposed in [12] to define a group of parallel data flows that are related to each other and also share a common performance goal. To optimize the performance of big data analytics, we need to optimize data flows transferred at the level of coflow rather than individual ones. This is because the completion time of a job (i.e., a distributed join) depends on the time it takes to complete the entire coflow, which is also called the *coflow completion time* (CCT), instead of the time to complete the individual flows composing it [10].

An individual flow f within a coflow can be defined by a 3-tuple $[src, des, v]$, where src and des are the source and destination nodes, and $v > 0$ is the flow volume [13]. In fact, coflows have been shown to be able to express most communication patterns in data-parallel applications [12]. For example, the three data flows in the plan SP_2 can be seen as a single coflow (note that the tuples with keys 1 and 2 in SP_2 will be combined as a single data flow in real implementations because their source and destination are the same). In this condition, the problem of improving network communication time of a join execution can be transformed into the problem of optimizing its CCT cost.



(a) SP_0 (hash-based) (b) SP_1 (suboptimal) (c) SP_2 (optimal)

Figure 1. An example of difference schedule plans (SP) for data movement in a distributed join over three nodes (application-level).



(a) the worst schedule for SP_2 (b) optimal coflow schedule for SP_2 (c) optimal coflow schedule for SP_1

Figure 2. Network-level schedules for the application-level SPs presented in Figure 1. The vertical height of each filled bar indicates the used bandwidth.

Online coflows (e.g., each individual flow starts at a different time point [9]) are very common in real data systems, and their scheduling techniques have been studying in the filed of data communications. Regardless, without sacrificing generality, we assume that each individual data flow starts at the same time in this work. In the meantime, though detailed performance analysis of coflows in complex network environments (e.g., routing [10]) have been explored, we adopt an approach similar to that of Varys [8] and model the underlying network abstract as a non-blocking switch, which interconnects all the machines[1]. Moreover, we assume that all the network ports have the same normalized unit capacity, and bandwidth competition only appears in ingresses or egresses. Such an abstraction is simple, yet it is practically reasonable and matches with recent full bisection bandwidth topologies widely used in current production data centers [8].

C. Potential Benefits of Co-optimization

As described in Section I and demonstrated in Figure 1, current optimization techniques on join executions in the data management domain only focus on application-level data movement, but have not considered the impacts of underlying data communications. On the other hand, current research on coflow scheduling in data communication domain assumes that the detailed information (i.e., $[src, des, v]$) of each data flow is known before a coflow starts [8], [10], [13]. Namely, they have not realized that high-level scheduling of data movement could actually impact the final communication performance.

To illustrate above problems, here we give an example of network-level schedules for the previously described application-level schedule plans SP_1 and SP_2. Assuming that each node (i.e., network port P_i) transfers one data tuple in one time unit, for the plan SP_2, two potential strategies on its coflow scheduling are demonstrated in Figure 2(a) and Figure 2(b) respectively[2]. Obviously, it can be observed that an optimal coflow schedule can indeed decrease the CCT of SP_2, from 6 to 4. However, as shown in Figure 2(c), using an optimal coflow scheduling, a sub-optimal application-level plan SP_1 can even lead to a better performance, i.e., the CCT of SP_1 is 3. Then our question is: to achieve the best performance on distributed operators, *where should the data exactly go*? In fact, the above example has implied that both the schedules on data locality and data communications must be jointly considered in order to minimize the CCT cost, which motivates our design as below.

III. CCF Design and Implementation

In this section, we introduce the detailed design of CCF. Also, we present an efficient implementation of our approach in large systems.

A. Architecture and Model

A logical view of the proposed CCF architecture is demonstrated in Figure 3. There, an analytical job is decomposed into a sequential distributed data operators. For each operator, based on the input information of underlying data and network, the application-level scheduler and the coflow scheduler at the schedule/control layer will co-optimize to

[1]Note that our proposed framework is based on the coflow abstraction, thus it can be extended to online and complex network cases very easily.

[2]Here, we use a bandwidth-based model [10] to describe the coflow scheduling. Namely, all the data flows are ended at the same time point.

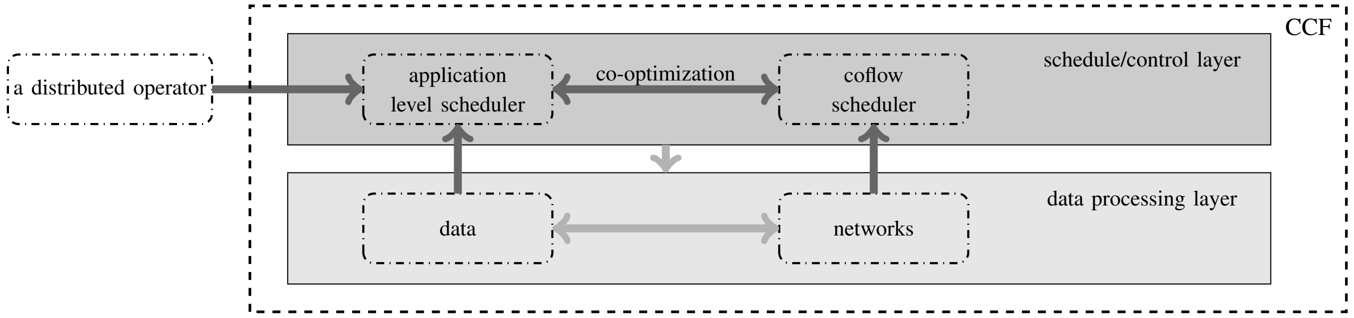


Figure 3. A logical view of the architecture of CCF.

get an optimal execution plan on the final data movement. After that, the plan will be delivered to the underlying data processing layer (e.g., a coflow system) for execution.

To describe the detailed co-optimization process of a join execution, we use the follow model: There are n computing nodes and the two input relations on each node have been partitioned into p parts. For a general case, we assume that data tuples are hash partitioned based on their join keys in this work, the hash value of the k-th partition is k, and the size of included data chunks[3] at each node i is denoted as h_{ik}. Tuples at each node with the same hash value must be assigned to a same node to implement the final local joins, therefore, we can use the decision variables $x_{jk} \in \{0, 1\}$ to indicate whether a data partition k is assigned to the node j. Namely, $x_{jk} = 1$ represents that partition k is assigned to node j, and $x_{jk} = 0$ means not. Moreover, we use f_{ij} to indicate the data flow generated by data movement from node i to node j, the size of which is v_{ij}, the assigned bandwidth for the transmission is b_{ij}, and the set of its communication links is L_{ij}. Obviously, this model contains both high-level scheduling of data assignment (i.e., x_{jk}) and low-level scheduling of data communications (i.e., b_{ij}). For convenience of our presentation, we use the notations as listed in Table I.

We only compute the network communication cost for the data moved to **remote** nodes, since a local movement will not consume any network resources. Based on the information of data flow source, destination, volume and the network resource status (i.e., the residual bandwidth on each link), we can formulate the problem to minimize the network communication time t of a join/query execution (i.e., CCT) as following:

$$minimize \quad t \tag{1}$$

Table I
TABLE OF NOTATIONS

Notation	Meaning
n	number of computing nodes
p	number of data partitions
x_{jk}	decision variable to indicate whether the k-th partition is assigned to node j
h_{ik}	the size of the k-th data chunk on node i
f_{ij}	the data flow from node i to node j
b_{ij}	transmission bandwidth assigned to f_{ij}
v_{ij}	size of data flow f_{ij}
L_{ij}	the link set of data flow f_{ij}
R_l	the available bandwidth of link l

subject to:

$$v_{ij} = \sum_{k=1}^{p} h_{ik} x_{jk} \quad \forall i \neq j \tag{1.1}$$

$$t = \frac{v_{ij}}{b_{ij}} \tag{1.2}$$

$$\sum_{j=1}^{n} x_{jk} = 1 \quad \forall k \tag{1.3}$$

$$x_{jk} \in \{0, 1\} \quad \forall j, k \tag{1.4}$$

$$\sum_{l \in L_{ij}} b_{ij} \leq R_l \quad \forall i \neq j \tag{1.5}$$

For our network model (i.e., each link set contains two links in a *node-switch-node* way), constraint (1.5) can be represented by the following two constraints[4]. Namely, the available bandwidth for the output and input links should not be larger than the bandwidth of the physical port:

[3]In our presentation, we denote an individual partitioned data chunk at each node as a *chunk* and a group of data chunks with a same hash value as a *partition*. For example, in Figure 1(b), the 1^3 is a data chunk on the Node 0, and the group of 1^3 and 1^6 is a partition of the input data.

[4]Recall again that our model can be easily extended to complex network conditions (e.g., routing) by adding parameters to these two constraints.

$$\sum_{j=1}^{n} b_{ij} \le R_l \quad \forall i \neq j$$

$$\sum_{i=1}^{n} b_{ij} \le R_l \quad \forall j \neq i$$

Moreover, based on constraint (1.2), we know that the rate of each flow b_{ij} is directly proportional to its volume v_{ij}, i.e., $b_{ij} = \alpha \cdot v_{ij}$. Based on this, the optimization problem (1) can be converted into:

$$maximize \;\; \alpha \tag{2}$$

subject to:

$$\sum_{j=1}^{n} \alpha \sum_{k=1}^{p} h_{ik} x_{jk} \le R_l \quad \forall i \neq j \tag{2.1}$$

$$\sum_{i=1}^{n} \alpha \sum_{k=1}^{p} h_{ik} x_{jk} \le R_l \quad \forall j \neq i \tag{2.2}$$

$$(1.3) \quad and \quad (1.4)$$

In fact, $\alpha = 1/t$, the larger α, the more bandwidth is obtained by flows in the coflow, and the smaller the completion time will be. Moreover, because R_l is a constant for a given network, if we set $T = R_l/\alpha$ (i.e., $R_l \cdot t$), from the basis of the programming model (2), we will obtain:

$$minimize \;\; T \tag{3}$$

subject to:

$$\sum_{j=1}^{n} \sum_{k=1}^{p} h_{ik} x_{jk} \le T \quad \forall i \neq j \tag{3.1}$$

$$\sum_{i=1}^{n} \sum_{k=1}^{p} h_{ik} x_{jk} \le T \quad \forall j \neq i \tag{3.2}$$

$$(1.3) \quad and \quad (1.4)$$

It is hard to solve the optimization problem (1) directly, since the programming is not only nonlinear, but also has binary integer variables. Regardless, as the model (3) shows, our optimization problem is able to be transformed into a mixed integer linear programming (MILP) problem. There, we only have the binary integer variables (note that each h will be a constant for a given data partitioning method). Consequently, we can easily get an optimal solution of our co-optimization problem by using an optimizer (e.g., Gurobi[5]).

B. Implementation in Large Systems

A system containing hundreds or thousands of computing nodes becomes common in modern data centers. This would make the optimizer-based implementation not suitable in a real data center environment from a practical angle. The reason is that the optimization problem is an integer multi-commodity flow problem [14], of which the computational complexity is NP-complete. From a theoretical perspective, the problem solving time is exponential time, and thus the scheduling process could bring in a heavy overhead in an analytical job: (1) an analytical job always contains multiple distributed operators; and (2) for each operator, when the number of nodes n and the number of data partitions p are large, the problem instances will get too large to be solved in a timely manner. Actually, in our initial tests, we find that the overhead indeed can not be ignored in some large-scale scenarios. For example, for a single join execution, with a configuration with 500 nodes and 7500 partitions, the Gurobi optimizer takes more than half an hour to get the final optimal solution on a commodity machine. To reduce such overhead and get an approximately optimal solution quickly, we propose an efficient heuristic as following.

Implementation. Based on the model (3), our target is to get the destination node for each data partition and guarantee that T is minimized. To reduce the value of T as much as we can, we use a step-by-step strategy to examine the destination of each partition sequentially, and keep that T is minimal in each step. From the constraint (3.1) and (3.2), it can be observed that the value of T would be more sensitive on large data chunks (i.e., with a greater h) rather than small ones. Therefore, we first sort the size of data chunks in a descending order and ensure that partitions including large data chunks are processed with higher priority than the ones with small chunks.

The details of our implementation is shown in Algorithm 1. We start our searching process after the sorting as we have described. For each partition k (line 3), we track

Algorithm 1 Heuristic implementation of CCF

```
    Input: data and system information $n$, $p$, $h_{ik}$, $R_l$
    Output: values for decision variables $x_{jk}$
 1: Collect the information of $h$ of each partition and group
    them into $p$ groups $G$ based there hash values, then sort
    $G$ based on the $max\{h\}$ in each $G_k$
 2: Initialize $x_{jk} = 0 \;\; \forall j, k$
 3: for $k = 1, 2, ..., p$ do
 4:     for $d = 1, 2, ..., n$ do
 5:         Set $x_{dk} = 1$, and $x_{\bar{d}k} = 0 \;\; \forall \bar{d} \neq d$
 6:         Based on all the values of $x_{jk}$,
              compute each $C_i$ ($\forall i \neq j$) based on (3.1),
              compute each $C_j$ ($\forall j \neq i$) based on (3.2)
 7:         Get $T_d = max\{C_i, C_j\} \; \forall i, j$
 8:     end for
 9:     Get the $d$ with $min\{T_d\}$ achieved, and reset $x_{dk} = 1$
        and $x_{\bar{d}k} = 0 \;\; \forall \bar{d} \neq d$
10: end for
```

[5]www.gurobi.com

the cost of C_i and C_j for all possible destinations (i.e., in total n possibilities as line 4). In this process, there are $C_i = \sum_{j=1}^{n} \sum_{k=1}^{p} h_{ik} x_{jk}$ and $C_j = \sum_{i=1}^{n} \sum_{k=1}^{p} h_{ik} x_{jk}$ based on the constraint (3.1) and (3.2). For each potential destination d, the value of current T (i.e., T_d) will be determined by the maximum value of the computed C_i and C_j (line 7), because of $T \geq C$. After comparing all the values T_d for the n potential destinations, we will choose the node as the destination node with the minimal T_d achieved (line 9), to guarantee that current T is the minimal one. Then, the value of x_{jk} will be updated for the computation of following partitions, and the whole searching process will be terminated until all the p partitions have been examined.

C. Skew Handling

Data skew occurs naturally in big data applications [5], [15], and transfer skewed data will bring in heavy network traffic and result in load imbalncing. Therefore, it is very important for practical data systems to perform efficiently in such contexts [5]. To further improve the CCT for the above proposed implementations, here, we focus on how to extend our co-optimization model to handle the skew issue.

To data, large number of techniques have been proposed to handle data skew in join executions [5], [6], [11], [16]. Among them, we have chosen a very efficient method, *partial duplication* [11], in our implementations. Its core idea is: large number of skewed tuples in an input relation are kept locally and not transferred at all, instead, just a very small number of non-skewed tuples from another relation are broadcast to all other nodes. Within such a scheme, the constraints in our optimization model (1) is extended as:

$$\begin{cases} v_{ij} = v'_{ij} + \sum_{k=1}^{p} h'_{ik} x_{jk} & \forall i \neq j \\ (1.2),\ (1.3),\ (1.4) \quad and \quad (1.5) \end{cases}$$

where v'_{ij} means flow volume generated by the broadcast behavior, and this information will be considered as an initial status of each flow f_{ij}. Moreover, h'_{ik} means the size of each data chunk, excluding the tuples, the destinations of which have been assigned in the partial duplication process (keep locally is considered as a local move).

The above skew handling method focuses on the processing of skewed data, therefore, we can treat it as a pre-processing in our approach. Based on its output, we can use the same implementation as described in the Algorithm 1 to schedule the rest data and then get the final execution plan. It is obvious that extra operations in the skew handling process, such as *skew detection*, will bring in extra overheads to our join executions. Nevertheless, various efficient approaches have been proposed to solve this issue, and various results in real big data applications have shown that the overhead can be ignored, compared to the performance improvement it brings [17]. Therefore, we will not consider its detailed overheads in our following evaluations.

IV. Evaluation

In this section, we evaluate the performance of our CCF through a set of simulation-based experiments.

A. Experimental Framework

We compare the network communication time of join executions of our approach with the following two schemes:

- Baseline: the most commonly used *hash-based* approach [11] (referred to as Hash), in which, after the hash partitioning, each data chunk is assigned to a node based on its responsible hash value.
- Minimize network traffic: for each data partition, we examine all the possible destinations and choose the one that can minimize the network traffic. We refer this method as *Mini*. This strategy has been adopted in various advanced join approaches (e.g., track-join [4][6]).

As previously described, the CCF approach, which is based on the solving of linear programming problem, will be not suitable for large-scale systems, due to its overhead. Therefore, we just use our heuristic implementation (referred to as CCF) here. Moreover, current join techniques in the data management domain (e.g., [11], [4]) seldom consider underlying network communications, for the fare of comparison, we have used an optimal coflow scheduling to optimize the data communications for the Hash and Mini approach. Additionally, since the skew handling method we have described can also efficiently reduce network traffic when the input is skewed, we have integrated it into the Mini implementation. In such scenarios, Hash represents the methods focusing on network-level optimization, while Mini represents the approaches focusing on both application and network level optimization, but in a *decoupled* way. In contrast, our approach adopts a *co-optimization* way.

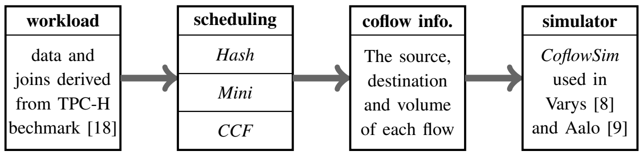

workload
data and joins derived from TPC-H bechmark [18]
scheduling
Hash
Mini
CCF
coflow info.
The source, destination and volume of each flow
simulator
CoflowSim used in Varys [8] and Aalo [9]

Figure 4. Experimental approach used in the evaluation.

1) Methdology: In general, our experimental approach is shown as Figure 4. Namely, for a given workload including a dataset and a join, detailed data flow information generated by different schedule techniques will be delivered to a coflow system to measure the network communication time. In our evaluations, we have chosen the state-of-the-art *CoflowSim*[7]

[6]It should be noticed that the track-join [4] focuses on exploring the join relationships between input relations and minimize network traffic in a *per-key* level. Our approach can be also extended to that level, regardless, this will be beyond the scope of this paper.

[7]https://github.com/coflow/coflowsim

as our back-end simulation system, which is also used in Varys [8] and Aalo [9]. Moreover, we have used the widely used TPC-H benchmark [18] to generate test data, and used the following join in our experiments.

```
select *
from CUSTOMER C join ORDER O
on C.CUSTKEY = O.CUSTKEY
```

2) Datasets: The scaling factor of TPC-H is set to 600. The number of tuples in the two input relations is 90 millions and 900 millions correspondingly. In the meantime, the payload in each tuple is set to 1000 Bytes[8], leading to around 1TB input size. The generated data is evenly distributed, resulting in that the size of data chunks in each partition is very closed to each other. To evaluate our approach in more complex conditions, we make the difference of the size more obvious: for each partition, we let the size of included data chunks follow the Zipfian distribution over the n nodes (referred to as *zipf*). Moreover, as data skew (referred to as *skew*) is quit common in join executions, in order to control the skewness in our tests, similar to current work [11], [19], we randomly choose a portion of data and change their `custkey` to a specified value. For example, we randomly choose 20% of the tuples and set their key to 1, which will make the skewness to 20%. In this way, we can easily identify on-going experiments and capture the essence of a skew distribution.

3) Setup: We have compared the communication performance of Hash, Mini and our CCF. For the original CCF approach, we have used the Gurobi version 6.5.2 with C++ to solve the optimization problem. All the schedule approaches and simulations are implemented on a commodity machine with two 4-core Intel Xeon CPU E5430 processors running at 2.66 GHz and 32GB of RAM. The operating system is Linux kernel version 3.13.0-91 with gcc version 4.8.4. There are two parameters for our test datasets: we set the *zipf* to 0.8 and the *skew* to 20% as default. Moreover, without loss of generality, we just use a simple hash function $f(k) = k \ mod \ p$ to partition data tuples. Because of increasing the value of p would let us have a more fine-grained control on data assignment, we have set p to a value which is 15 times the number of used nodes in each test. Additionally, we use the default configuration of *CoflowSim* in our experiments.

B. Experimental Results

1) Over number of nodes: We compare the performance of Hash, Mini and CCF by varying the number of nodes, from 100 to 1000, over the default data. The results of the network traffic and network communication time are presented in Figure 5. As demonstrated in Figure 5(a), the Mini approach transfers the least amount of data over networks, which is consistent with our expectation, because it focuses on reducing network traffic in join executions. Moreover, we can see that CCF has less network traffic than Hash. The reason could be that Hash just simply redistributes all the data chunks, while CCF could be able to explore part of data locality based on the constraints in our optimization model.

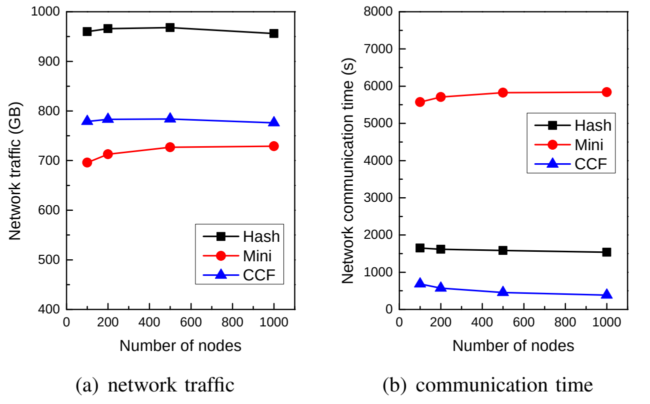
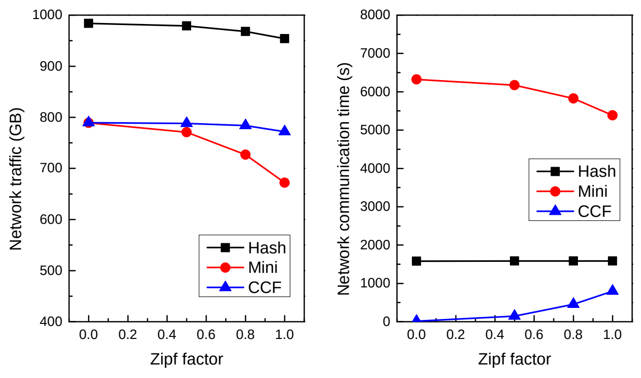


Figure 5. Performance comparison of the three approaches by varying the number of nodes (*zipf*=0.8, *skew*=20%).

Looking at the communication time in Figure 5(b), it can be seen that both Hash and CCF always perform much faster than Mini by varying the number of computing nodes, i.e., less than 2000 secs versus more than 5000 secs. The main reason is that the network traffic can be spread out over all the nodes in the former two approaches, and thus the network bandwidth can be efficiently utilized. In contrast, Mini focuses on transferring data chunks to a node with the largest size, for each given partition. In our case here, following the Zipf distribution, the first node always holds the largest data chunk for each partition, which means that all the data will be flushed to the first node for Mini, and this leads to longer communication time because of the network congestion. Based on these results, we can see that focusing on application-level optimization only could sometimes result in a very poor join performance. In comparison, combining the optimization at both the application- and network-level, our CCF can perform much faster than the other two techniques. Here, we have achieved a speedup of $8.1 - 15.2\times$ over Mini and $2.1 - 3.7\times$ over Hash, which highlights the performance advantages of our framework in various node configurations.

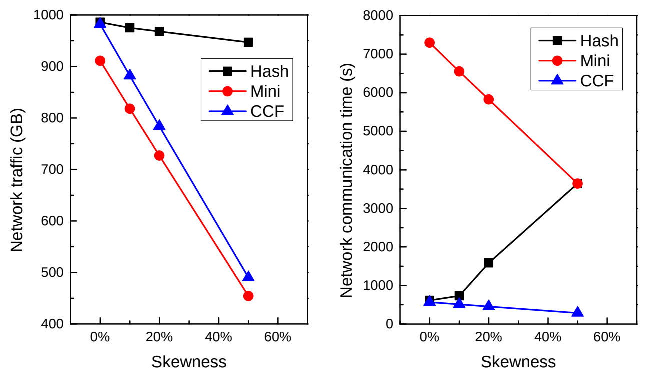
2) With different Zipf factors: We examine the efficiency of each algorithm over 500 nodes with increasing the parameter *zipf* from 0 to 1. As shown in Figure 6(a), similar to above results, Mini still transfers less data than the other two approaches. In the meantime, the network traffics of the three approaches are decreasing with increasing the value of the Zipf factor. The reason is that the large data chunks

[8]This is just for the simplification of our tests, as we can then easily control the size of the dataset. Moreover, we can also easily get the volume of a data flow by counting the number of tuples in it.



(a) network traffic (b) communication time

Figure 6. Performance comparison of the three approaches by varying the Zipf factor (500 nodes, *skew*=20%).



(a) network traffic (b) communication time

Figure 7. Performance comparison of the three approaches by varying the data skewness (500 nodes, *zipf*=0.8).

become even larger with the increment of *zipf*, and the data locality of the three approaches is consequently increased. Moreover, it can be seen that the network traffic of Mini decreases more sharply than others. This is because that all the largest chunks are kept locally in Mini but not in Hash and CCF. In terms of network communication time, as shown in Figure 6(b), it can be observed again that the Mini approach performs the worse in all the cases, though its time cost is decreasing with increasing the Zipf factor. Moreover, the communication time of Hash is nearly constant and CCF increases with increasing the factor. The reason could be that the Hash approach redistributes all the data chunks in an even way. For CCF, it is possible that the power of our co-optimization becomes weak when some data chunks are huge (because transferring such chunks could start to dominate the final communication time). Regardless, we can see that our approach is still obviously faster than Hash and Mini, with a speedup of $6.7 - 395\times$ over Mini and $1.9 - 98.7\times$ over Hash in all the cases.

3) Over various skews: We evaluate the performance of the three approaches over 500 nodes with various skews, increasing from 0 to 50%. The results are presented in Figure 7. As illustrated in Figure 7(a), the network traffic of Mini and CCF decreases linearly with increasing the skew while Hash only decreases slightly. The reason is that the skew handling technique we have adopted in Mini and CCF can efficiently reduce network traffic. For Hash, the data locality of a specified node increases (i.e., the node with hash value equals to 1), and consequently the size of whole transferred data is decreased. Moreover, Mini has less network traffic than the other two approaches, demonstrating its ability on minimizing network traffic once again. For the communication time shown in Figure 7(b), Hash increases sharply with increasing the skewness while Mini and CCF decreases in a linear way. This is reasonable, because the data skew will bring in network hotspots in Hash [5]. For Mini and CCF, the skewed tuples are simply kept locally, namely, the network resources originally assigned to these tuples are used to transfer other tuples, and thus the communication time can be reduced. Again, we can see that CCF always outperforms the Hash and Mini approach, with a speedup of $12.8\times$ over Mini and $1.1 - 12.8\times$ over Hash. More specifically, even when the skewness is 0, namely the whole dataset is uniform distributed, CCF is still about 50 secs faster than Hash.

4) Brief summary: From above results, it can be seen that in various conditions, although optimization techniques on application-level data movement can efficiently reduce or even minimize network traffic, their network communication performance could be still very poor in large-scale distributed systems, such as the Mini approach we have studied here. On the other hand, scheduling approaches focusing on improving underlying network communications only, without any considerations of application-level optimization, could also lead to a sub-optimal performance, e.g., the Hash we have evaluated here. In contrast, the CCF proposes the novel idea of co-optimization, which bridges the gap of schedules at the application and network levels, and thus it can always perform faster on network communications than current techniques.

V. Related Work

As the most essential tasks in large-scale environments, distributed operations such as joins and aggregations can incur significant time costs on network communications and hence improving their execution efficiency would have a significant impact on the overall performance of current big data analytics. In fact, these operations have been extensively studied in the field of data management, and large number of methods have been proposed to improve their performance. For example, for distributed join executions, current research focuses on the challenge on how to efficiently move data, either in the presence of different join workloads (e.g., skew) or different computing platforms (e.g., clusters and Cloud) or both [4], [5], [6], [11], [16], [20]. Their main target is either to reduce network traffic or to improve load-balancing or both, so as to balance computations and improve network communication time.

Although all the solutions have been shown to be very efficient, few of them has ever considered the impacts of underlying networks. Namely, their designs are concentrated on application workloads and computing platforms, but are totally independent from networks. However, as we have shown in this work, these approaches could result in poor performance in large systems, even in the condition that an efficient communication schedule approach (i.e., coflow) has been employed. Moreover, although recent work [7], [19] has ever considered the network communication problem on data redistribution in join executions, they only focus on application-level controls of network communications, which is different from our framework that implements co-optimization schedulers based on a data flow abstraction.

To realize high performance data analytics, a lot of techniques have been proposed in various domains in recent years. For example, the scheduling of I/O [21] and computing resources [22] in data centers, the optimization of VM migrations [23] and system parameter turning [24] in Cloud, or even the designs of scheduling and security strategies [25], [26]. These approaches can indeed speed up data processing. However, because of agnostic on the existence of coflows, they would not be able to achieve the best or even could be harmful to the application-level performance [13]. Current research on coflow scheduling aims at minimizing the average CCT cost and meeting coflow deadlines [8]. Advanced approaches and implementations, such as online conditions [9], with complex networks [10], have been proposed for practical environments. All of these methods are based on the assumption the source and destination of each data flow have been known. Namely, their designs are totally isolated from the application-level optimization. As we have demonstrated in this paper, this design will bring in a sub-optimal or poor performance. Compared to this, our CCF can always achieve a better performance. From another aspect, our framework is based on the coflow model, therefore, more advanced techniques on coflow scheduling (e.g., routing [10], [13]) will be able to be integrated in our framework, and consequently enhance its application in practical data center environments.

In this era of software-defined networking (SDN [27]), the entire network of a system becomes visible and programmable, which gives us the great opportunities for its application on big data applications. It is true that SDN has been applying to the domain of big data management in these two years. For example, the work [28] presents a method that be able to adaptively select an optimal query plan based on the information provided by the network before a query execution. However, these techniques just focus on using SDN to move data in a distributed way. Namely, at each specified time point, they just move data from a node to another node, but not like the problem we have studied in this work, in which the data from different nodes moves in a parallel way. We should know that parallel cases (e.g., coflows) are very common in large-scale data applications, since data sets from different resources are always loaded in computation nodes in a parallel way so that the loaded data can undergo further downstream processing (e.g., for analytics) as quickly as possible. Additionally, though some related problems such as network path selection [29] are being studied using SDN, none of them has ever considered the coflow cases.

To the best of our knowledge, this is the first work to analyze the co-optimization opportunities of coflow scheduling for distributed data operator optimization. We believe that various big data applications will benefit from our designs, e.g., organizations can make business decisions in time to boost their sales, and governments can faster their response to disasters, etc. It is also our hope that this will open up a rich area of research and technology development for the large scale data-analytics community.

VI. Conclusion and Future Work

In this paper, we propose a novel Coflow-based Co-optimization Framework (CCF), which targets for speeding up big data analytics in large-scale distributed systems. We have presented the detailed implementation of the CCF, and conducted a detailed performance evaluation using large-scale simulations. Our experimental results show that the proposed CCF can perform faster than current approaches on network communications under various workloads.

Our future work lies in extending our framework model to more complex workloads (e.g., analytical queries) and more complex computing environments (e.g., InfiniBand supported HPC Cloud [30]). Our long term goal is to develop a high-performance data analytics system which is always highly efficient and robust in the presence of different workloads and network configurations in large-scale distributed scenarios.

References

[1] W. Li, H. Liu, P. Yang, and W. Xie, "Supporting regularized logistic regression privately and efficiently," *PloS one*, vol. 11, no. 6, p. e0156479, 2016.

[2] W. Xie, M. Kantarcioglu, W. S. Bush, D. Crawford, J. C. Denny, R. Heatherly, and B. A. Malin, "SecureMA: protecting participant privacy in genetic association meta-analysis," *Bioinformatics*, vol. 30, no. 23, pp. 3334–3341, 2014.

[3] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "VL2: a scalable and flexible data center network," *Communications of the ACM*, vol. 54, no. 3, pp. 95–104, 2011.

[4] O. Polychroniou, R. Sen, and K. A. Ross, "Track join: distributed joins with minimal network traffic," in *SIGMOD*, 2014, pp. 1483–1494.

[5] L. Cheng, S. Kotoulas, T. Ward, and G. Theodoropoulos, “Robust and skew-resistant parallel joins in shared-nothing systems,” in *CIKM*, 2014, pp. 1399–1408.

[6] N. Bruno, Y. Kwon, and M.-C. Wu, “Advanced join strategies for large-scale distributed computation,” *PVLDB*, vol. 7, no. 13, pp. 1484–1495, 2014.

[7] W. Rödiger, T. Muhlbauer, P. Unterbrunner, A. Reiser, A. Kemper, and T. Neumann, “Locality-sensitive operators for parallel main-memory database clusters,” in *ICDE*, 2014, pp. 592–603.

[8] M. Chowdhury, Y. Zhong, and I. Stoica, “Efficient coflow scheduling with Varys,” in *SIGCOMM*, 2014, pp. 443–454.

[9] M. Chowdhury and I. Stoica, “Efficient coflow scheduling without prior knowledge,” in *SIGCOMM*, 2015, pp. 393–406.

[10] Y. Zhao, K. Chen, W. Bai, M. Yu, C. Tian, Y. Geng, Y. Zhang, D. Li, and S. Wang, “RAPIER: Integrating routing and scheduling for coflow-aware data center networks,” in *INFOCOM*, 2015, pp. 424–432.

[11] Y. Xu, P. Kostamaa, X. Zhou, and L. Chen, “Handling data skew in parallel joins in shared-nothing systems,” in *SIGMOD*, 2008, pp. 1043–1052.

[12] M. Chowdhury and I. Stoica, “Coflow: A networking abstraction for cluster applications,” in *HotNets*, 2012, pp. 31–36.

[13] Y. Li, S. H.-C. Jiang, H. Tan, C. Zhang, G. Chen, J. Zhou, and F. C. Lau, “Efficient online coflow routing and scheduling,” in *MobiHoc*, 2016, pp. 161–170.

[14] S. Even, A. Itai, and A. Shamir, “On the complexity of time table and multi-commodity flow problems,” in *FOCS*, 1975, pp. 184–193.

[15] L. Cheng, A. Malik, S. Kotoulas, T. E. Ward, and G. Theodoropoulos, “Fast compression of large semantic web data using X10,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 9, pp. 2603–2617, 2016.

[16] L. Cheng and S. Kotoulas, “Efficient skew handling for outer joins in a cloud computing environment,” *IEEE Transactions on Cloud Computing*, in press.

[17] S. Kotoulas, J. Urbani, P. Boncz, and P. Mika, “Robust runtime optimization and skew-resistant execution of analytical SPARQL queries on Pig,” in *ISWC*, 2012, pp. 247–262.

[18] Transaction Processing Performance Council, “TPC-H benchmark specification,” *available at http://www.tpc.org/tpch/*, 2016.

[19] L. Cheng and T. Li, “Efficient data redistribution to speedup big data analytics in large systems,” in *HiPC*, 2016, pp. 91–100.

[20] L. Cheng, I. Tachmazidis, S. Kotoulas, and G. Antoniou, “Design and evaluation of small-large outer joins in cloud computing environments,” *Journal of Parallel and Distributed Computing*, 2017.

[21] Z. Yang, J. Tai, J. Bhimani, J. Wang, N. Mi, and B. Sheng, “GReM: Dynamic SSD resource allocation in virtualized storage systems with heterogeneous IO workloads,” in *IPCCC*, 2016, pp. 1–8.

[22] H. Gao, Z. Yang, J. Bhimani, T. Wang, J. Wang, N. Mi, and B. Sheng, “AutoPath: Harnessing parallel execution paths for efficient resource allocation in multi-stage big data frameworks,” in *ICCCN*, 2017.

[23] F. Zhang, X. Fu, and R. Yahyapour, “LayerMover: Storage migration of virtual machine across data centers based on three-layer image structure,” in *MASCOTS*, 2016, pp. 400–405.

[24] J. Wang, Y. Yao, Y. Mao, B. Sheng, and N. Mi, “Fresh: Fair and efficient slot configuration and scheduling for Hadoop clusters,” in *CLOUD*, 2014, pp. 761–768.

[25] C. Hao, J. Shen, C. Chen, H. Zhang, Y. Wu, and M. Li, “PC-Ssampler: Sample-based, private-state cluster scheduling,” in *CCGrid*, 2017, pp. 599–608.

[26] Y. Chen, Z. Wang, D. Whalley, and L. Lu, “Remix: On-demand live randomization,” in *CODASPY*, 2016, pp. 50–61.

[27] D. Kreutz, F. M. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, “Software-defined networking: A comprehensive survey,” *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[28] P. Xiong, H. Hacigumus, and J. F. Naughton, “A software-defined networking based approach for performance management of analytical queries on distributed data stores,” in *SIGMOD*, 2014, pp. 955–966.

[29] M. V. Neves, C. A. De Rose, K. Katrinis, and H. Franke, “Pythia: Faster big data in motion through predictive software-defined network optimization at runtime,” in *IPDPS*, 2014, pp. 82–90.

[30] J. Zhang, X. Lu, M. Arnold, and D. K. Panda, “MVAPICH2 over openstack with SR-IOV: an efficient approach to build HPC clouds,” in *CCGrid*, 2015, pp. 71–80.