Wide-Area Spark Streaming: Automated Routing and Batch Sizing

Wenxin Li, Di Niu, *Member, IEEE,* Yinan Liu, Shuhao Liu, and Baochun Li, *Fellow, IEEE*

Abstract—Modern stream processing frameworks, such as Spark Streaming, are designed to support a wide variety of stream processing applications, such as real-time data analytics in social networks. As the volume of data to be processed increases rapidly, there is a pressing need for processing them across multiple geo-distributed datacenters. However, these frameworks are not designed to take limited and varying inter-datacenter bandwidth into account, leading to longer query latencies. In this paper, we present the design and implementation of an extended Spark Streaming framework to automatically and optimally schedule tasks, select data flow routes and determine micro-batch sizes across geo-distributed datacenters in wide-area networks. To make these decisions, we propose a sparsity-regularized ADMM algorithm to efficiently solve a nonconvex optimization problem, based on readily measurable operating traces. Toward incremental real-world deployment, we take a non-intrusive approach to support flexible routing of micro-batches by adding a new DStream transformation we have developed to the existing Spark Streaming framework. As a result, our implementation can enforce scheduling decisions by modifying application workflows only. We have deployed our implementation on Amazon EC2 with emulated bandwidth constraints, and our experimental results on various types of queries have demonstrated the effectiveness of our proposed framework, as compared to the existing Spark Streaming scheduler and other data-locality-based heuristics.

Index Terms—Wide-area Networks, Spark Streaming, Routing and Batch Sizing.

1 Introduction

Many types of big data streaming analytics are being generated, computed and aggregated over the wide area. A social network, such as Facebook and Twitter, may need to detect popular keywords in minutes. A search engine may wish to obtain the number of clicks on a recommended URL once every few seconds. A cloud service operator may wish to monitor system logs in its distributed datacenters to detect failures in seconds. In all these streaming analytics, log-like data are generated from all over the world and are collected at local nodes, datacenters or points of presence (PoP) first, before being aggregated to a central site to repeatedly answer a standing query.

Spark Streaming [1] is an extension of Spark that enables and simplifies the processing of streaming analytics using micro-batches of data. Spark is built on the concept of Resilient Distributed Datasets (RDDs) [2], where an RDD is a *batch* of input data. Similarly, Spark Streaming relies on the concept of discretized streams (DStreams) for data abstraction. A DStream is a continuous sequence of RDDs arriving at different time steps, where each RDD contains a one-time slice of data in the stream. The length of the time slice is referred to as the *batch size* [3]. Spark Streaming performs a transformation on a DStream by applying the same transformation on each RDD in the DStream. For example, an operator may wish to compute the word counts in a document stream once every five seconds, where the batch size is 5 seconds. In other words, Spark Streaming is based on a "micro-batch" architecture, where streaming computation is carried out as a continuous series of MapReduce operations on the micro-batches.

While recognizing the significance of Spark Streaming, one may want to directly apply the Spark Streaming framework to perform streaming analytics over wide-area networks. For example, one can first use Spark Streaming framework to launch a map task at each of the local nodes, so as to perform partial computation over the streams of data. After that, a reduce task can be launched at the central location, which is responsible for collecting intermediate results from all the local nodes, performing the remaining computation, and displaying querying results to analytics users [4].

Needless to say, in this paradigm, resources are typically limited at the wide-area network that may have highly constrained and variable bandwidth [5]–[7]. Despite this, Spark Streaming is not specifically designed to take into account the significant bandwidth variation on wide area network (WAN) links. Directly transferring all the collected data from a source to its central collecting site may not always be the best choice, if the link bandwidth between them is limited. In fact, since Spark Streaming needs to process all the micro-batches generated at the same timestamp together, even a bottleneck link at a single data source can significantly slow down the overall query response rate. With delayed responses, the operator may lose the key opportunity to make decisions based on the query.

In this paper, we extend state-of-the-art batched stream processing systems, typically represented by Spark Streaming, to operate in wide-area networks, jointly considering bandwidth-aware batch sizing, task scheduling and routing of data streams. Note that simple data locality mechanisms, such as performing reduction or local aggregation at data sources, are insufficient as a solution in WAN. In order to increase the query response rate

- *W. Li is with the Department of Electrical and Computer Engineering, University of Toronto and the School of Computer Science and Technology, Dalian University of Technology. E-mail: liwenxin@mail.dlut.edu.cn.*
- *D. Niu is with the Department of Electrical and Computer Engineering, University of Alberta, Edmonton, Alberta, T6G 2V4, Canada. Email: dniu@ualberta.ca.*
- *Y. Liu, S. Liu and B. Li are with the Department of Electrical and Computer Engineering, University of Toronto, Ontario, M5S 3G4, Canada. Email: {yinan, shuhao} @ece.utoronto.ca, bli@ece.toronto.ca.*

and meet batch processing deadlines, the system must address the following challenges: (1) selecting a fast path to route the DStream from each source to its collecting site and avoid bottleneck links, where the fastest path is not necessarily the direct route but might be a detoured path with higher bandwidth; (2) determining the minimum batch size for each query, which may rely on data generated from multiple input sources, when multiple queries coexist; and (3) placing reducer tasks at proper intermediate nodes to perform data reduction and to support flexible detoured routing. The complexity of these challenges comes from their coupled nature: both the achievable batch sizes and reducer placement will depend on data routing decisions, while routing decisions, in turn, depend on the required batch sizes.

To tackle these challenges, we have made two original contributions in this paper:

First, we have formulated the problem of joint batch sizing, task placement and routing as a *nonconvex optimization* problem, and solve it with the alternating direction method of multipliers (ADMM) [8], an optimization framework that has recently gained widespread popularity in machine learning and big data applications. Through an innovative problem reformulation, we propose an ADMM algorithm to jointly learn the efficient data aggregation paths for each query together with its optimal batch sizing based on readily available runtime traces, including link bandwidth and the relationship between input and output sizes in the application of interest. We have also introduced a sparsity regularizer into our ADMM algorithm to yield sparse route selection for each query and to avoid traffic splitting.

Second, we have designed a routing functionality for Spark Streaming to support flexible routing and task placement decisions, including transfers with a detour route. In our implementation, we adopted a non-intrusive approach by adding a new transformation of DStreams to Spark Streaming. It allows explicit DStream migration between any two nodes in a geo-distributed cluster. The new transformation can carry out any computed routing and task scheduling decisions by only modifying application workflows; in the meantime, it can still leverage load balancing and data locality mechanisms implemented by the task scheduler in the original Spark.

We have implemented the proposed Wide-Area Spark Streaming framework, and performed an extensive evaluation of this framework through real-world experiments conducted on a cluster of Amazon EC2 instances, with pre-specified bandwidth capacities between different instances to emulate a bandwidth-limited environment in wide area networks. We have also used container-based virtualization and Docker Swarms to ensure the concurrent execution of tasks from coexisting Spark Streaming jobs in the shared emulated environment. By running coexisting streaming queries of different types (including `WordCount`, `Grep` and `Top-k`) based on a real-world dataset, we show that through data-driven machine intelligence, our extended wide-area Spark Streaming framework leads to a significantly higher query rate (i.e., lower batch size) while reducing the processing latency—especially the tail latency—of each micro-batch. It also improves network transfer times as compared to the original Spark Streaming framework that adopts heuristic scheduling optimization.

2 Problem Formulation and Procedure

We first motivate the importance of intelligent data routing and task scheduling for wide-area streaming using a simple example in Fig. 1. In this example, a streaming query needs to compute certain statistics based on the micro-batches generated on nodes A, B and C, and displays the computed result at node D once every 1 second, i.e., the batch size is 1 second. With its data locality mechanism, as shown in Fig. 1 (a), Spark Streaming will first perform local computation to generate the intermediate result on each of the input nodes A, B and C. The intermediate results are then sent to node D for reduction into the final result. Suppose the size of intermediate data per batch after local computation is 5 Mb on each input node. Obviously, with the bottleneck link A $\rightarrow$ D, the processing time of each micro-batch is at least 2.5 seconds, which is the time for the straggler node A to transfer its intermediate data to node D. This cannot meet the per batch deadline of 1 second.

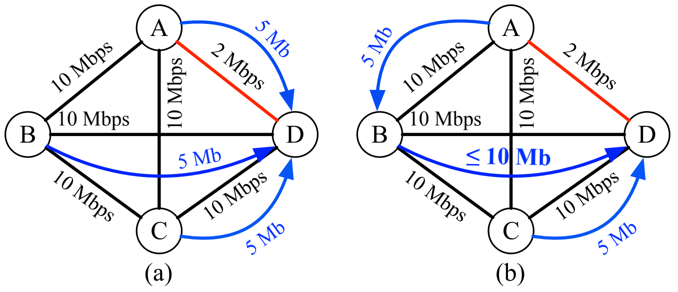


Fig. 1. The topology of a motivating example. Red lines represent bottleneck links.

A better solution, as shown in Fig. 1 (b), is for node A to use the detour path A $\rightarrow$ B $\rightarrow$ D to transfer its intermediate data to avoid the bottleneck link. Moreover, we can further place an additional reduce task in node B to merge the intermediate data from nodes A and B into 10 Mb or less. This scheme will lead to a processing time per micro-batch of less than 1 second, meeting the per batch deadline. This example suggests that Spark Streaming is not always optimal in wide area networks.

In this section, we formulate a joint problem of automatic micro-batch sizing, task placement and routing for multiple concurrent streaming queries on the same wide area network. Such a joint problem will be formulated as a nonconvex optimization problem. Note that task placement may be implicitly decided by the routing path selection. The problem we will formulate is *biconvex* in terms of routing path selection and batch sizing, i.e., given one set of variables fixed, the optimization of the other set is a convex problem. We will subsequently propose an ADMM algorithm to solve the problem.

2.1 System Model and Goodput Maximization

We use a graph $G = (V, E)$ to represent a network of geo-distributed nodes. The number of nodes is denoted as $N = |V|$. Each node may consist of multiple co-located servers and can host multiple Spark workers. Let C_e be the available bandwidth on each link $e \in E$. Suppose there are Q streaming analytics queries on G. Each query i monitors the data generated from a set of source nodes $S_i \subset V$, and collects output at a single destination node $D_i \in V$. Each query i has a *batch interval* τ_i (in terms of seconds), specifying how frequently new data should be processed in query i. The batch interval τ_i corresponds to the *batch size* in Spark Streaming. Important notations used throughout this paper are listed in Table 1.

TABLE 1
Notations and Definitions.

Symbol	Definition
G	the network of geo-distributed nodes
V	the set of nodes in G
E	the set of links in G
C_e	the available bandwidth on link $e \in E$
Q	the total number of streaming analytics queries
$S_i \subset V$	the set of source nodes for query $i \in Q$
$D_i \in V$	the destination node for query $i \in Q$
τ_i	the *batch interval* or *batch size* of query i
τ_i^l	the lower bound for τ_i (i.e., $\tau_i \geq \tau_i^l, \forall i$)
r_{vi}	the data generation rate for query i at source $v \in S_i$
$M_i(\tau_i)$	the output size of query i per batch
$R_i(\tau_i)$	the goodput of query i
J_i	the total number of aggregation trees from sources S_i to D_i
T_{ij}	the j-th aggregation tree in $\{T_{i1}, T_{i2}, \cdots, T_{iJ_i}\}$
x_{ij}	an indicator variable that represents whether $T_{i,j}$ is selected

We denote the output size of query i per batch at the collecting node D_i as $M_i(\tau_i)$ which is a function of the batch interval τ_i. Then, given the data generation rate r_{vi} at each source $v \in S_i$ of query i, the amount of output per batch $M_i(\tau_i)$ is a function of the amount of total input data generated in this batch, i.e.,

$$M_i(\tau_i) = U_i\Bigg(\sum_{v \in S_i} r_{vi}\tau_i\Bigg),$$

where U_i is a function that can be learned from the data characteristics of each particular application, which we will illustrate in Sec. 2.2. Also, we will show that U_i is often a linear function based on some real-world data, although our algorithm does not rely on the linearity assumption.

We can then define the *goodput* of query i given τ_i as

$$R_i(\tau_i) := M_i(\tau_i)/\tau_i,$$

which is the rate at which useful information is collected by query i at the collecting node. Assume the batch size τ_i of each query i has a lower bound τ_i^l.

Now consider a particular query i with source nodes $S_i \subset V$ and destination node $D_i \in V$. For each source $v \in S_i$, it is easy to enumerate all the paths, denoted by the set P_{vi}, from v to D_i. Choosing one path for each source and combining them over all the sources in query i will lead to a tree from the sources S_i to D_i. For query i, we denote all these feasible trees as $T_{i1}, \ldots, T_{iJ_i}$, where each T_{ij} is an *aggregation tree* from sources S_i to D_i. We only consider trees up to two hops to limit the propagation delay and variable space. By considering trees instead of paths, we can perform data aggregation (e.g., `ReduceByKey`, `Union`, `Join`, etc.) at an intermediate node if two paths $p_{v_1,i}$ and $p_{v_2,i}$ of two source nodes v_1 and v_2 in query i shares a link.

We then need to derive the data rate incurred on link e due to each selected aggregation tree, in order to avoid violating the bandwidth constraints. Similar to $M_i(\tau_i)$ defined above, let $M_{ij}^e(\tau_i)$ denote the amount of data transmitted on link e for query i if tree T_{ij} is selected. Just like $M_i(\tau_i)$, $M_{ij}^e(\tau_i)$ can also be learned as a function of τ_i, i.e.,

$$M_{ij}^e(\tau_i) = U_i\Bigg(\textstyle\sum_{v \in S_i \bigcap \text{Descendants of } e \text{ in } T_{ij}} r_{vi}\tau_i\Bigg).$$

Similar to $R_i(\tau_i)$, if tree T_{ij} is selected, the data rate on link e due to query i is given by

$$R_{ij}^e(\tau_i) := M_{ij}^e(\tau_i)/\tau_i.$$

Our objective is to jointly select the optimal aggregation tree as well as the optimal batch size τ_i for each query i, to maximize the total goodput $\sum_{i=1}^{Q} R_i(\tau_i)$. Let $x_{ij} \in \{0,1\}$ represent tree selections, where $x_{ij} = 1$ indicates that tree T_{ij} is selected and $x_{ij} = 0$ indicates otherwise. Then, our problem is to find $\{x_{ij}\}$ and $\{\tau_i\}$ by solving the following problem:

$$\underset{\{x_{ij}\},\{\tau_i\}}{\text{maximize}} \quad \sum_{i=1}^{Q} R_i(\tau_i) \tag{1}$$

$$\text{subject to} \quad \sum_{i=1}^{Q} \sum_{j: e \in T_{ij}} R_{ij}^e(\tau_i) \cdot x_{ij} \leq C_e, \quad \forall e \in E, \tag{2}$$

$$\sum_{j=1}^{k_i} x_{ij} = 1, \quad i = 1, \ldots, Q, \tag{3}$$

$$x_{ij} \in \{0,1\}, \quad j = 1, \ldots, J_i, \ i = 1, \ldots, Q, \tag{4}$$

$$\tau_i \geq \tau_i^l, \quad i = 1, \ldots, Q, \tag{5}$$

where constraints (3) and (4) require that *only one tree* should be chosen for each query, while constraint (2) ensures that the total data rate on link e will not exceed the link capacity C_e. Since $R_{ij}^e(\cdot)$ and $R_i(\cdot)$ can be learned offline, they serve as predetermined *basis functions* in problem (1).

Problem (1) is a hard *non-convex* problem, since 1) x_{ij} is a binary integer, and 2) there is a multiplication between $R_{ij}^e(\tau_i)$ and x_{ij} that leads to non-convexity. In Sec. 3, we will propose an efficient ADMM algorithm to solve Problem 1, through an innovative reformulation to decouple path selection and batch sizing. We also adapt the ADMM algorithm to incorporate an iteratively reweighted sparsity regularizer to ensure sparse tree selection for each query.

2.2 Learning the Basis Functions

As has been shown in Sec. 2.1, the basis functions $R_i(\cdot)$ and $R_{ij}^e(\cdot)$ depend on the input-output relationship U_i in a particular application as well as data generation rates r_{vi}, which can be monitored at each input source v.

For a variety of typical queries, the input-output relationship U actually demonstrates an approximately linear and stable shape over different input sizes [9], and thus can be readily profiled in advance. For example, `WordCount` calculates the count of each distinct word in documents, where the number of distinct words linearly increases with the arrival of input documents. This is true at least in the regime of micro-batches, as we will verify by profiling real Wikipedia data. `Grep` finds all the input strings that match a particular pattern, e.g., all the lines containing a specific word in system logs. Again, more lines will match the pattern as more input logs become available. The `Top-`k `Count` gathers the counts of the most popular k elements (e.g., URL, keywords), and thus the output remains constant as the input data size increases.

In fact, in Facebook's big data analytics cluster, the ratio of intermediate to input data sizes is 0.55 for a median query, with 24% of queries even having this ratio greater than 1 [10]. Since Spark Streaming is based on micro-batches, it exhibits a similar input-output scaling pattern, although in the regime of small batch sizes.

The particular U can be learned for each application either based on benchmarking data or adaptively from past data during runtime. Here we specifically characterize the input-output relationship U of `WordCount`, which is a widely used benchmarking

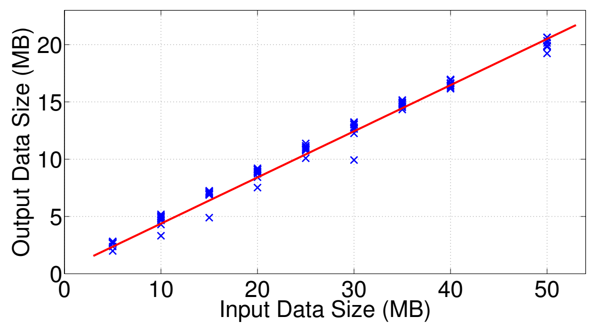


Fig. 2. The input-output relationship of `WordCount` based on 12 GB of Wikipedia data. Linear regression reveals the fit of the model for the output size $U(I) = 0.40I + 0.43$ as a function of the input size I.

query for big data platforms, based on a publicly available 12 GB text dataset from Wikipedia [11]. To focus on the micro-batch regime, we split the Wikipedia dataset into small chunks of different sizes ranging from 5–50 MB, and randomly choose 10 chunks of each size to serve as the inputs of `WordCount` in Spark. We perform a linear regression for the input-output relationship and show it in Fig. 2. We can observe that the output size is approximately $U(I) = 0.40I + 0.43$ for an input size I between 5 MB and 50 MB, a range compatible to micro-batches in streaming analytics.

If U_i is linear, $M_i(\tau_i)$ is linear in τ_i, and $R_i(\tau_i) := M_i(\tau_i)/\tau_i$ will be a linear function of the *query frequency* $1/\tau_i$. Similarly, in this case $R^e_{ij}(\tau_i)$ is linear in $1/\tau_i$. Hence, in the rest of the paper, we may abuse notation and use $R_i(\frac{1}{\tau_i})$ to equivalently represent $R_i(\tau_i)$ and $R^e_{ij}(\frac{1}{\tau_i})$ to represent $R^e_{ij}(\tau_i)$.

3 An ADMM Algorithm for Joint Tree Selection and Batch Sizing

In this section, we propose an ADMM algorithm to solve the joint batch sizing and sparse tree selection problem. Instead of solving problem (1) with integer constraints, we relax x_{ij} to be a fractional number and solve the following problem penalized by a sparsity regularizer:

$$\begin{aligned}
\underset{\{x_{ij}\},\{\frac{1}{\tau_i}\}}{\text{maximize}} \quad & \sum_{i=1}^{Q} R_i\left(\frac{1}{\tau_i}\right) - \lambda \sum_{i=1}^{Q}\sum_{j=1}^{J_i} \mathbb{1}(x_{ij} > 0) && (6)\\
\text{subject to} \quad & \sum_{i=1}^{Q}\sum_{j:e\in T_{ij}} R^e_{ij}\left(\frac{1}{\tau_i}\right)\cdot x_{ij} \le C_e, \quad \forall e \in E, && (7)\\
& \sum_{j=1}^{k_i} x_{ij} = 1,\ \forall i, \quad 0 \le x_{ij} \le 1,\ \forall (i,j),\\
& 0 < \frac{1}{\tau_i} \le \frac{1}{\tau_i^l},\ i = 1,\dots,Q,
\end{aligned}$$

where $\mathbb{1}(x_{ij} > 0)$ is an indicator function defined as

$$\mathbb{1}(x > 0) = \begin{cases} 1, & \text{if } x > 0,\\ 0, & \text{otherwise.}\end{cases} \qquad (8)$$

The regularizing term $\sum_{i=1}^{Q}\sum_{j=1}^{J_i} \mathbb{1}(x_{ij} > 0)$ is a penalty to yield sparse tree selections by pushing most x_{ij} to zero.

Note that Problem (6) is still a non-convex problem involving an integer penalty term $\sum_{i=1}^{Q}\sum_{j=1}^{J_i} \mathbb{1}(x_{ij} > 0)$ and non-convex multiplicative terms in constraint (7). In the following, we first propose an ADMM algorithm to solve Problem (6) without the sparsity penalty. Then, to handle the sparsity penalty, we propose an iterative reweighting procedure to solve the linearization of a smooth surrogate of the sparsity penalty in each iteration, and merge this procedure into the proposed ADMM algorithm to yield sparse tree selections $\{x_{ij}\}$.

3.1 An ADMM Algorithm to Decouple Non-Convex Constraints

The rationales why ADMM is selected are three-fold. First, ADMM is exactly a method for addressing problems in *bi-convex* form, while problem (6) without the sparsity penalty is a *bi-convex* problem, i.e., convex for tree selections given fixed batch sizes, and convex for batch sizing given all the selected trees. Therefore, through an innovative reformulation, we can decouple this non-convex problem (6) into alternated minimizations of two quadratic programs (QPs), for tree selection and batch sizing, respectively. Second, ADMM has superior empirical performance in solving large-scale optimization problems. Third, ADMM has been shown as a theoretically grounded method for *bi-convex* problems [8]. In the following, we first provide a brief primer on ADMM.

A general bi-convex problem that ADMM solves is of the form

$$\begin{aligned}
\text{minimize} \quad & F(x, z)\\
\text{subject to} \quad & G(x, z) = 0,
\end{aligned} \qquad (9)$$

where $F : \mathbb{R}^n \times R^m \to \mathbb{R}$ is *bi-convex*, i.e., convex in x for a given z and convex in z for a given x, and $G : \mathbb{R}^n \times R^m \to \mathbb{R}^p$ is *bi-affine*, i.e., affine in x given z, and affine in z given x. This problem can be solved by the scaled form of ADMM with the following iterative updates [8]:

$$x^{k+1} := \arg\min_x \left(F(x, z^k) + \frac{\rho}{2}\|G(x, z^k) + u^k\|_2^2 \right) \qquad (10)$$

$$z^{k+1} := \arg\min_z \left(F(x^{k+1}, z) + \frac{\rho}{2}\|G(x^{k+1}, z) + u^k\|_2^2 \right) \qquad (11)$$

$$u^{k+1} := u^k + G(x^{k+1}, z^{k+1}). \qquad (12)$$

We now reformulate problem (6) without the sparsity penalty into the general bi-convex problem above. Introducing the auxiliary variable z_{ije} to decouple constraint (7), problem (6) without the sparsity penalty is equivalent to the following problem:

$$\begin{aligned}
\underset{\{x_{ij}\},\{\frac{1}{\tau_i}\},\{z_{ije}\}}{\text{maximize}} \quad & \sum_{i=1}^{Q} R_i\left(\frac{1}{\tau_i}\right) && (13)\\
\text{subject to } & R^e_{ij}\left(\frac{1}{\tau_i}\right)\cdot x_{ij} - z_{ije} = 0,\ \forall (i,j), \forall e \in E,\\
& \sum_{i=1}^{Q}\sum_{j:e\in T_{ij}} z_{ije} \le C_e,\ \forall e \in E,\\
& \sum_{j=1}^{k_i} x_{ij} = 1,\ \forall i, \quad 0 \le x_{ij} \le 1,\ \forall (i,j),\\
& 0 < \frac{1}{\tau_i} \le \frac{1}{\tau_i^l},\ \forall i,
\end{aligned}$$

which is equivalent to

$$\begin{aligned}\underset{\{x_{ij}\},\{\frac{1}{\tau_i}\},\{z_{ije}\}}{\text{maximize}}\quad & \sum_{i=1}^{Q} R_i\left(\frac{1}{\tau_i}\right) + h(\{1/\tau_i\}) + g\big(\{x_{ij}\},\{z_{ije}\}\big)\\ \text{subject to}\quad & R_{ij}^{e}\left(\frac{1}{\tau_i}\right)\cdot x_{ij} - z_{ije} = 0,\ \forall (i,j), \forall e\in E,\end{aligned}$$

where $g(\{x_{ij}\},\{z_{ije}\})$ is an indicator function of all the *uncoupled* constraints of $\{x_{ij}\},\{z_{ije}\}$ in problem (13), i.e., it is zero if the second and third constraints of problem (13) are all satisfied and is negative infinity otherwise. Similarly, $h(\{\frac{1}{\tau_i}\})$ is zero if $0 < \frac{1}{\tau_i} \le \frac{1}{\tau_i^l}$, $\forall i$, and is negative infinity otherwise.

Now we have converted problem (13) into the same form as the general bi-convex problem above with a bi-affine constraint, where all $\{x_{ij}\}$ and $\{z_{ije}\}$ are treated as one set of variables and $\{\frac{1}{\tau_i}\}$ is another set of variables. Then we can solve problem (13) with ADMM, as described in Algorithm 1, which updates the two sets of variables alternately. Apparently, steps 4 and 5 of Algorithm 1 only involve quadratic programming (QP), which can be efficiently solved by a number of existing QP solvers such as interior point methods.

Algorithm 1 An ADMM algorithm for Problem (6) without the sparsity penalty.

```
1: Input: Basis functions $\{R_i(\cdot)\}$, $\{R_{ij}^e(\cdot)\}$; link capacities $\{C_e\}$.
2: Output: $\{x_{ij}^k\}$, $\{\frac{1}{\tau_i^k}\}$ when the algorithm stops.
3: $k := 0$. Initialize $\{x_{ij}^0\}$, $\{z_{ije}^0\}$, $\{u_{ije}^0\}$.
4: Solve the following subproblem to obtain $\{x_{ij}^{k+1}\}$, $\{z_{ije}^{k+1}\}$
     $$\begin{aligned}\underset{\{x_{ij}\},\{z_{ije}\}}{\text{minimize}}\quad & \sum_{(i,j,e)}\left(R_{ij}^{e}\left(\frac{1}{\tau_i^k}\right)x_{ij} - z_{ije} + u_{ije}^k\right)^2 \qquad (14)\\ \text{subject to}\quad & \sum_{i=1}^{Q}\sum_{j:e\in T_{ij}} z_{ije} \le C_e,\ \forall e\in E,\\ & \sum_{j=1}^{k_i} x_{ij} = 1,\ \forall i,\quad 0\le x_{ij}\le 1,\ \forall (i,j)\end{aligned}$$
5: Solve the following subproblem to obtain $\{\frac{1}{\tau_i^{k+1}}\}$:
     $$\begin{aligned}\underset{\{\frac{1}{\tau_i}:0<\frac{1}{\tau_i}\le\frac{1}{\tau_i^l}\}}{\text{minimize}}\quad & -\sum_{i=1}^{Q} R_i\left(\frac{1}{\tau_i}\right) \qquad (15)\\ & +\frac{\rho}{2}\sum_{(i,j,e)}\left(R_{ij}^{e}\left(\frac{1}{\tau_i}\right)x_{ij}^{k+1} - z_{ije}^{k+1} + u_{ije}^k\right)^2\end{aligned}$$
6: Update $u_{ije}^{k+1}$ by
     $$u_{ije}^{k+1} := u_{ije}^k + R_{ij}^e\left(\frac{1}{\tau_i^{k+1}}\right)x_{ij}^{k+1} - z_{ije}^{k+1}. \qquad (16)$$
7: $k := k+1$, repeat until the stop criterion is met.
```

3.2 Generating Sparse Tree Selection

When the sparsity penalty in problem (6) is present, the derivation above for the ADMM algorithm still holds except that we need to add the sparsity penalty $\lambda\sum_{i=1}^{Q}\sum_{j=1}^{J_i}\mathbb{1}(x_{ij} > 0)$ to the objective function in subproblem (14). Then, subproblem (14) becomes a typical ℓ_0-norm regularized least squares problem, which can be efficiently solved by LASSO [12] in statistical learning, which replaces the ℓ_0-norm sparsity penalty by ℓ_1 norm, i.e., a linear term $\lambda\sum_{i=1}^{Q}\sum_{j=1}^{J_i} x_{ij}$.

It has been shown [13] that reweighted ℓ_1 minimization can further enhance the sparsity of the solutions. To solve subproblem (14), we should iteratively penalize its objective by $\lambda\sum_{i=1}^{Q}\sum_{j=1}^{J_i} x_{ij}/(x_{ij}^{t-1}+\delta)$ in each iteration t. Such an algorithm can achieve fast convergence, minimizing the linearization of a concave log-sum penalty iteratively [14].

To further speed up convergence, we merge the iterations of reweighted ℓ_1 minimization with those of ADMM, leading to a revised ADMM algorithm. This algorithm is the same as Algorithm 1, except that in step 4, we replace the objective function (14) by

$$\sum_{(i,j,e)}\left(R_{ij}^{e}\left(\frac{1}{\tau_i^k}\right)x_{ij} - z_{ije} + u_{ije}^k\right)^2 + \lambda\sum_{i=1}^{Q}\sum_{j=1}^{J_i}\frac{x_{ij}}{x_{ij}^k+\delta},$$

which places a weight $1/(x_{ij}^k+\delta)$ for each x_{ij} when solving for x_{ij}^{k+1}. Intuitively speaking, the smaller the x_{ij}^k, the larger the weight, and the smaller the produced x_{ij}^{k+1}, leading to a sparse solution. After the procedure above is done, most queries will select only a single tree. If a query i still has multiple nonzero x_{ij}, we simply choose the tree with the largest x_{ij}. In Sec. 6, we show that the algorithm can effectively select the most appropriate aggregation tree for each query.

We now highlight several merits of our algorithm. First, our algorithm can characterize the intertwined relationship between batch sizing and tree selection, while aiming to increase the goodput. More specifically, in each iteration, a smaller batch size will lead to a tree selection that has more available bandwidth on its links, whereas a tree with less bandwidth will force the system to choose a larger batch size. The root reason is that the smaller the batch size, the more frequently a query is executed, which generates more network flows. Hence, higher bandwidth is needed to accommodate this query. Second, our algorithm has the ability of self-regulation: if the tree on which a query is routed has less bandwidth, our algorithm will reduce the query goodput (by increasing its batch size) to make sure the query can be supported by the underlying network. Third, the proposed ADMM algorithm can quickly converge within only 3-4 iterations (which we will show in Section 5) and in the meantime, is amenable to parallelization. This makes our algorithm more suitable for large-scale problems.

4 Implementation

In this section, we describe our implementation of an extended Spark Streaming framework to ensure that the streaming data are routed and aggregated according to the aggregation tree selected by our algorithm in each query. In the original Spark Streaming, although the generator and consumer of a DStream can be written in the application itself, the routing of DStream is completely decided by the `TaskScheduler`, a component that is responsible for task placement.

For example, consider a streaming `WordCount` query generated at node A with the output collected at node B. The streaming job has two computation tasks, including a map and a reduce. For data locality, it is the best for the original Spark `TaskScheduler` to place the map operation at node A and place the reduce task

```
val sparkConf = new SparkConf() \
                    .setAppName("word count application") \
                    .set("spark.scheduler.mode", "FAIR") \
                    .set("spark.streaming.concurrentJobs","2")
val sc = new SparkContext(sparkConf)
val ssc = new StreamingContext(sc, Seconds(1))

val Node_A: List[String] = List("10.6.3.4")
val Node_B: List[String] = List("10.6.3.6")
val result = ssc.socketTextStream(Node_A(0), 6688, StorageLevel.MEMORY_AND_DISK_SER) \
                    .flatMap(_.split(" ")).map(word => (word, 1)) \
                    .transferTo(Node_B.toSeq) \
                    .reduceByKey() \
                    .print()
ssc.start()
ssc.awaitTermination()
```

Fig. 3. Example code for a `WordCount` streaming query.

at node B. However, the current Spark Streaming cannot support detoured routes or relays, e.g., transferring the intermediate map result of node A to node B via node C, which may exist in our computed two-hop aggregation trees for the purpose of avoiding bottleneck links.

One intuitive approach is to revise the default Spark `TaskScheduler`. However, this approach is undesirable, since 1) the required modification will be too intrusive and non-incremental to the existing Spark framework; 2) simplify modifying task scheduling still cannot enable detoured routing of DStreams, since there is no "relaying" operation in native Spark.

Alternatively, we take a non-intrusive approach which is amenable to incremental deployment, without modifying the existing internal behavior of Spark Streaming. In our implementation, we first realize the routing function with a new transformation, which we call `addRoute()`, as an additional method on DStream, the data stream abstraction in Spark Streaming. We then enforce the scheduling decisions, including detoured routing and aggregation at intermediate nodes, by adding `addRoute()` together with other reduce tasks to the application workflow.

Overview. From a high level, `addRoute()` can explicitly route the DStream to a specified worker node, adding relay tasks whenever necessary. It takes a single parameter, which specifies the hostname to which the DStream should be routed. Similar to other Spark Streaming transformations, `addRoute()` returns a new instance of DStream, representing the data stream that has been received by the destination worker.

Usage. The usage of `addRoute()` can be explained with `WordCount` example mentioned at the beginning of this section. To route the intermediate DStream generated at node A to node B via node C, when writing the application, we can simply add a `addRoute(‘‘C’’)` transformation between the `map()` and `reduce()` transformations in the application program. Thus, the results from `map()` will first be sent to node C and the final `reduce()` transformation will be applied onto the DStream that has already been transferred to C by `addRoute()`.

With `addRoute()` as a fundamental building block, we can easily enforce the aggregation tree selected by our optimization algorithm together with the implicit task placement decisions in Spark Streaming. For example, the optimization results can be fed to the `DAGScheduler`, the application workflow analyzer of Spark Streaming. The `DAGScheduler` is fully aware of all transformations applied onto the data stream, and will automatically insert `addRoute()` to the original application workflow as additional transformations, realizing the desired paths for DStream flows.

Since `addRoute()` is implemented as an additional transformation to be inserted to the application flow of a given Spark Streaming program of interest, our implementation is non-intrusive and is fully compatible to the original Spark Streaming framework. In other words, all existing data locality mechanisms, task scheduling optimizations, customizations and code patches in Spark Streaming can still be leveraged.

Implementation of `addRoute()`. In Spark Streaming, the micro-batches in a DStream are processed separately and continuously, by a series of computation tasks. Each task has an important attribute, `preferredLocations`, which specifies its placement preferences with a list of candidate worker nodes. `addRoute()` leverages this mechanism to explicitly specify the route for a data stream, by modifying the `preferredLocations` of the subsequent task. While these tasks are being scheduled, the `TaskScheduler` will attempt to satisfy the preferences, by assigning a task to one of its available candidate worker nodes.

Our implementation is based on Spark 1.6.1, and the detailed implementation of `addRoute()` is in the RDD class. This class contains many transformations, such as `groupby` and `reduceby`. In this class, we have added a new transformation—`TransferTo`. With this transformation, users can explicitly specify a relay node by modifying application code only. When users specify such relay node, `TransferTo` will launch a task on this node by invoking `addRoute()` function. This task mainly receives the map output, encapsulates them into a new RDD—`transferredRDD`, and forwards the `transferredRDD` to reduce task.

Example Code. As an example, Fig. 3 shows the code for a `WordCount` streaming query. This query performs `WordCount` statistics based on the micro-batches generated on node A, and displays the result at node C every one second, using the detoured path A→B→C. Specifically, a `StreamingContext` is created with a batch interval of 1 second. Using this context, the source DStream that represents streaming data from a TCP source (node A) is generated. The `flatmap` (one-to-many transformation) is used to split each record in the source DStream by space characters into words, such that a streaming of words could be generated. Such words DStream is further mapped to a DStream of `(word, 1)` pairs, via the `map` function (one-to-one transformation). The DStream of `(word, 1)` pairs is then transferred to the destination node C via the intermediate node B, through the `transferTo` transformation. The `transferTo` transformation will invoke the `addRoute("B")` to add node B for transferring data from node A to node C. The `reduceByKey` is then performed on the DStream of `(word, 1)` pairs to get the frequency of words, and finally `print()` will print the results at the destination node. It should be noted that the `WordCount` streaming application should be submitted at node C, such that node C can act as the destination.

5 Simulation

We first conduct simulations to verify the effectiveness of our proposed ADMM-based algorithm. We simulate a WAN network with 7 datacenters (please refer to Fig. 5 for the details of this network). Specifically, we randomly generate around 30 streaming queries, and inject these queries into the network. Similarly, each query has two randomly generated input sources, and the input data generation rate for each input source is a random value in the range of $[1, 5]$ Mbps.

We compare the following two algorithms with the proposed ADMM-based algorithm. The first one is the optimization algorithm used in our previous work [15], denoted as "Heuristic Iteration". Specifically, in each iteration, this algorithm heuristically minimizes the batch sizes when the path selection is fixed and leaves more residual bandwidth by adjusting path selection

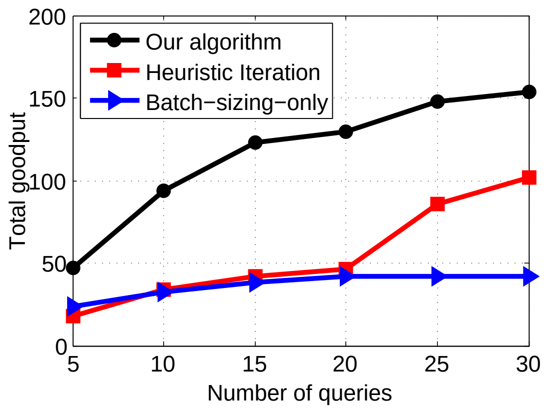


Fig. 4. The total goodput achieved by our algorithm and the comparison method, under different number of streaming queries.

when the batch sizes are fixed. The second comparison algorithm (denoted as "Batch-sizing-only") optimizes the batch sizes for all queries but the involved data flows are all routed with direct paths.

Goodput is important for users that have streaming querying demands, as it measures the number of useful information bits delivered by the network to a certain user per unit of time. Therefore, we record the total goodput across all queries, i.e., $\sum_{i=1}^{Q} R_i(\tau_i)$. By varying the number of streaming queries from 5 to 30, we plot the total goodput for different methods in Fig. 4. It is clear that for both our algorithm and the Heuristic Iteration, the total goodput increases as the number of queries grows. For Batch-sizing-only, the total goodput increases until the number of queries grows to 20 and remains the same when the number of queries is larger than 20. The reason is that in our simulation, Batch-sizing-only can only support up to 20 queries and will crash when the number of queries is larger than 20. We can further observe that the goodput achieved by our algorithm is always higher than that achieved by both Heuristic Iteration and Batch-sizing-only algorithms, irrespective of the number of queries. To be more specific, compared to the Heuristic Iteration, our algorithm can improve the total goodput by up to 66.12%, and the average improvement on the goodput can be 55.34%. On the other hand, the total goodput can be improved by up to 67.4%, when comparing our algorithm with Batch-sizing-only. Note that when there are less than 20 queries, the total goodput achieved by Batch-sizing-only is a little bit smaller than that achieved by Heuristic Iteration. This is because that Batch-sizing-only algorithm ignores an indispensable component — routing, making it less efficient for maximizing the total goodput of the queries.

We now investigate the convergence and running time of our algorithm. By varying the number of streaming queries from 5 to 30, Table 2 shows the number of iterations needed to achieve convergence, under our algorithm, the Heuristic Iteration algorithm and the Batch-sizing-only algorithm. Note that, for each algorithm, the convergence is achieved when the maximum number of iterations (i.e., 100) is reached or the gap of goodput between two adjacent iterations is less than 0.0001. We can clearly see that for both the Heuristic Iteration and Batch-sizing-only algorithms, the number of iterations increases significantly with the increasing of the number of streaming queries. Note that when the number of queries is larger than 20, Batch-sizing-only will make the system crash. We can further observe that our algorithm is extremely fast and always converges within 3-4 iterations, irrespective of the number of streaming queries. This demonstrates the fast convergence of our algorithm compared to conventional heuristics. We finally study the running time of our algorithm. We observe that one iteration takes on average 13.0753 seconds on a Dual-Core Intel(R)Core(TM)i5-2430M 2.40GHz CPU. As such, if we have a computing cluster with tens of servers [16] and run our algorithm in parallel on this cluster, each iteration takes around 0.13 seconds. Thus, it takes less than one second to solve the problem and to obtain the batch sizing and routing decisions for all queries, which demonstrates the efficiency of our algorithm in real systems. One may wonder how the system will react when tiny tasks [17] exist or the batch size is less than 1 second, e.g., 50-100 ms. According to the study [18], Spark Streaming will crash when the batch size is less than 100 ms. However, it is worth noting that our optimization techniques can support as small a batch size as the original Spark Streaming can support—the routing and batch sizing decisions can be reused across micro-batches as long as the workload profiles are slowly changing. In fact, the workload characteristics are largely static and can change at a much slower rate for a large number of typical streaming jobs [18]–[20].

TABLE 2
The number of iterations that our algorithm and the Heuristic Iteration need to achieve convergence.

Num. of iterations \ Methods / Num. of queries	Our algorithm	Heuristic Iteration	Batch-sizing-only
5	3	2	2
10	3	3	4
15	4	6	5
20	3	10	7
25	3	12	⊘
30	4	19	⊘

6 Performance Evaluation

Now, we conducted real-world experiments to evaluate our extended Spark Streaming framework in an emulated bandwidth-constrained environment on a cluster of Amazon EC2 instances. We compare the proposed Wide Area Spark Streaming with original Spark Streaming with a data locality mechanism when adopted in WAN.

6.1 Experimental Setup

Testbed Setup: We build a testbed of 7 compute instances in the US East region on Amazon EC2 with controllable inter-instance bandwidth constraints, as shown in Fig. 5. Since the cost of storage and processors is decreasing at a faster pace than that of provisioning bandwidth [21], [22], bandwidth is more likely to become the bottleneck in wide area streaming analytics. We therefore use `c3.8xlarge` instances to emulate adequate compute and memory resources. Each `c3.8xlarge` has 32 vCPUs and 64 GB memory. To emulate the bandwidth constraints that would exist in wide area networks, we leverage Linux Traffic Control to limit the link bandwidth between compute instances. Detailed bandwidth connections are shown in Fig. 5, where the link bandwidth is chosen from 5 to 35 Mbps at random. Even though each compute instance we launched is not as large as a commodity datacenter, we believe that the testbed can faithfully emulate the bandwidth bottlenecks in a wide area network.

Deployment: To emulate a multi-user environment, where each user runs a separate streaming query, we leverage Docker to

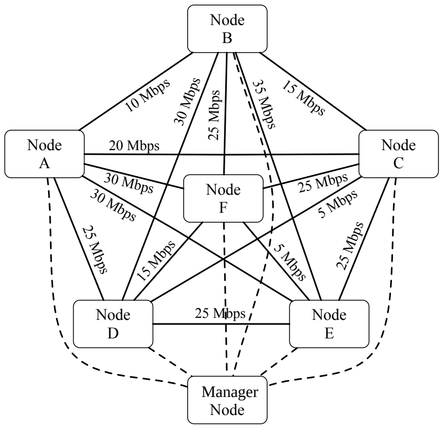


Fig. 5. The WAN emulation testbed launched for streaming experiments.

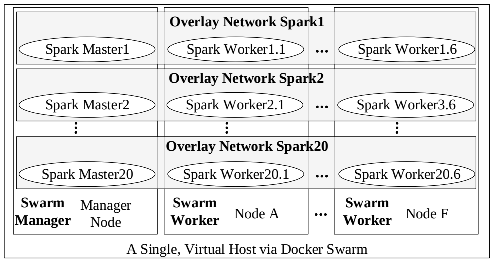


Fig. 6. Deploying wide area streaming on the launched cluster via Docker.

deploy Spark Streaming clusters on the launched testbed. Docker is a widely used technique to automate the deployment of software applications in a lightweight, portable, and self-sufficient way. Specifically, we use Docker Swarm to turn a pool of hosts into a single virtual host, as shown in Fig. 6. Then, we deploy a Spark cluster for each user by launching a set of containers in the virtual host. In such a case, each user can run its respective streaming query independently in its own Spark cluster (consisting of containers located in all 7 instances), while different Spark clusters will run simultaneously and share the network.

In our experiments, we use one compute instance to serve as the manager node, while the other six instances are worker nodes. We deploy 20 Spark clusters, where each Spark cluster has a separate container in each worker node running Spark Worker service. Furthermore, each Spark cluster uses a container in the manager node for running the Spark Master service. For each Spark cluster, there is an overlay network that connects its containers in different nodes.

Workloads: We use 20 recurring `WordCount` queries for testing, as `WordCount` is a typical streaming query used in the benchmarking of big data processing systems. For each query, there are two input sources placed on two randomly chosen nodes and there is one randomly chosen output node where the query is answered. In addition, the input data generation rate for each input source is uniformly chosen from 1 to 5 Mbps. The experimental results when using these 20 `WordCount` queries will be shown in Sec. 6.3. We also tested mixed query types, including `WordCount`,`Grep`,`Top-K`. The `Grep` streaming query finds all the lines containing a specific word (i.e., "the"), while the `Top-k` streaming query outputs the most popular 100 words, by number

TABLE 3
Input data generation rates used in our experiment.

	Query 1	Query 2	Query 3	Query 4	Query 5
Input 1	1 Mbps	2 Mbps	5 Mbps	2 Mbps	2 Mbps
Input 2	4 Mbps	2 Mbps	1 Mbps	4 Mbps	1 Mbps
	Query 6	Query 7	Query 8	Query 9	Query 10
Input 1	2 Mbps	4 Mbps	4 Mbps	2 Mbps	2 Mbps
Input 2	4 Mbps	3 Mbps	2 Mbps	1 Mbps	3 Mbps
	Query 11	Query 12	Query 13	Query 14	Query 15
Input 1	2 Mbps	5 Mbps	1 Mbps	2 Mbps	1 Mbps
Input 2	2 Mbps	2 Mbps	4 Mbps	2 Mbps	2 Mbps
	Query 16	Query 17	Query 18	Query 19	Query 20
Input 1	2 Mbps	1 Mbps	5 Mbps	1 Mbps	2 Mbps
Input 2	3 Mbps	2 Mbps	2 Mbps	4 Mbps	4 Mbps

Note: each query has two input sources, and two corresponding input data generation rates.

TABLE 4
Path selection for each input in each query.

Query Index	Wide-area Streaming	Original Streaming	Query Index	Wide-area Streaming	Original Streaming
1	AFC	AC	11	ADE	AE
	BEC	BC		DE	DE
2	ADB	AB	12	AD	AD
	DB	DB		FD	FD
3	CA	CA	13	EBA	EA
	DEA	DA		FA	FA
4	BD	BD	14	BF	BF
	CD	CD		ECF	EF
5	BC	BC	15	AE	AE
	EBC	EC		BE	BE
6	DB	DB	16	AFB	AB
	EDB	EB		FB	FB
7	CAE	CE	17	BEA	BA
	DAE	DE		FEA	FA
8	CFD	CD	18	CBF	CF
	FD	FD		DBF	DF
9	EC	EC	19	AFB	AB
	FAC	FC		CFB	CB
10	DF	DF	20	CF	CF
	EDF	EF		ECF	EF

A, B, C, D, E and F are node indexes in our testbed.

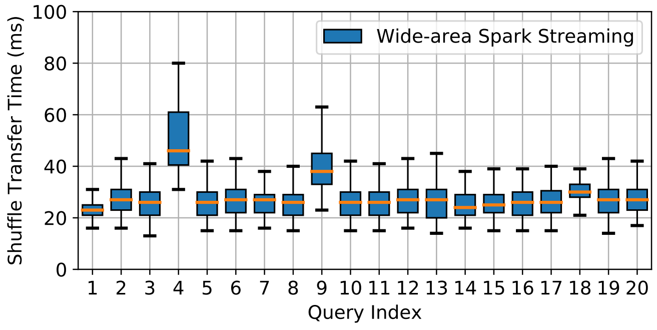
of times. We use 5 queries for each query type, and thus there are 15 queries in total. The experimental results using those mixed queries will be shown in Sec. 6.4.

6.2 Evaluated Methods

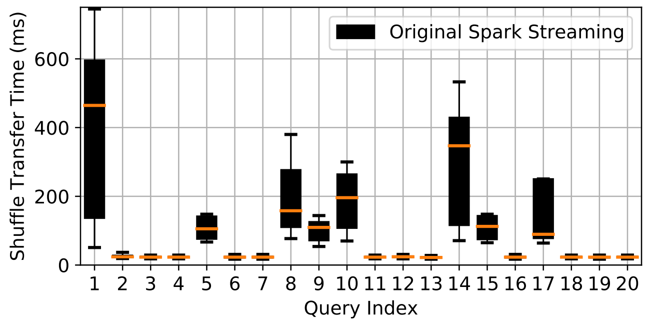
In our experiments, we compare the following two versions of Spark Streaming systems, which represent different methods to make routing and batch sizing decisions:

Wide-area Spark Streaming: This method jointly determines the batch size and routing paths for each query by using the proposed Algorithm 1, and runs the application flow generated by Algorithm 1 with the automatically inserted `addRoute()` transformations. The computation based on Algorithm 1 is extremely fast and always converges within 3-4 iterations. Note that in some rare cases, Algorithm 1 may not produce a sparse solution for a query, i.e., a query i selects two trees. In such a case, we choose the tree with the largest x_{ij}.

Original Spark Streaming: In this method, each source node of a query sends its local output data to the central collecting site directly, where the data is further processed and aggregated to produce final query output. In other words, this method sends



(a) Wide-area Spark Streaming



(b) Original Spark Streaming

Fig. 7. Box plot for the shuffle transfer time per batch for each of the 20 queries.

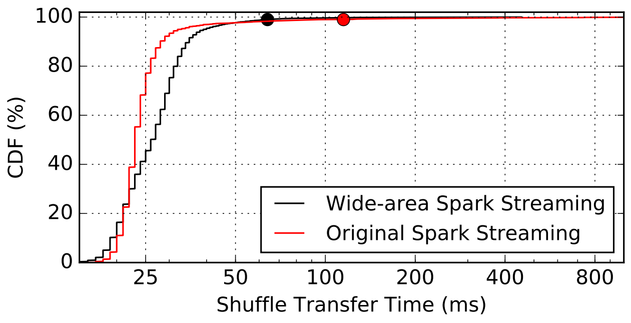


Fig. 8. CDF of the shuffle transfer time per batch in all the queries.

out intermediate data on the direct path between each source-destination pair. With such direct path selection, it can get a convex problem by substituting $x_{i,j}, \forall i, \forall j$ to the problem (1), and accordingly the batch size can be obtained by solving this convex problem. Furthermore, the built-in data locality mechanisms in Spark are used to enforce all the reduce tasks to be placed at the output node. This strategy represents a common industrial practice conforming to data locality.

Apparently, with bandwidth constraints, the computed batch size for Wide-area Spark Streaming is significantly smaller than that for Original Spark Streaming for each query. This implies that Wide-area Spark Streaming, which intelligently selects detours to avoid bottleneck links, can support a potentially higher query rate in this bandwidth-constrained scenario. The selected paths for each query under both Wide-area Spark Streaming and Original Spark Streaming strategies are illustrated in Table 4.

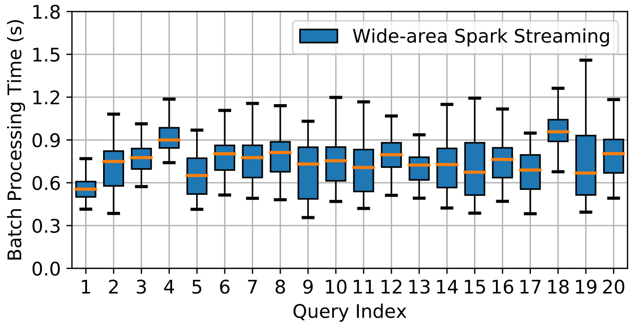
For validation and performance evaluation, we run all queries for a period of 5 minutes under both Wide-area Spark Streaming and Original Spark Streaming strategies to measure the processing time per batch and shuffle transfer time.

6.3 Results of `WordCount` queries

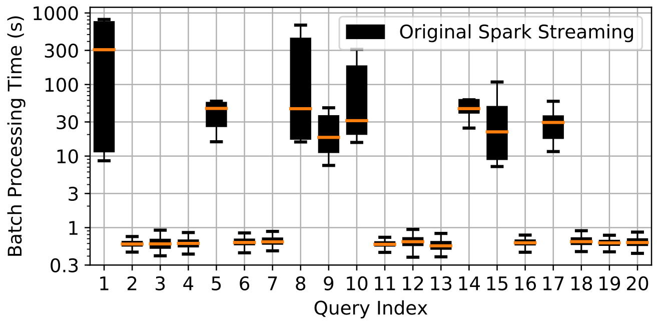
Fig. 7 first depicts the shuffle transfer time for each query achieved by both Wide-area Spark Streaming and Original Spark Streaming Strategies, in the form of box plot. Each bar shows five values of shuffle transfer time for a query: 98th percentile, 75th percentile, 50th percentile, 25th percentile, 2th percentile. It is clear that Wide-area Spark Streaming outperforms Original Spark Streaming in the shuffle transfer time for 8 queries, e.g., Query 1, 5, 8, 9, 10, 14, 15, 17. This is because Original Spark Streaming chooses direct paths for these queries, while the bandwidth on those direct paths is relatively low. For instance, the direct path (i.e., E$\rightarrow$F) that Original Spark Streaming selected for query 10 only has 5 Mbps bandwidth, while our Wide-area Spark Streaming would choose the detoured path (i.e., E$\rightarrow$D$\rightarrow$F) that has bandwidth higher than 5 Mbps. To be more particular, the shuffle transfer time for Original Spark Streaming can be as large as 750 ms (e.g., the 98th percentile value in Query 1), while this value is less than 100 ms for our Wide-area Spark Streaming. This demonstrates the benefits of leveraging detoured paths in wide area streaming analytics, as detoured path have a potentially higher bandwidth than the direct path. We can further observe that Wide-area Spark Streaming achieves a similar value of shuffle transfer time for the remaining queries, compared to Original Spark Streaming. This implies that Wide-area Spark Streaming will not build its superiority on performance loss of any other queries. The key is that Wide-area Spark Streaming uses smaller batch size for each query, leading to higher frequency in sending the input data. By taking a detailed analysis, we observe that the average shuffle time per batch over the 20 queries is only 28 ms for our Wide-area Spark Streaming, while that value is 78 ms for the Original Spark Streaming. This implies that our Wide-area Spark Streaming can reduce the shuffle time per batch by 64.1% on average, compared to the Original Spark Streaming.

To understand on a microscopic level, we also plot the CDF of the shuffle transfer time across all micro-batches and all queries in Fig. 8, under both Wide-area Spark Streaming and Original Spark Streaming. In this figure, one may wonder at this point that, the fraction of shuffles with transfer time less than a same value (e.g., 25 ms) for Wide-area Spark Streaming is unexpectedly smaller than that for Original Spark Streaming. This is actually reasonable because that Original Spark Streaming uses higher batch size, leading to a lower frequency in sending data and eventually it makes less congestion in the wide-area network. However, such an unexpected result does not mean that the Wide-area Spark Streaming is inferior to Original Spark Streaming, as the streaming performance is closely related to the lowest shuffle transfer time. More precisely, Original Spark Streaming is more likely to put a shuffle on a bottleneck link, as it makes no attempt to use the detoured path that may have higher link bandwidth. So, this is why all shuffles under Wide-area Spark Streaming can be completed within a smaller time, compared to Original Spark Streaming. Such an interesting property directly demonstrates that our Wide-area Spark Streaming is more practical in reducing the tail processing latency.

A lower shuffle transfer time may not always lead to a lower batch processing time, as processing a batch of streaming

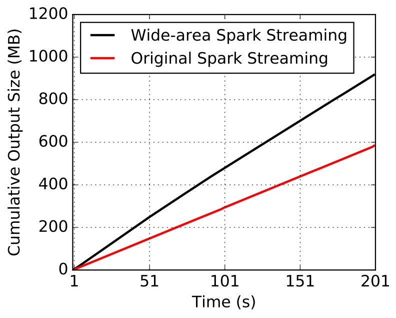


(a) Wide-area Spark Streaming

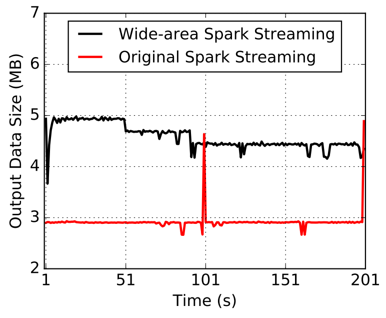


(b) Original Spark Streaming

Fig. 9. Box plot of the processing time per batch for each of the 20 queries.



(a) Cumulative output



(b) Output per second

Fig. 11. The overall system output rates of Wide Area Spark Streaming and original Spark Streaming during a period of 200 seconds.

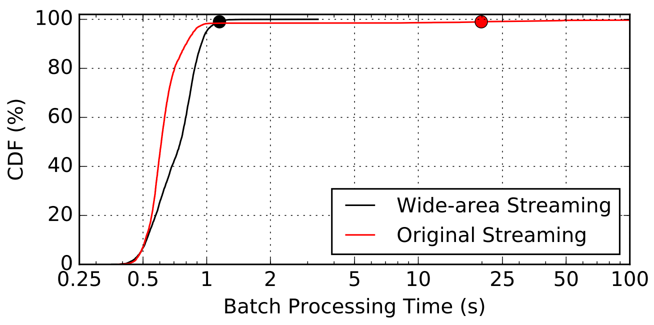
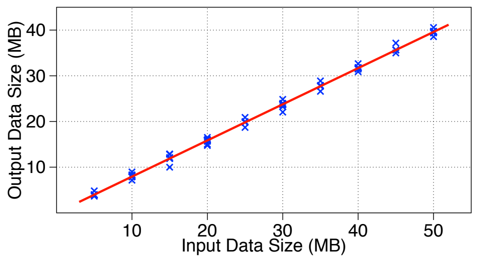


Fig. 10. CDF of the processing time per batch in all the queries.

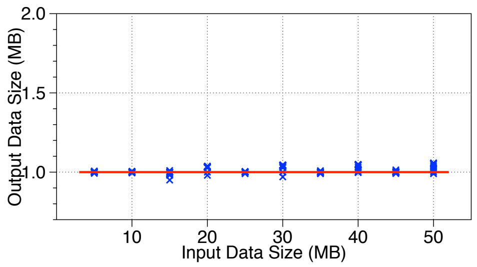
data involves both computation stage and network stage. So, we investigate the batch processing time for each query, as shown in Fig. 9. Similarly, for Query 1, 5, 8, 9, 10, 14, 15 and 17, Wide-area Spark Streaming achieves much smaller batch processing times, as compared to Original Spark Streaming. One of the reasons is still that our Wide-area Spark Streaming will select detoured paths with potentially higher bandwidth. Another reason is that the batch sizes for those queries under Wide-area Spark Streaming are only a few seconds, while the batch sizes under Original Spark Streaming are tens of seconds. One may wonder if the Original Spark Streaming will perform better when the corresponding batch sizes are reduced. The answer is yes. But we cannot reduce the batch sizes of those queries (i.e., Query 1, 5, 8, 9, 10, 14, 15 and 17) under the Original Spark Streaming, since reducing the batch sizes of those queries will negatively affect the performance of other queries significantly, and may even cause the system to crash if the batch sizes are set to too small values [18]. The 98th percentile batch processing time of Original Spark Streaming is nearly up to 1000s (e.g., Query 1), which makes no sense for modern streaming processing system. While the maximal 98th percentile batch processing time of Wide-area Spark Streaming is just over 1s. We can further observe that there is not much difference between the two streaming strategies for the remaining queries, in terms of the batch processing time. Even though there is a little bit difference, it is mainly because that Wide-area Spark Streaming uses smaller batch sizes and accordingly brings more overhead on the computation stage. We can easily check that the results of batch processing time exactly follow the same trends with the shuffle transfer time in Fig. 7, verifying that the link bandwidth is the only bottleneck in wide area streaming processing. This further implies that our launched testbed on EC2 platform is appropriate and solid to benchmarking our Wide-area Spark Streaming. We can further observe that across all the 20 queries, the average processing time per batch is only 0.75s when using our Wide-area Spark Streaming, whereas that value is 46.5s for the Original Spark Streaming, leading to a reduction of 98% on the average processing time per batch.

Again, we plot the CDF of batch processing time across all batches and all queries for both Wide-area Spark Streaming and Original Spark Streaming in Fig. 10. We observe that 99% of batches can be processed faster by using Wide-area Spark Streaming, compared to Original Spark Streaming. This implies that Wide-area Spark Streaming is efficient in reducing tail batch processing time, which has great significance in modern wide area streaming processing systems with high concurrency requirements.

To indicate the goodput, Fig. 11(a) first presents the cumulative output data across all queries during the 5-minute period of time for both Wide-area Spark Streaming and Original Spark



(a) The input-output relationship of Grep application can be fitted by a linear function $U(I) = 0.795I + 0.02$ of the input size I.



(b) The output data size of Top-K application is a constant value, irrespective of the input data size.

Fig. 12. Measuring the input-output relationships for `Grep` and `top-k` streaming queries, based on 12 GB of Wikipedia data.

Streaming, as the query output data size closely relates to the goodput in stream processing system. We can easily check that across all the time, the cumulative output data generated by Wide-area Spark Streaming is always larger than that of Original Spark Streaming. We can further observe that the output data generation rates are 4.6MB/s and 3MB/s for Wide-area Spark Streaming and Original Spark Streaming respectively. Therefore, our Wide-area Spark Streaming can improve the output data generation rate by 34.8%, compared to the Original Spark Streaming. To be more precise, Fig. 11(b) further depicts the output data size in each time, across all queries, where the curve of Wide-area Spark Streaming is higher than that of Original Spark Streaming at most of the time. This indirectly implies that Wide-area Spark Streaming is capable of processing more batches given a same duration when comparing to Original Spark Streaming.

6.4 Results of queries of mixed type

As aforementioned in Sec. 1, there are typically many types of streaming analytics queries. We now focus on evaluating the impact of mixed queries on the performance of our Wide-area Spark Streaming, in terms of the shuffle transfer time as well as the processing time.

We use three types of streaming queries: `WordCount`, `Grep` and `Top-k`, working on the Wikipedia dataset. Fig.12 shows the input-output relationship for `Grep` and `Top-k` streaming queries, by varying the input data sizes. It is clear that the input-output relationship for `Grep` query is approximately linear, while the output size for `Top-k` query remains constant irrespective of the input data sizes. It should be noted that when using a different dataset, we may need to do another job profiling to learn the input-output relationship for each application. But fortunately, such job profiling will not bring too much impact to the application. The reason is that the streaming query could be repeated and the workload is largely static and changing infrequently for typical streaming jobs [18]–[20]. Hence, we can calculate the routing and batch sizing decisions for multiple micro-batches at once and reuse these decisions across micro-batches. To further reduce the overhead added by such job profiling, we can use techniques like `pseudo-distributed execution` [5] or online profiling [7].

With the above input-output relationships, we can incorporate these three types of streaming queries in our algorithm. Specifically, in our experiment, we use 5 queries for each streaming query type, and accordingly there are 15 queries in total. We run these 15 queries for one hour. In the beginning of every 5-minute interval, we change the input data generate rate for each query, and run our algorithm 1 to calculate the updated batch size and routing path for each streaming query.

Fig. 13 first shows the shuffle transfer time per batch for each of the queries during the period of one hour, under both Wide-area Spark Streaming and Original Spark Streaming. We can clearly find that Wide-area Spark Streaming achieves a lower shuffle transfer time per batch for `WordCount1` query, while maintains almost the same shuffle transfer time for the other `WordCount` queries, compared to Original Spark Streaming across all time intervals. For `Grep` and `Top-K` queries, Wide-area Spark Streaming can also outperform the Original Spark Streaming, with respect to the shuffle transfer time per batch. To be more specific, we can further observe that across all time intervals and all `WordCount` queries, the average shuffle transfer times for both the Wide-area Spark Streaming and the Original Spark Streaming are 19.47 ms and 31.91 ms, respectively. As for the `grep` queries, the average shuffle transfer time of Wide-area Spark Streaming, i.e., 13.18 ms, is a little bit higher than that of Original Spark Streaming, i.e., 12.6 ms. The root reason is that `Grep` queries generate more output data than `WordCount` application, making it to be an inferior position when competing network resource with `WordCount` applications. Furthermore, for the `top-K` queries, the average shuffle transfer times for both the Wide-area Spark Streaming and the Original Spark Streaming are 16.9 ms and 18.9 ms, respectively.

Fig. 14 further presents the processing time per batch for all queries over the one-hour period, under both Wide-area Spark Streaming and Original Spark Streaming. It is clear that for `wordcount1`, `Grep2`, `Grep4`, `Top-K1`,`Top-K3`, and `Top-K5` queries, Wide-area Spark Streaming can significantly reduce the processing time per batch, when compared to the Original Spark Streaming. For the remaining streaming queries, Wide-area Spark Streaming and Original Spark Streaming have approximately the same processing time per batch. After taking a further observation, we find that: 1) For `WordCount` queries, the average processing time across all time intervals is 264,6 ms for Wide-area Spark Streaming, while that value can be as high as 9124.3 ms for the Original Spark Streaming; 2) For the `Grep` and `Top-K` queries under the Wide-area Spark Streaming, the average processing times are 8.18 ms and 5.71 ms respectively, while these two values are a little bit higher under the Original Spark Streaming, i.e., 8.35 ms and 6.01 ms, respectively. These results directly demonstrate that our Wide-area Spark Streaming can reduce the average processing time over all time intervals and all types of streaming queries, compared to the Original Spark Streaming.

7 Related Work

With the emergence and popularity of Internet of Things (IoT), online social networks and massive social media, large volumes

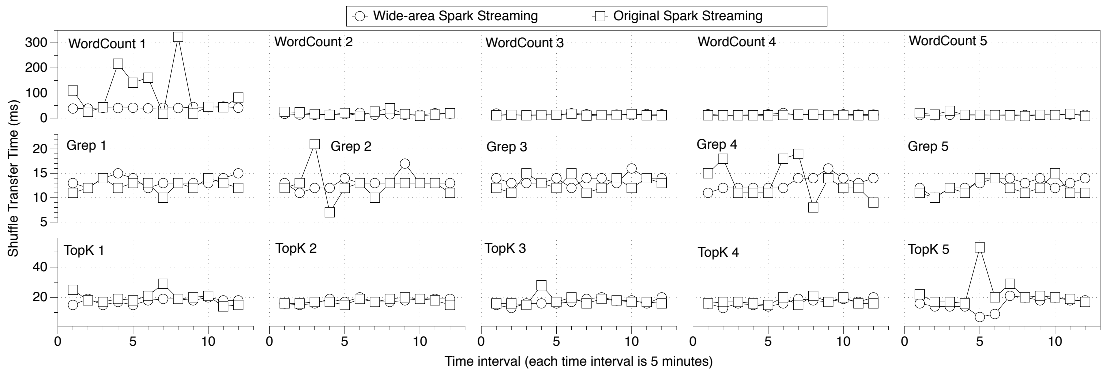


Fig. 13. The shuffle transfer time per batch across all queries during a period of one hour.

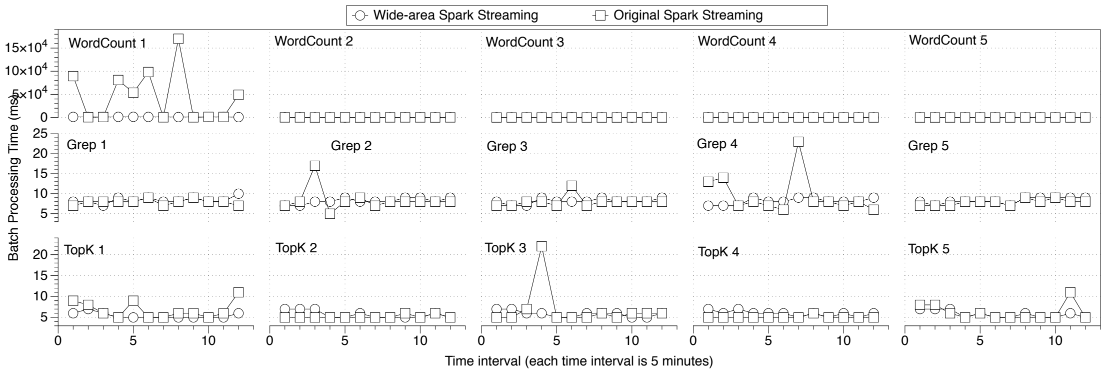


Fig. 14. The processing time per batch across all queries during a period of one hour.

of data streams have been generated at the edge of the Internet. As a result, there is a growing demand for improving the processing of geo-distributed data streams. We will summarize the following three categories of related work:

7.1 Batch processing in the wide area

Batch processing refers to the processing of a large and fixed amount of input data that is collected ahead of time. Existing solutions on batch processing in the wide area focused on minimizing either the WAN bandwidth cost or the job completion time.

With respect to minimizing the WAN bandwidth cost, Geode proposed to minimize the WAN bandwidth utilization with regulatory constraints by optimizing the query execution plan and data replication strategy [5]. Pixida studied a new graph partitioning problem to schedule the tasks with minimum data movement across different regions [23]. WANalytics was proposed as an extension of Pixida, and used an efficient cache mechanism to reduce data transfer among datacenters [24]. Li *et al.* proposed to jointly consider input data movement and task placement to minimize the inter-datacenter traffic generated by MapReduce jobs targeting on geo-distributed data, while providing predicted job completion times [25]. These solutions only focused on reducing the bandwidth cost without considering job completion times.

Towards optimizing the job completion time, Iridium optimized the placement of both the input data and reduce task, with the purpose of achieving low latency for geo-distributed data analytical queries [9]. It relied on the idealistic assumption that reduce tasks are infinitesimally divisible, and solved a mixed integer programming (MIP) problem instead in its implementation. Flutter proposed to schedule reduce tasks close to the data by using the exact amount of intermediate data that each reduce task would read, and by solving an integer linear programming (ILP) problem [26]. Jayalath *et al.* proposed G-MR, which considers both the job completion time and cost when executing a sequence of jobs [27].

Hung *et al.* proposed SWAG, which leveraged a greedy job scheduling algorithm to optimize the average job completion time [28]. Zhou *et al.* developed a new method to map parallel processes to physical nodes in a near optimal way [29]. Nevertheless, they neglected the issue of routing, which makes their solution less efficient for minimizing the job completion times. Clarinet used heuristics to jointly selects the placements and schedules of tasks for reducing the query completion time [30]. In contrast, our work focused on streaming analytics, instead of batch processing. We have seamlessly combined the routing and batch sizing to reduce job completion times.

7.2 Stream processing in the wide area

A number of new streaming computing engines for large-scale stream processing have been proposed. For example, Spark Streaming [1], MillWheel [31], Storm [32] and Comet [33] have been designed for single-datacenter environments with high-bandwidth networks. However, they were unaware of the significant bandwidth variation on WAN links. Moreover, all of them focused on providing fault-tolerant streaming services, which are orthogonal to our concerns of bandwidth variations on WAN links with scarce bandwidth.

Hwang *et al.* proposed to replicate stream processing operators, with the aim of achieving fast and reliable stream processing over wide area networks [34]. Pietzuch *et al.* proposed a new layer between a stream-processing system and the physical network that manages operator placement, with a series of objectives such as improving network utilization and providing low stream latency [35]. Rather than optimizing the operator placement, we leveraged batch sizing and detour paths to speed up stream processing across wide area networks. JetStream [22] proposed to trade accuracy for reducing data sizes, while our work preserves data fidelity.

7.3 Batch sizing in stream processing systems

Storm [32], TimeStream [36], TeleGraphCQ [37] processed data streams based on continuous operator models, where long-lived operators exchange messages with each other in a predefined order. The streaming data can be computed immediately as it arrives. Spark Streaming [1] and Comet [33] can be classified as *micro-batch* computing engines, where streaming data arriving within a batch interval will be collected together as a *micro-batch*, and they will be computed just like a traditional batch.

Our work mainly focused on stream processing systems based on the notion of micro-batches. Das *et al.* [3] discussed how the batch size affected the performance of streaming workloads. They designed a control algorithm to automatically adapt batch sizes to make sure that the batched data can be processed as fast as they arrive. It is implemented within a datacenter where available bandwidth is consistently high. In contrast, we consider WAN links and aim at forwarding intermediate results on faster paths to reduce the transfer time and to achieve a smaller batch size. As a result, the processing time can be reduced as well.

8 Concluding Remarks

This paper has proposed a sparsity-penalized ADMM algorithm for streaming analytics over wide area where the bandwidth is not always sufficient and varies significantly on all the WAN links. By jointly selecting the best path and batch sizing, our algorithm can take advantage of detoured route with sufficient bandwidth to effectively reduce the processing time of each micro-batch. Our routing functionality implemented in Spark Streaming enforces intermediate data to follow the path optimized by our algorithm, and allows the additional task to be placed at our desired location. Extensive performance evaluation based on experiments conducted on an Amazon EC2 cluster implies that our solution can support higher query response rates, a larger query output rate, with significantly reduced tail processing latencies and network transfer times, while keeping the stream processing system stable.

References

[1] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized Streams: Fault-Tolerant Streaming Computation at Scale," in *Proc. Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013.

[2] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing," in *Proc. USENIX NSDI*, 2012.

[3] T. Das, Y. Zhong, I. Stoica, and S. Shenker, "Adaptive Stream Processing Using Dynamic Batch Sizing," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. Proc. ACM SoCC, 2014.

[4] B. Heintz, A. Chandra, and R. K. Sitaraman, "Trading timeliness and accuracy in geo-distributed streaming analytics," in *Proceedings of ACM Symposium on Cloud Computing (SoCC)*, 2016, pp. 361–373.

[5] A. Vulimiri, C. Curino, P. B. Godfrey, T. Jungblut, J. Padhye, and G. Varghese, "Global Analytics in the Face of Bandwidth and Regulatory Constraints," in *Proc. USENIX NSDI*, 2015.

[6] K. Hsieh, A. Harlap, N. Vijaykumar, D. Konomis, G. R. Ganger, P. B. Gibbons, and O. Mutlu, "Gaia: Geo-distributed machine learning approaching lan speeds." in *Proceedings of USENIX NSDI*, 2017, pp. 629–647.

[7] B. Zhang, X. Jin, S. Ratnasamy, J. Wawrzynek, and E. A. Lee, "Awstream: Adaptive wide-area streaming analytics," in *Proceedings of ACM SIGCOMM*, 2018, pp. 236–252.

[8] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein, "Distributed optimization and statistical learning via the alternating direction method of multipliers," *Foundations and Trends® in Machine Learning*, vol. 3, pp. 1–122, 2011.

[9] Q. Pu, G. Ananthanarayanan, P. Bodik, S. Kandula, A. Akella, P. Bahl, and I. Stoica, "Low Latency Geo-Distributed Data Analytics," in *Proc. ACM SIGCOMM*, 2015.

[10] Y. Yu, P. K. Gunda, and M. Isard, "Distributed Aggregation for Data-parallel Computing: Interfaces and Implementations," in *Proc. ACM SOSP*, 2009.

[11] "Wikipedia," https://en.wikipedia.org/wiki/Main_Page.

[12] R. Tibshirani, "Regression shrinkage and selection via the lasso," *Journal of the Royal Statistical Society. Series B (Methodological)*, pp. 267–288, 1996.

[13] E. J. Candes, M. B. Wakin, and S. P. Boyd, "Enhancing sparsity by reweighted âĎŞ 1 minimization," *Journal of Fourier analysis and applications*, vol. 14, no. 5-6, pp. 877–905, 2008.

[14] M. Fazel, "Matrix rank minimization with applications," Ph.D. dissertation, PhD thesis, Stanford University, 2002.

[15] W. Li, D. Niu, Y. Liu, S. Liu, and B. Li, "Wide-area spark streaming: Automated routing and batch sizing," in *Proceedings of IEEE International Conference on Autonomic Computing (ICAC)*, 2017, pp. 33–38.

[16] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel, "The cost of a cloud: research problems in data center networks," *ACM SIGCOMM computer communication review*, vol. 39, no. 1, pp. 68–73, 2008.

[17] K. Ousterhout, A. Panda, J. Rosen, S. Venkataraman, R. Xin, S. Ratnasamy, S. Shenker, and I. Stoica, "The case for tiny tasks in compute clusters." in *Proceedings of HotOS*, 2013.

[18] S. Venkataraman, A. Panda, K. Ousterhout, M. Armbrust, A. Ghodsi, M. J. Franklin, B. Recht, and I. Stoica, "Drizzle: Fast and adaptable stream processing at scale," in *Proceedings of ACM SOSP*, 2017.

[19] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan, "Availability in globally distributed storage systems." in *Proceedings of USENIX OSDI*, 2010.

[20] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch, "Power management of online data-intensive services," in *Proceedings of ACM ISCA*, 2011.

[21] Y. M. Chen, L. Dong, and J.-S. Oh, "Real-Time Video Relay for UAV Traffic Surveillance Systems through Available Communication Networks," in *2007 IEEE Wireless Communications and Networking Conference*, 2007.

[22] A. Rabkin, M. Arye, S. Sen, V. S. Pai, and M. J. Freedman, "Aggregation and Degradation in Jetstream: Streaming Analytics in the Wide Area," in *Proc. USENIX NSDI*, 2014.

[23] K. Kloudas, M. Mamede, N. Preguiça, and R. Rodrigues, "Pixida: Optimizing Data Parallel Jobs in Wide-Area Data Analytics," *Proc. VLDB Endow.*, vol. 9, no. 2, Oct. 2015.

[24] A. Vulimiri, C. Curino, P. B. Godfrey, T. Jungblut, K. Karanasos, J. Padhye, and G. Varghese, "WANalytics: Geo-Distributed Analytics for a Data Intensive World," in *Proc. ACM SIGMOD International Conference on Management of Data*, 2015.

[25] P. Li, S. Guo, T. Miyazaki, X. Liao, H. Jin, A. Y. Zomaya, and K. Wang, "Traffic-aware geo-distributed big data analytics with predictable job completion time," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 6, pp. 1785–1796, 2017.

[26] Z. Hu, B. Li, and J. Luo, "Flutter: Scheduling Tasks Closer to Data Across Geo-Distributed Datacenters," in *Proceedings of IEEE INFOCOM*, 2016.

[27] C. Jayalath, J. Stephen, and P. Eugster, "From the cloud to the atmosphere: running mapreduce across data centers," *IEEE Transactions on Computers (TC)*, vol. 63, no. 1, pp. 74–87, 2014.

[28] C.-C. Hung, L. Golubchik, and M. Yu, "Scheduling jobs across geo-distributed datacenters," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*. ACM, 2015, pp. 111–124.

[29] A. C. Zhou, Y. Gong, B. He, and J. Zhai, "Efficient process mapping in geo-distributed cloud data centers," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017.

[30] R. Viswanathan, G. Ananthanarayanan, and A. Akella, “Clarinet: Wan-aware optimization for analytics queries,” in *Proceedings of USENIX OSDI*, 2016.

[31] T. Akidau, A. Balikov, K. Bekiroglu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle, “MillWheel: Fault-Tolerant Stream Processing at Internet Scale,” in *Very Large Data Bases*, 2013.

[32] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy, “Storm@Twitter,” in *Proc. ACM SIGMOD*, 2014.

[33] B. He, M. Yang, Z. Guo, R. Chen, B. Su, W. Lin, and L. Zhou, “Comet: Batched Stream Processing for Data Intensive Distributed Computing,” in *Proc. ACM SoCC*, 2010.

[34] J. H. Hwang, U. Cetintemel, and S. Zdonik, “Fast and Reliable Stream Processing over Wide Area Networks,” in *Proc. IEEE International Conference on Data Engineering Workshop*, 2007.

[35] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer, “Network-Aware Operator Placement for Stream-Processing Systems,” in *Proc. IEEE International Conference on Data Engineering*, 2006.

[36] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang, “TimeStream: Reliable Stream Computation in the Cloud,” in *Proc. ACM EuroSys*, 2013.

[37] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah, “TelegraphCQ: Continuous Dataflow Processing,” in *Proc. ACM SIGMOD*, 2003.

Wenxin Li received the B.E. degree from the School of Computer Science and Technology, Dalian University of Technology, China, in 2012. Currently, he is a visiting student in the Department of Electrical and Computer Engineering, University of Toronto, Toronto, Canada as well as a Ph.D. candidate in the School of Computer Science and Technology, Dalian University of Technology, China. His research interests include datacenter networks and cloud computing.

Di Niu received the B.Engr. degree from the Department of Electronics and Communications Engineering, Sun Yat-sen University, China, in 2005 and the M.A.Sc. and Ph.D. degrees from the Department of Electrical and Computer Engineering, University of Toronto, Toronto, Canada, in 2009 and 2013. Since 2012, he has been with the Department of Electrical and Computer Engineering at the University of Alberta, where he is currently an Assistant Professor. His research interests span the areas of cloud computing and storage, multimedia delivery systems, data mining and statistical machine learning for social and economic computing, distributed and parallel computing, and network coding. He is a member of IEEE and ACM.

Yinan Liu received her B.Engr. degree from the School of Telecommunications in Xidian University, Xiąŕan, China, in July 2014 and her M.S. degree from the Department of Electrical and Computer Engineering, University of Toronto, Toronto, Canada. Her general research interests cover inter-datacenter and software-defined networking, and she hopes to design something that can solve practical problems.

Shuhao Liu received his B.S. degree from Department of Automation, Tsinghua University in 2012. Then he joined National University of Defence Technology, Changsha, where he received a M.Sc. degree in 2014. During that time, he visited the City University of Hong Kong for five months, working as a research assistant. Currently, he is a Ph.D. candidate in the Department of Electrical and Computer Engineering, University of Toronto, Toronto, Canada. His research topic includes datacenter networking, software-defined networking and big data analytics.

Baochun Li received his B.Engr. degree from the Department of Computer Science and Technology, Tsinghua University, China, in 1995 and his M.S. and Ph.D. degrees from the Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, in 1997 and 2000. Since 2000, he has been with the Department of Electrical and Computer Engineering at the University of Toronto, where he is currently a Professor. He holds the Bell Canada Endowed Chair in Computer Engineering since August 2005. His research interests include cloud computing, multimedia systems, applications of network coding, and wireless networks. He was the recipient of the IEEE Communications Society Leonard G. Abraham Award in the Field of Communications Systems in 2000, the Multimedia Communications Best Paper Award from the IEEE Communications Society in 2009, and the University of Toronto McLean Award in 2009. He is a member of ACM and a Fellow of IEEE.