Wide-Area Spark Streaming: Automated Routing and Batch Sizing

Wenxin Li*, Di Niu†, Yinan Liu‡, Shuhao Liu‡, Baochun Li‡
*University of Toronto & Dalian University of Technology
†University of Alberta
‡University of Toronto

***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 focus on reducing latencies for spark streaming queries in wide-area networks, by automatically selecting data flow routes and determining micro-batch sizes across geo-distributed datacenters. Specifically, we formulate a nonconvex optimization problem, and solve it with an efficient heuristic algorithm based on readily measurable operating traces. We conducted experiments on Amazon EC2 with emulated bandwidth constraints. Our experimental results have demonstrated the effectiveness of our proposed algorithm, as compared to the existing Spark Streaming.

I. 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*. 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 the streaming computation is carried out as a continuous series of MapReduce operations on the micro-batches.

However, the current Spark Streaming framework mainly focuses on fast recovery from faults and stragglers (slow nodes) [1] in a single datacenter, with high-bandwidth networks. Although streaming analytics find a wide variety of applications in the wide area, 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 response, the operator may lose the key chance to make decisions based on the query.

In this paper, we jointly consider bandwidth-aware batch sizing, task scheduling and routing of data streams, to reduce latencies for queries in state-of-the-art batched stream processing systems, typically represented by Spark Streaming, across wide-area networks. 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 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 make two original contributions in this paper:

First, we formulate the problem of joint batch sizing, task placement and routing as a *nonconvex optimization* problem. We solve it with an efficient heuristic algorithm, decoupling path selection and batch sizing into two alternately solved

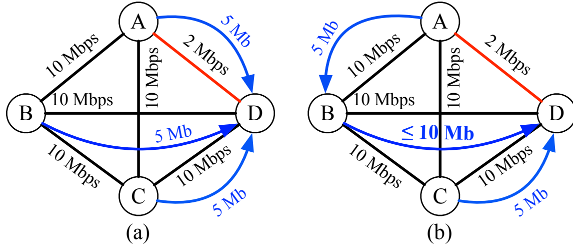

Fig. 1. The topology of a motivating example. Red lines represent bottleneck links.

subproblems. Given a certain batch size, we also describe how to estimate the output rate for each query as well as the network usage on each link based on statistical learning. Such estimation will serve as the input to the joint batch sizing and routing problem.

Second, we evaluated the performance of the proposed 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 20 coexisting streaming `WordCount` queries 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.

II. System Model and Problem Formulation

We first motivate the importance of intelligent data routing and task scheduling in wide area networks 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 display 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 Mbits 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.

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 Mbits 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.

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 *bi-convex* 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 use a graph $G = (V, E)$ to represent a network of geo-distributed node, $N = |V|$ being the number of nodes. 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.

Let $M_i(\tau_i)$ denote the output size of query i per batch at the collecting node D_i as 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.,

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

where U_i is a function that can be learned from the data characteristics of each particular application, which we will illustrate in Sec. III. 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 throughput 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.,

$$M_{ij}^e(\tau_i) = U_i\left(\sum\nolimits_{v\in S_i \bigcap \text{Descendants of } e \text{ in } T_{ij}} r_{vi}\tau_i\right).$$

Similar to the goodput $R_i(\tau_i)$, if tree T_{ij} is selected, the throughput 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 for each query i, $i = 1, \ldots, Q$, the optimal aggregation tree as well as its optimal batch size τ_i, in order to maximize the total goodput $\sum_{i=1}^{Q} R_i(\tau_i)$ of information. 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:

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

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

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

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

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

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

III. Solution and Algorithms

In this section, we present a heuristic algorithm to solve the optimization problem (1) and describe how to learn the basis functions $R_{ij}^e(\cdot)$ and $R_i(\cdot)$) from trace data.

Assuming the basis functions $R_{ij}^e(\cdot)$ and $R_i(\cdot)$ are learnt, problem (1) is a hard *non-convex* problem, since there is a multiplication between $R_{ij}^e(\tau_i)$ and x_{ij} that leads to non-convexity in the constraints. Hence, we solve Problem 1 by proposing a heuristic algorithm which decouples path selection and batch sizing. The key idea is to alternately solve two subproblems, for tree selection and batch sizing, respectively.

Our heuristic algorithm is summarized in Algorithm 1. Specifically, it initially chooses the direct path for each query i and initializes $\{x_{ij}\}$ accordingly. In each iteration, it solves two subproblems alternately: it first maximizes the total goodput $\sum_{i=1}^{Q} R_i(\tau_i)$ over $\{\tau_i\}$ while keeping the path selection $\{x_{ij}\}$ fixed; it then maximizes the aggregate *residual network bandwidth* by adjusting the path selection $\{x_{ij}\}$ while keeping the batch sizes $\{\tau_i\}$ fixed. For the path selection subproblem, $x_{ij}'s$ are first relaxed to fractional numbers and finally rounded to 0 or 1, such that each query only chooses a single aggregation tree. The intuition is that we should minimize the batch sizes when the path selection is fixed and should create more residual bandwidth by adjusting path selection when the batch sizes are fixed.

Algorithm 1 Iterative alternated optimization for Problem (1).

1: **Input**: Basis functions $\{R_i(\cdot)\}$, $\{R_{ij}^e(\cdot)\}$; link capacities $\{C_e\}$.
2: **Output**: $\{x_{ij}^k\}$, $\{\tau_i^k\}$ when the algorithm stops.
3: $k := 0$. Initialize $\{x_{ij}^0\}$ by choosing the direct path for each query i.
4: Solve the following subproblem to obtain $\{\tau_i^k\}$:

$$\underset{\{\tau_i\}}{\text{maximize}} \quad \sum_{i=1}^{Q} R_i(\tau_i) \tag{6}$$
$$\text{subject to} \quad \sum_{i=1}^{Q} \sum_{j:e\in T_{ij}} R_{ij}^e(\tau_i)\cdot x_{ij}^k \le C_e, \quad \forall e \in E,$$
$$\tau_i \ge \tau_i^l, \quad i = 1, \ldots, Q.$$

5: Solve the following subproblem to obtain $\{x_{ij}^{k+1}\}$:

$$\underset{\{x_{ij}:0\le x_{ij}\le 1\}}{\text{maximize}} \quad \sum_{e\in E}\left(C_e - \sum_{i=1}^{Q} \sum_{j:e\in T_{ij}} R_{ij}^e(\tau_i^k)\cdot x_{ij}\right) \tag{7}$$
$$\text{subject to} \quad \sum_{j=1}^{k_i} x_{ij} = 1, \quad i = 1, \ldots, Q,$$

6: $k := k + 1$. Repeat Steps 4-5 until the stop criterion is met.
7: For each particular query i, round the largest x_{ij} to 1 and the remaining $x_{ij}'s$ to 0.

It is worth noting that the subproblems involved in Steps 4 and 5 of Algorithm 1 and be efficiently solved by standard optimization techniques or linear programming.

Now we describe the procedure to learn the basis functions, which are required as the input to Algorithm 1. As has been shown in Sec. II, 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 [3], 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,

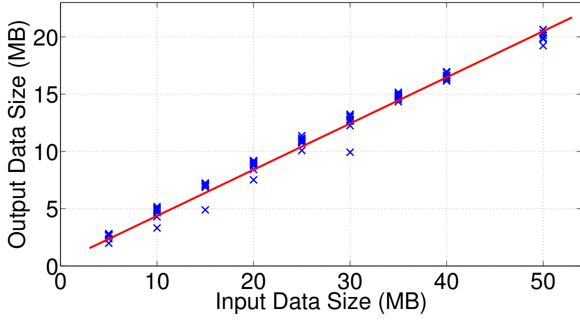

Fig. 2. The input-output relationship of `WordCount` based on 12 GB of Wikipedia data. Linear regression reveals the fit of the model for the output size $U(I) = 0.40I + 0.43$ as a function of the input size I.

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 [4]. 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 query for big data platforms, based on a publicly available 12 GB text dataset from Wikipedia [5]. 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, $R^e_{ij}(\tau_i)$ is linear in $1/\tau_i$. Therefore, in this case, Problem (6) can be converted to a linear program.

IV. Performance Evaluation

We conducted real-world experiments to evaluate our extended Spark Streaming framework in an emulated bandwidth-constrained environment on Amazon EC2 instances. We compare the proposed Wide-area Spark Streaming to the original Spark Streaming with a data locality mechanism.

A. 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. Since the cost of storage

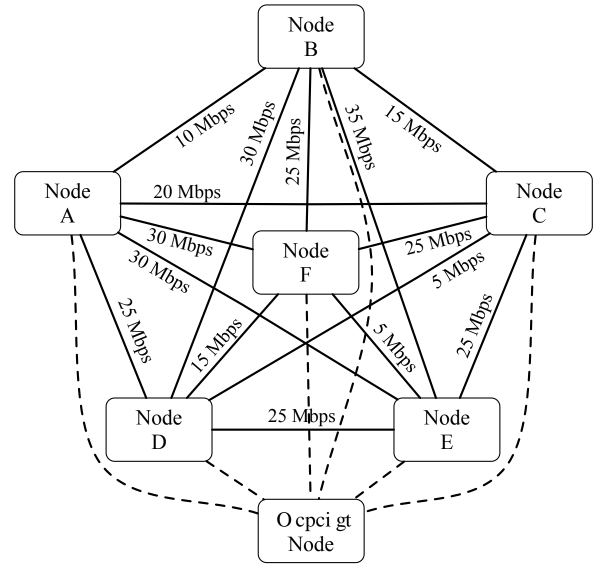

Fig. 3. The WAN emulation testbed launched for streaming experiments.

and processors is decreasing at a faster pace than that of provisioning bandwidth [6], [7], 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. 3, 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 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. 4. 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.

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

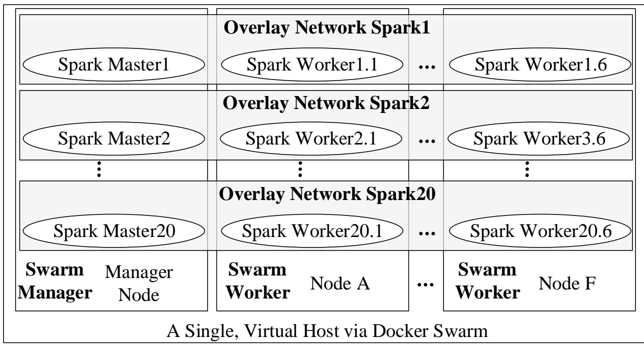

Fig. 4. Deploying wide area streaming on the launched cluster via Docker.

TABLE I
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.

Mbps, as shown in Table I.

B. Batch Sizing and Evaluation Methodology

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: use the proposed Algorithm 1 to jointly determine the proper batch size τ_{BW} for each query as well as the routing paths, i.e., the aggregation tree, for each query. Enforce scheduling decisions by modifying application workflows.

Original Spark Streaming: each input source node of a query sends its locally aggregated input data to the central collecting site, where the data is further processed and aggregated to produce final query output. We run Algorithm 1 with such direct path selection for each query to obtain its optimal batch sizes. The built-in data locality mechanisms in Spark are used to enforce all the reduce tasks be placed at the output node. This strategy represents a common industrial practice conforming to data locality.

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.

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.

C. Experimental Results

The processing time per micro-batch is the most important performance metric for streaming queries, involving both the computation time and shuffling time. The latter may include the time to transfer data in WANs. We show the average and 95th percentile processing time per micro-batch for each query in Fig. 5. It is clear that for Query 1, 5, 8, 9, 10, 14, 15 and 17, Wide-area Spark Streaming achieves smaller batch processing times, as compared to Original Spark Streaming. We can further observe that there is not much difference for the remaining queries, in terms of the batch processing time. Wide-area Spark Streaming intelligently leverages high-bandwidth detoured paths to serve distributed streaming queries, leading to a higher query rate and reduced processing times. The above results verify that our launched testbed on the Amazon EC2 platform with Docker-based deployment is appropriate to evaluate Wide-area Spark Streaming.

V. RELATED WORK

Geo-distributed data analytics has become an active research topic recently. Existing work mainly falls into two categories:

Batch processing in the wide area. Geode [8], [9] and Pixida [10] are proposed to reduce cross-datacenter traffic by optimizing the query execution plan, data replication and task scheduling. Flutter [11] and Iridium [3] aim to reduce the job completion time by moving reduce tasks close to data. In contrast, our work focuses on streaming analytics, instead of batch processing.

Stream processing in wide area. A number of new streaming computing engines for large-scale stream processing are presented in recent literature [1], [7], [12]–[16]. Some of them [1], [12], [14] focus on providing fault-tolerant streaming services, which are orthogonal to our concerns of bandwidth variations on WAN links with scarce bandwidth. The most related work is JetStream [7], while our work is different from it. Though we both care the bandwidth constraints, the strategy of aggregation and degradation leveraged in JetStream trades accuracy for reducing data size while our work preserves the data fidelity. Regarding batch-sizing in streaming processing, Das et al. [17] have designed a control algorithm to automatically adapt batch size to make sure that the batched data can be processed as fast as they arrived. It is implemented within a datacenter which available bandwidth is consistently high. However, our work faces to WAN links.

Task scheduling. There is a recent trend to enhance the performance of distributed jobs by carefully scheduling tasks [18]–[21]. These work intended to accomplish different goals. For example, Quincy [18] took data locality into consideration to reduce job completion time; Delay Scheduling [19] balanced the locality and fairness. However, our work builds upon the default Spark scheduler [19], and seamlessly combines

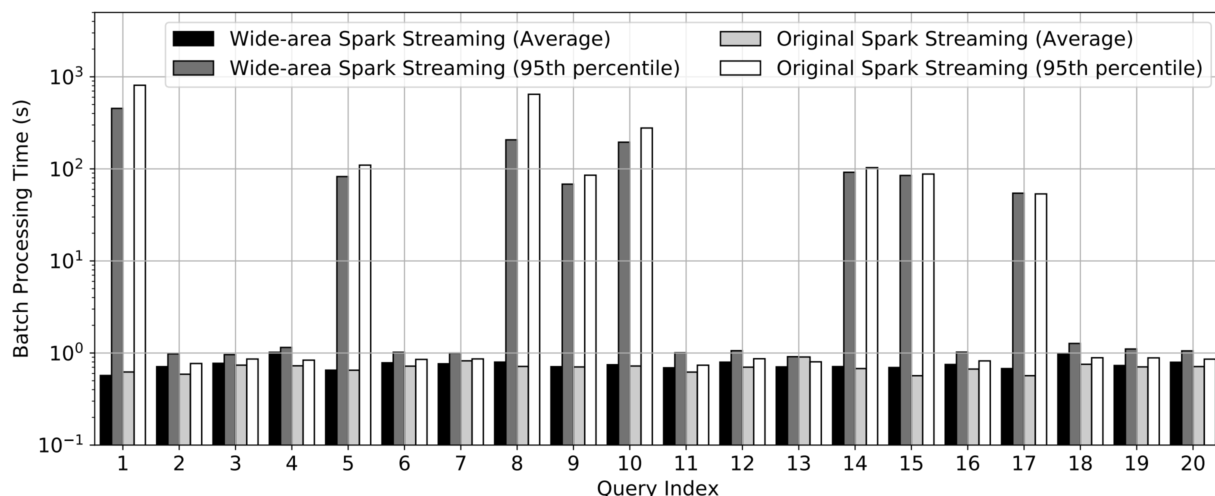

Fig. 5. The average and 95th percentile processing time per batch for each of the 20 queries.

routing and batch sizing to optimize the process time of streaming query applications.

VI. Concluding Remarks

This paper addresses the challenging problem of automated data routing and batch sizing for running Spark Streaming in wide area networks where the bandwidth is not always sufficient and varies significantly among different WAN links. To jointly select the best path and batch sizes, we formulate a nonconvex optimization, and solve it by decoupling routing and batch sizing into two alternately solved subproblems. Our algorithm can take advantage of detoured route with higher bandwidth to effectively reduce the processing time of each micro-batch. 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.

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] 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.

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

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] Z. Hu, B. Li, and J. Luo, "Flutter: Scheduling Tasks Closer to Data Across Geo-Distributed Datacenters."

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: Fair Scheduling for Distributed Computing Clusters," in *Proc. ACM SIGOPS*, 2009.

[19] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling," in *Proc. ACM EuroSys*, 2010.

[20] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: Distributed, Low Latency Scheduling," in *Proc. ACM SOSP*, 2013.

[21] X. Ren, G. Ananthanarayanan, A. Wierman, and M. Yu, "Hopper: Decentralized Speculation-Aware Cluster Scheduling at Scale," in *Proc. ACM SIGCOMM*, 2015.