Delay-Resistant Geo-Distributed Analytics

Habib Mostafaei, *Member, IEEE*, Georgios Smaragdakis, *Senior Member, IEEE*, Thomas Zinner, *Member, IEEE*, and Anja Feldmann

***Abstract*—Big data analytics platforms have played a critical role in the unprecedented success of data-driven applications. However, real-time and streaming data applications, and recent legislation, e.g., GDPR in Europe, have posed constraints on exchanging and analyzing data, especially personal data, across geographic regions. To address such constraints data has to be processed and analyzed in-situ and aggregated results have to be exchanged among the different sites for further processing. This introduces additional network delays due to the geographic distribution of the sites and potentially affecting the performance of analytics platforms that are designed to operate in datacenters with low network delays. In this paper, we show that the three most popular big data analytics systems (Apache Storm, Apache Spark, and Apache Flink) fail to tolerate round-trip times more than 30 milliseconds even when the input data rate is low. The execution time of distributed big data analytics tasks degrades substantially after this threshold, and some of the systems are more sensitive than others. A closer examination and understanding of the design of these systems show that there is no winner in all wide-area settings. However, we show that it is possible to improve the performance of all these popular big data analytics systems significantly amid even transcontinental delays (where inter-node delay is more than 30 milliseconds) and achieve performance comparable to this within a datacenter for the same load.**

***Index Terms*—Wide-area analytics, big data analytics, geo-distributed systems, networked systems.**

I. Introduction

BIG DATA analytics platforms [1], [2], [3], [4], [5], [6] have played a critical role in the unprecedented success of data-driven applications. Such platforms are typically deployed in server clusters and datacenters to support applications that analyze big data ranging from personal data to Web visit logs and purchase histories [7], [8]. As the data-driven applications become more complex and sophisticated, the source of data can be volatile and the traffic volume too high to be handled in a single datacenter. For example, a global advertisement campaign may take as input the tweets of the global population of Twitter users to place advertisements. In such scenarios, the data cannot be forwarded to a single datacenter, as it adds delays that many applications cannot tolerate [9] or yields drops in sales [10].

Moreover, in recent years, there are additional constraints on how to exchange and analyze data, especially personal data, across different regions. For example, the European Union General Data Protection Regulation (GDPR) [11] requires the user to give consent for her data to be processed or exchanged with other systems, or stored and processed in other countries and infrastructures that cannot guarantee a level of privacy protection as in the European Union. Similar privacy regulations are now in effect in other regions [12], [13], e.g., in California (California Consumer Privacy Act (CCPA) [14]), Canada [15], Israel [16], Japan [17], and Australia [18]. Accordingly, there is an increasing need to analyze data that are received at different geographic locations (sites) in-situ to comply with regulations thus complicating continental or global analytics tasks. Hence, to support such superior analytics tasks, results have to be aggregated and shared with other sites. In such scenarios, input data can be modeled as an infinite stream of data that arrives at a processing site. It has to be analyzed fast, and aggregated results have to be exchanged with all the other processing sites towards fulfilling the intended task in a timely fashion. Recall that the intermediate results (which have significantly lower volume than the raw data) have to be exchanged among all sites until the task is completed. Thus, the slowest exchange of data dictates the overall time to execute the task.

Data Stream Processing (DSP) systems, such as Storm [1], Spark [3], and Flink [5] have been evolved over the years offering efficient and reliable solutions in a datacenter environment to process streams of data. How easy is it for these DSPs that are proven exceptionally robust and thriving in the data center to adapt in the wide-area setting? Should we develop new systems or is it enough to properly understand and tune the existing ones to achieve performance in the wide-area environment close to this within a datacenter?

In this paper, we show that the above three popular implementations of DSPs fail to tolerate round-trip times of more than 30 milliseconds even with a low input data rate. Both the throughput as well as the execution time degrades substantially after this threshold. Indeed, 30 milliseconds is a rather

Manuscript received 10 November 2021; revised 12 April 2022 and 12 July 2022; accepted 16 July 2022. This work was supported in part by the European Research Council (ERC) Starting Grant ResolutioNet (ERC-StG-679158) and by the German Ministry for Education and Research (BMBF) as BIFOLD - Berlin Institute for the Foundations of Learning and Data (01IS18025A, 01IS18037A) and "Souverän. Digital. Vernetzt." Joint project 6G-RIC, project identification number: PIN 16KISK027. The associate editor coordinating the review of this article and approving it for publication was K. Xue. *(Corresponding author: Habib Mostafaei.)*

Habib Mostafaei was with Technische Universität Berlin, 10623 Berlin, Germany. He is now with the Department of Mathematics and Computer Science, Eindhoven University of Technology, 5600 MB Eindhoven, The Netherlands (e-mail: h.mostafaei@tue.nl).

Georgios Smaragdakis is with the Cybersecurity Group, Intelligent Systems Department, Faculty of Electrical Engineering, Mathematics, and Computer Science, TU Delft, 2626 XE Delft, The Netherlands, also with the Internet Architecture Department, Max Planck Institute for Informatics, 66123 Saarbrücken, Germany, and also with the Berlin Institute for the Foundations of Learning and Data, 10587 Berlin, Germany.

Thomas Zinner is with the Department of Information Security and Communication Technology (IIK), Norwegian University of Science and Technology, 7491 Trondheim, Norway.

Anja Feldmann is with the Internet Architecture Department, Max Planck Institute for Informatics, 66123 Saarbrücken, Germany.

Digital Object Identifier 10.1109/TNSM.2022.3192710

high round-trip delay in a datacenter environment; the architecture of such systems was designed and optimized for very low round-trip times (less than 10 milliseconds). In the past, streaming systems, e.g., Jetstream [19] and AWStream [20], were developed to improve the performance of streaming applications in the wide-area environment by introducing novel architectures for distributed server log processing, and video-related applications such as augmented reality, pedestrian detection, and monitoring log analysis, respectively.

In our study, we show that in many settings, it is sufficient to deploy and operate one of the three popular DSPs in the wide-area environment. This is preferable, because *(i)* these systems are already in use at datacenters and can better interoperate with others deployed at other sites, and *(ii)* it does not require retraining of data engineers and programmers that are already familiar with the programming and operation of these systems. Although there is no clear winner in all topologies and operational settings we consider in this study, some of the three DSPs are significantly more sensitive than others in different settings with regards to pairwise site delay, number of sites, and data load. A closer examination and understanding of the differences in the architecture and implementation of each DSP shows that, in many settings, the performance of some of them can be improved, and be comparable with that of a datacenter where the inter-node delays are minimal. Indeed, we showcase that it is possible to reduce the execution time of stream analytics queries that rely on geographically distributed DSPs by orders of magnitudes by tuning some of the parameters. Surprisingly, this is even the case amid transcontinental delays.

In this paper, we show that it is possible to achieve performance comparable to the datacenter environment (where RTT is close to 0 msec) when operating existing DSPs with inter-node delays of tens of milliseconds and a small load. This is possible by adequately tuning already popular and widely used big data analytics platforms rather than utilizing experimental systems that require huge investments for installation, debugging, and personnel training. We hope that our insights will help towards realizing delay-resistant geo-distributed analytics.

Our contributions can be summarized as follows:

- We study three popular big data analytics systems (DSPs), namely, Apache Storm, Spark, and Flink, and we comment on the fundamental inherent limitations of each architecture in wide-area network settings.
- We study the sensitivity of each system on network delay and we derive characteristic delays beyond which each system has performance issues. We also confirm our findings by evaluating emulations of DSPs in Europe, U.S., and around the Globe.
- We propose ways to tune each and every system to achieve performance in the wide-area setting comparable with that of a datacenter despite the geo-distributed site deployment.
- We report that there is no single winner DSP in all settings, however, significant improvements are possible for all the systems we consider in our study.

The remainder of this paper is organized as follows. Section II surveys geo-distributed data analytics. The architectures of distributed stream processing systems come in Section III and we report the impact of the network delay on big data analytics in Section IV. Section V discusses the system- and network-related parameters of DSPs while Section VI reports the results of our evaluations. Finally, Section VII concludes our work.

II. Related Work

We categorize geo-distributed data analytics into two subcategories: (i) Wide-area batch processing [21], [22], [23], [24], [25] and (ii) Wide-area stream processing [19], [20], [26], [27], [28], [29], [30], [31], [32]. We also discuss related work that characterizes the dynamics of wide-area network latency and investigates the impact on the the performance of DSPs, as well as related work on the streaming benchmarks for DSPs in category (ii) that we will focus on in this work.

Wide-Area Batch Processing: In this case, the input data is available prior to the query execution. The goal is to schedule the query execution to either minimize the execution response time or wide-area network (WAN) bandwidth usage. WANalytics [25] is a system that pushes the computations to edge datacenters to optimize the workflow execution and minimize WAN bandwidth usage while replicating data when needed. Iridium [22] achieves low-latency query execution by optimizing the placement of tasks and data while dealing with the capacity of WAN links. CLARINET [21] and Tetrium [23] consider query optimization for WAN bandwidth availability and resource heterogeneity, respectively. These research works are not suitable for stream processing scenarios that focus on the response time of each input data rather than a batch of data. The work in [24] proposes a framework to balance the associated costs like bandwidth, storage, computing, migration, and latency with processing data across geo-distributed datacenters. The work of [33] focuses on multi-path routing optimization for the flows of data analytic jobs to better utilize the inter-datacenter links' capacity. SDTP [34] considers the dynamicity of the network bandwidth and computation parallelism to optimize job response time. However, the work of [35] showed that it is possible to put regulation on the data movement in geo-distributed datacenters. All the above works assume that the size and location of data are available before the query execution. Thus, query execution can be planned, accordingly taking into account the available resources. In summary, none of these works deal with the volatility of input data in real streaming applications that we consider in our study.

Wide-Area Stream Processing (WASP): In WASP scenarios, the input data comes from many resources like sensors or Internet of Things (IoT) devices and the data has to be processed in-situ due to several constraints like privacy and WAN limitations. In such scenarios, the computing resources are also geographically distributed across datacenters to process the input data. The streaming query is continuous, i.e., input data is continuously fed to DSPs and it has to be processed immediately [36]. AWStream [20] adapts the performance of the system to network changes using different degradation functions aiming at low-latency and high accuracy query execution. For example, if the network suffers from congestion it decreases the input data rate. Sol [29] improves the overall resource utilization of query execution in Apache Spark by adapting the system performance to network conditions such as latency and bandwidth. It proposes a federated execution engine that is aware of underlying network conditions. For instance, by decoupling computations from communications, Sol can scale down the task's CPU requirements to meet the available communication bandwidth of bandwidth-sensitive applications. Nevertheless, non of these works consider the popular DSPs to understand the impact of high gradually increasing WAN link latency on the performance of those systems in geo-distributed settings. Li *et al.* [27] consider the WAN bandwidth limitation to minimize the query execution time using a flexible routing mechanism for micro-batches in Apache Spark. Sana [28] tries to find the shared operations among queries in a multi-query scenario to run them only once by considering WAN bandwidth. This way, it aims to achieve high throughput and low latency during the query execution. However, the above works consider Apache Spark for the performance evaluation while ignoring the other popular stream processing systems, e.g., Apache Storm and Apache Flink. The work in [31] proposes a query execution mechanism adaptive to network conditions while considering the quality/accuracy of the results. JetStream [19] considers the WAN bandwidth limitations when running the query. It strives to achieve this by making a trade-off between the quality of the query reply and the system performance. Despite considering WAN bandwidth limitations, JetStream is not a popular stream processing system in use. Kumar *et al.* [26] study the trade-off between WAN delay and traffic using a TTL-based mechanism for windowed aggregation. The performance of this system is just studied for Apache Flink and one network topology. However, the contribution of this paper considers more systems and networks.

WAN Link Measurements: Several studies have shown that the WAN latency varies over time [37], [38], [39]. The measurement of the Amazon *AWS* [38] shows that it is common for flows to change the path in intervals of 10 seconds. Consequently, the traffic flows experience diverse performance due to the path changes and delays. The work in [37] proposes a tool to localize the performance degradation of cloud services due to the WAN latency of Azure whose datacenters are globally distributed. SWAN [39] is a system that improves the efficiency of inter-datacenter WAN to carry more traffic by coordinating the different services of providers.

Streaming Benchmarks for DSPs: The Yahoo! streaming benchmark [40] generates streams of data to measure the throughput and latency of three widely used engines, namely, Apache Spark, Apache Storm, and Apache Flink. Karimov *et al.* [41] propose a benchmark for the three widely used stream processing engines like those of Yahoo! benchmark to measure the performance of each system for windowed operations, i.e., windowed aggregations and windowed joins that are two main operations used to monitor user feeds from different sources. Lopez *et al.* [42] study the performance of the open-source stream processing systems for node failure scenarios. All the above benchmarking works focus on the performance of the systems where latency is negligible or not an issue. Hence, none of the studies that used the above benchmarks assess the performance of DSPs in scenarios where the performance bottleneck is the latency, as is the case in the WAN environment.

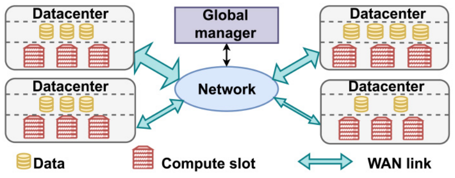


Fig. 1. Big data platform computational model distributed across several datacenters. The user submits analytics job to the global manager to be processed by the computing resources on each datacenter connected through WAN links with diverse delay and bandwidth.

III. Big Data Architectures

In this section, we present the general computational architecture adopted by three of the most popular big data analytics systems, namely, Apache Storm [1], [2], Apache Spark [3], [4], and Apache Flink [5], [6]. We also discuss the implementation differences among the three systems. Each of the systems follows a master-slave architecture where the master node executes the tasks on a set of worker (slave) nodes. We use the term "worker" for a node that executes streaming tasks. Each worker offers a set of task slots, i.e., access to the available resources such as CPU and memory, to execute tasks. The number of available task slots depends on several factors, e.g., the amount of resources or the number of currently running tasks per worker.

A. Big Data Computational Model

Big data platforms are distributed in nature, as they span multiple racks in a datacenter, multiple datacenters within a country or around the globe. The datacenters are connected via WAN links having different properties like delay and bandwidth, see Fig. 1. Each of the datacenters runs the same compute daemon. In this model, the input is streaming data that comes from different resources and is fed to the closest datacenter for processing. There is also a global manager that receives the user data analytic jobs– consists of an input, query, and an output– as input. Then, it translates them into platform-specific executable tasks. Each task is a basic unit of execution and the execution plan for a set of tasks can form a Directed Acyclic Graph (DAG). Each DAG has a set of vertices and edges in which each vertex represents an operation such as *map* or *reduce*, and each edge refers to an operation applied on a data tuple. Then, the global manager allocates resources to execute the query in the stream of data that is processed in each datacenter. Each query is a request for the results of processed information. For example, a query can ask for the number of clicks on a specific item over a window of 10 seconds on a retailer Website. The manager schedules the execution of the query based on the query requirements and the, typically heterogeneous, available resources at each datacenter. Examples of such available resources are CPU cores and memory. In the above setting, *response time* is the time that the system takes to execute the query. However, for the streaming scenarios

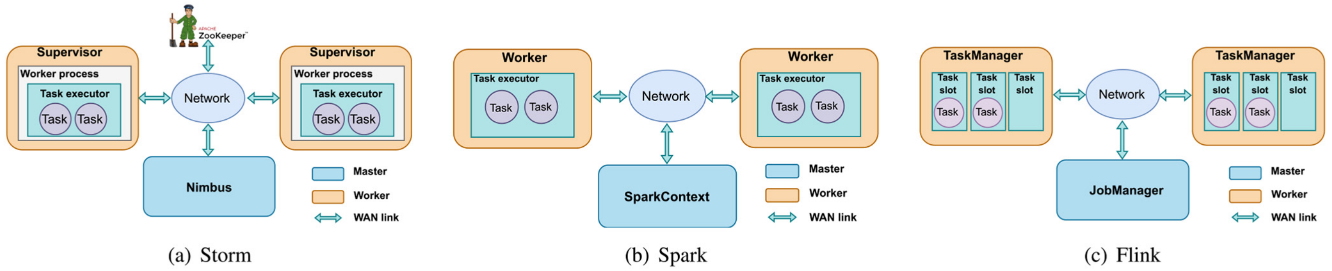


Fig. 2. Architecture and computate components for Storm, Spark, and Flink (2-node deployment).

response time specifies the time taken by the system to process an event or tuple (depending on the nature of the application). Furthermore, in this model, the incoming streaming data on each datacenter is first processed locally and then the intermediate results are exchanged among all sites or transferred to a central location for responding to a user query. Thus, the delays of the WAN links negatively impact the *response time* of query execution which yields an increase of the overall *response time*. In addition, the heterogeneity of the involved resources and the input data plays an important role:

WAN Heterogeneity: For datacenters that are geographically distributed, WAN links are typically the bottleneck, due to high delay, asymmetry of bandwidth, or congestion [22].

Compute Heterogeneity: Each datacenter has heterogeneous computing resources to run the streaming queries. There are several reasons for heterogeneity like the local computational requirements, the computational installations and technology, the level of investments, constraints imposed by energy, cooling, and the size of the datacenter [23]. Furthermore, the availability of resources to execute the streaming queries depends on the currently running tasks.

Input Data Rate Heterogeneity: The input data rate can vary in a streaming scenario depending on the number of users or sessions that has a significant impact on the generated data rate by a stream processing platform. Depending on the generated data rate the system uses the compute resources to execute the streaming queries and subsequently the WAN links capacity to transfer the processing results.

We summarize that depending on the nature of big data applications, network-related parameters such as delay play a significant role. For example, when a huge amount of data should be transferred from one datacenter to another one, the cost of data transmission through WAN links is more important [24]. While for delay-sensitive application scenarios such as business transactions, the latency parameter dictates the performance metric. However, we conclude that WAN links delay plays the dominant role in the big data streaming scenarios.

B. Apache Storm

Apache Storm [1] is a distributed computing system that processes the streams, i.e., an unbounded sequence of data tuples in a stateless manner, of data without keeping any information, e.g., state, about them. Fig. 2(a) shows the computation components of Storm architecture. Apache Storm has a master node, called `Nimbus`, and a set of worker nodes, each one called `Supervisor`. Apache Zookeeper [43] manages the Storm cluster and stores the states of master and workers.

Query Execution: The user submits the query to the master node that creates the corresponding Storm topology consisting of the DAG of the query. Each Storm topology shows the processing logic of the query and links between the worker nodes to show how the streams of data to be processed. The master generates the tasks and distributes them to the workers for execution. The worker executes the tasks and uses the Netty framework [44] to exchange the intermediate data.

C. Apache Spark

Apache Spark [3] is designed for both batch processing applications and streaming applications. For streaming applications, there is a streaming engine that generates a stream of data from data in a batch form. Fig. 2(b) illustrates the computation components of Spark architecture. The architecture consists of a master node and a set of workers. There is a driver program residing inside the master node which dynamically creates a `SparkContext` for each Spark application. `SparkContext` performs the main functionalities of Spark to run the user applications. Furthermore, it allows the applications to access the Spark cluster. The Spark driver has other components like DAG-and task schedulers which are in charge of translating the user-written code to a set of tasks to run them on the workers. Each Spark worker has a `taskExecutor` being in charge of executing the tasks.

Query Execution: To run a query in Spark, the user submits the query to the master node. The master node creates a Resilient Distributed Dataset (RDD) to start the task execution. RDD is the fundamental data structure, i.e., an immutable collection of objects, of Spark and distributed among different nodes in the Spark cluster. Spark translates the RDD transformations into a DAG. Then, Spark submits it to the DAG scheduler that computes a DAG of stages, i.e., a physical unit of execution, for each task and finds the minimal execution schedule for each job. Then, it submits the job to the task scheduler, which is in charge of sending them to the cluster, running them, and retrying if there are failures. To execute the task, the `SparkContext` and driver interact with the Spark cluster manager to select a set of workers. Each worker runs its task executor which has a set of task slots. Each task slot can run a set of specific tasks. The workers of Spark use the Netty framework [44] for data exchange.

D. Apache Flink

Apache Flink [5] is a real-time stateful stream processing system. It supports both stream and batch processing applications. The input data is generated as the stream of events in Flink. Fig. 2(c) presents the computation components of Flink architecture. The architecture has a master node and a set of workers. The master node in Flink context is called `JobManager` while each worker is known as a `taskManager`. Each worker has a set of task slots to run a pipeline of parallel tasks. A pipeline consists of multiple parallel tasks such as Map/Reduce [45].

Query Execution: The user submits the query to the master for the execution on the cluster. The JobManager receives the query in a JobGraph form, i.e., similar to DAG concept in Storm and Spark. Each JobGraph is a data flow representation in a graph form that has operators and intermediate results. Each operator has properties such as the code to execute and the parallelism. Also, Flink adds the necessary libraries to the JobGraph to execute the task. The Flink scheduler checks the available pool of task slots provided by the workers to run the tasks. The Flink scheduler submits the scheduled tasks through the master to workers. The workers execute the tasks based on the plan given by the master. The workers of Flink use the Netty framework [44] for inter-worker communications such as exchanging the intermediate results.

E. Comparison

In this section, we briefly compare the main features offered by each DSP. All systems follow a master-slave architecture.

Processing Model: Apache Storm has a pure stream processor without batch capabilities. It can process real-time data for real-time applications like fraud detection. Apache Spark supports both batch and stream processing. Spark can be deployed as a stand-alone cluster and with Hadoop clusters. The streaming task in Spark is executed as mini-batches. To do so, Spark receives a stream of data and buffers them into the memory of workers. Then, the Spark engine runs short tasks, which are in order of tens of milliseconds, to process the batches and produce the output results to other systems. This kind of mini-batch processing is suitable for many applications. However, Spark is not the best choice for real-time applications like fraud detection. Apache Flink has a unified framework for stream and batch processing. Data records are immediately shipped from the producer to the receiver after collecting them into network buffer [46]. Flink processes events based on their entry timestamp to the system which helps the system to maintain their orders. It uses the orders to identify events in case of late arriving to take a suitable action [47].

Storm and Flink have different processing semantics to run the streaming tasks and keep the state of the system. Spark and Flink provide exactly-one processing semantic of the tasks meaning that each incoming event affects the final results exactly once, while Storm provides at least one processing semantics. This semantic ensures that even in the case of failure, i.e., the machine or the software, there is no duplicate data or data that needs to be processed more than once. However, Storm with Trident API [48] achieves exactly-once processing semantics but it comes with an extra cost like latency because it is built on top of the Storm core library. Both Flink and Storm can provide low-latency execution depending on the WAN. However, Spark streaming response time depends on the batch interval.

Message Delivery Semantics: Storm guarantees at-least-one delivery of messages in the system meaning that if the failure happens in the system Storm may deliver a message twice. Storm uses record-level acknowledgments that mean each worker checks the acknowledgment of the receiver for the task execution to commit the processed record. The number of record acknowledgments in Storm by default is 2. Spark offers exactly one message delivery which means that the system assures that no data is lost. This is also the case even if there is a failure in the Spark cluster. Flink provides exactly-one delivery of events by maintaining checkpoints at regular intervals. Maintaining checkpoints allows Flink to recover the state and positions of data in the stream. For WAN scenarios, the second message delivery slightly affects Storm response time and causes higher execution latency since the communication channel is affected by long Round-Trip Times (RTTs).

Fault Tolerance: The Storm daemons, Nimbus, and the Supervisors are stateless and fail-fast. This means that Storm will try to restart them in the case of failure. Spark provides fault tolerance using RDDs. To do so, Spark tracks the data lineage information and automatically rebuilds them upon failure. Then, the retrieved data is replicated among the workers of the cluster. Flink uses checkpoint and state snapshots to achieve fault tolerance. Flink uses a variant of the Chandy-Lamport algorithm for this purpose in which it checks the data streams for the marker checkpoint to keep the internal state of the system. Upon receiving the marker Flink commits all the processed records.

The above differences affect the performance of each system under stress or loss of messages due to drop of packets in the network, e.g., when there is congestion, multiple hops, or when sites are far apart. All three DSPs leverage the Netty framework differently for data exchange among the worker nodes. Depending on how the job schedulers of each DSP employs Netty, we may see a different impact on the execution latency of the queries in WASP scenarios. Therefore, job schedulers play a critical role in DSPs and we will detail them in Section V-A.

IV. Delay Sensitivity

In this section, we investigate the impact of network delay on the performance of big data analytics systems. We first present in detail the benchmark we utilize as well as the experimental testbed. Then, we report what is the impact of network delay on the performance of (the vanilla version) the three popular DSPs we consider in our study, namely, Apache Storm, Apache Spark, and Apache Flink under different topologies.

A. Benchmark

The Yahoo! streaming benchmark [40] is a popular streaming benchmark that has been used in other research studies related to the performance evaluation of big data analytics platforms [28], [31], [32], [36], [49]. The benchmark measures the performance of DSPs, e.g., Apache Storm, Apache Spark, and Apache Flink. The main idea of this benchmark is to emulate an advertisement analytics pipeline in the DSPs and

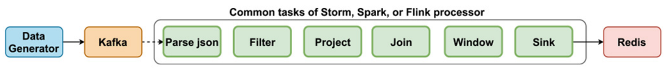


Fig. 3. The extended advertisement analytics pipeline of the Yahoo! streaming benchmark [50].

assess the performance of the various systems. The query of the benchmark simulates a number of advertising campaigns in which each one gets a set of advertisements. Each advertisement/event has the following fields: *user_id* (UUID), *page_id* (UUID), *ad_id* (UUID), *ad_type* (string), *event_type* (string), *event_time* (timestamp), *ip_address* (string).

The event generator of the benchmark generates events with a timestamp $t_{initial}$ and truncates them to a specific number to determine their belonging campaign. Additionally, each event carries a timestamp that specifies the last timestamp, i.e., t_{last}, it is updated by a DSP. The benchmark computes the event latency after processing the event by a DSP. To do so, it first calculates the difference of t_{last} and $t_{initial}$. Then, it deducts the obtained value from window duration, i.e., 10 seconds, to write as the latency of processed events [32].

The main task of the benchmark is to read data from Apache Kafka [51] and identify the relevant events. Then, it computes the number of viewed events and their execution time in 10 seconds time window. Finally, the results are stored in Redis [52]. Kafka and Redis are the performance bottlenecks in this benchmark. These bottlenecks are eliminated in the extended version of the benchmark [50] in which Kafka and Redis operations are shifted to the outside of the computation parts of the benchmark (Fig. 3). The Yahoo! streaming benchmark emulates the common streaming pipeline of most streaming analytics scenarios where they get streams of data, perform computations, and produce the results [40]. We make our source code publicly available here.[1]

B. Experimental Testbed

Our testbed consists of 22 VMs, and each equipped with AMD Opteron Processor 6272 running at 2.1 GHz with 16 cores. We use a dedicated VM for the master and run the experiments for scenarios with an equal number of workers, i.e., 2, 4, and 8, with 8 GB of RAM. The Kafka VMs have the same CPU but 16 GB of RAM. In addition to the number of VMs for the workers, we use a dedicated VM for Redis, up to 11 VMs to run Apache Kafka to generate the desired input data, and one VM for Apache Zookeeper [43]. We apply the recommendations of Yahoo! benchmark [40] to set the values of the parameter in each DSP. We use 6.7GMB of heap memory for the workers in Storm, Spark, and Flink as this is the maximum usable heap in our worker VMs. We also use 6 task slots per `taskManager` in Flink. All the Kafka VMs are connected to the workers with a link without applying RTT. All the workers have connectivity to each other and also to the master VM. We use a set of Kafka VMs to generate the desired input data rate.

We report the 99^{th} percentile execution latency of the query by varying different parameters as the slowest completion dictates the performance of the overall execution time [53]. The execution latency is measured as the difference between the time that an event is emitted at the sink and the time it was generated by the data generator [36]. We use Debian traffic control (*tc*) queuing disciplines tool [54] to apply inter-node delays. The goal of experiments is to understand the impact of inter-node RTTs on the performance of the DSPs without putting a massive amount of data into the network. The former case demands high link bandwidth and more computations that are out of the scope of this work. The recent report by Akamai shows that the available bandwidth of public clouds is less than 10 Mbps [55]. Therefore, we avoid over-utilizing the available WAN links bandwidth. Each experiment lasts 1,500 seconds and is repeated 5 times and in total we run more than 1,000 hours of experiments. We report the average of runs since the differences among the repetitions are very small and invisible in the figures. We first measure the performance of each DSP using the vanilla version. Specifically, we set the record acknowledgment of Spark to 2 and the batch interval of Spark to 3k msecs.

C. Full Mesh Topology Delay Sensitivity

To understand the delay sensitivity of different big data analytics systems, we first investigate a topology that is under our control, namely, the symmetric full mesh topology. This topology has the advantage that we can change the parameters of pairwise workers delay, the data rate, and the number of workers on demand, and thus, assess the impact of different parameters on the delay sensitivity. The nodes in the full mesh topology are placed at the same distance from each other resulting in the same RTT delay.

We first assess the performance of the three DSPs on the full mesh topology by varying the RTT from 0 to 100 msecs, and for topologies with 2, 4, and 8 workers. The data input rate is moderate at 22k events/second since all DSP can process this rate similar to a datacenter cluster with 2 workers. Also, each Kafka VM can generate events up to 15k events per second without putting additional delay by Kafka to the events before processing by the DSPs but for the first set of experiments, we use less load to avoid any possible side-effect of Kafka. We use 2 Kafka VMs for data generation, and each one generates 11k events/second. The aggregated bandwidth for the generated events can be achieved even without violating the bandwidth limitations of the WAN links of public clouds. The default record acknowledgment, i.e., 2, is used for Storm, while the Spark experiments have been conducted using 3k msecs of batch interval. We call this setting vanilla version of DSPs and the detail of tuned configurations are reported in Section V-B.

We report the 99^{th} percentile execution response time (latency) for all three DSPs in Fig. 4. With increasing RTT the results show that the latency in Storm increases, especially when the RTT is around 30 msecs. The latency of Spark and Flink increases very slowly with the RTT. However, the latency of Flink is about one order of magnitude smaller than the latency of Spark. The increase in the number of nodes from 2 to 4 seems to have a minimal impact on the performance

[1]https://mostafaei.bitbucket.io/Publications/WASP/

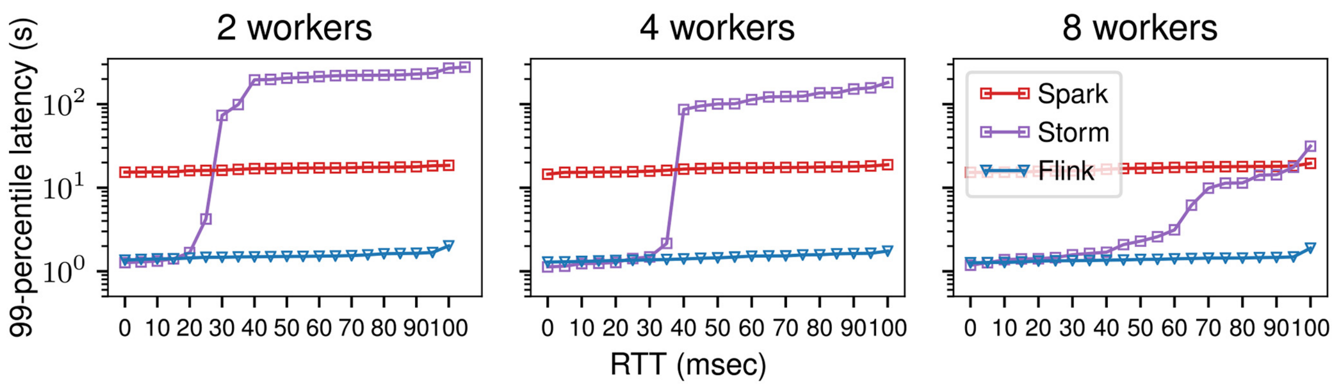
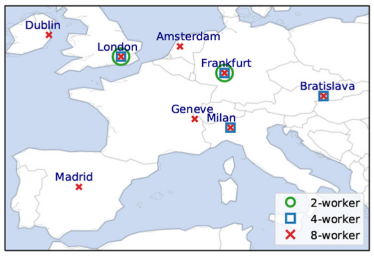
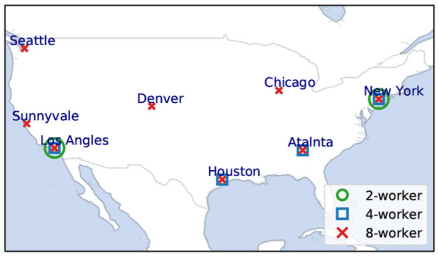


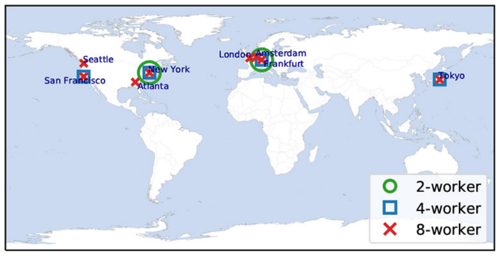
Fig. 4. Full mesh Topology: The 99 percentile execution response time using 22k events/second and RTT values varying from 0 to 100 msec.



(a) EU



(b) US



(c) Global

Fig. 5. The location of workers on the EU, U.S., and global topologies for scenarios with 2, 4, and 8 workers.

of both Spark and Flink. On the contrary, the performance of Storm improves with additional workers, especially for RTT delays of more than 30 msecs where the impact of RTT is smoothed down. The best performance of a system, for a given value of RTT, is obtained when using 8 workers. The main reason for such results in Storm is that with the high number of workers, Storm can better distribute the input data into their buffers [56]. Similar results are obtained for the 95th and 90th percentile execution response time (not shown) to confirm that all the events are processed within the reported percentile latencies.

D. Delay Sensitivity of Geo-Distributed Worker Topologies

For a more realistic assessment of the impact of the pairwise node delays, we consider three topologies where the workers[2] are geo-distributed, namely, in EU, U.S., and global. We emulate the placement of the workers in Europe, the continental U.S., and around the Globe by selecting cities that are points of presence for many datacenter providers, e.g., Equinix [57], Amazon [58], Microsoft Azure [59], and Google Datacenters [60]. Table I summarizes the location of workers in the U.S., EU, and global topology with their average RTT delay, i.e., $\overline{D}$, and maximum RTT, i.e., D_M, among the workers. We collocate the master node of each DSP with the first node, i.e., the first city name in each cell of Table I, in all topologies. The placement results in having zero-latency among the master and the first worker. We obtained the inter-node delays for the EU network by contacting the GEANT network administrators that operate measurement points in all the major cities in Europe. For the U.S. pairwise delays, we rely on the backbone delays reported by AT&T [61]. The RTT values for the pairwise delays of the global deployment are taken from WonderNetwork [62]. Fig. 5 shows the locations of the workers on the three topologies on the map. Note that the 4-worker set is a superset of 2-worker, i.e., it includes the locations of the 2-worker set, and the 8-worker is a superset of 4-worker. This rule applies to all three topologies.

TABLE I
THE LOCATION OF WORKERS ON THE DIFFERENT TOPOLOGIES WITH DIFFERENT NUMBER OF WORKERS

Topo	**2-worker**	**4-worker**	**8-worker**
EU	Frankfurt, London	Frankfurt, London, Milan, Bratislava	Frankfurt, London, Milan, Bratislava, Madrid, Dublin, Amsterdam, Geneva
	$\overline{D}$=10.7, D_M=16	$\overline{D}$=15.7, D_M=30	$\overline{D}$=21.1, D_M=41
US	New York, Los Angles	New York, Los Angles, Houston, Denver	New York, Los Angles, Houston, Denver, Washington, Seattle, Sunnyvale, Chicago
	$\overline{D}$=40, D_M=60	$\overline{D}$=34.8, D_M=60	$\overline{D}$=35.6, D_M=35.6
Global	Frankfurt, New York	Frankfurt, New York, San Francisco, Tokyo	Frankfurt, New York, San Francisco, Tokyo, Seattle, London, Amsterdam, Atlanta
	$\overline{D}$=79.4, D_M=79.4	$\overline{D}$=134.9, D_M=221.8	$\overline{D}$=119.9, D_M=241.98

We repeat all the experiments that we described in the previous section (full mesh topology) for all three deployments, namely, EU, U.S., and Global. We present a summary of the results in Fig. 6. We consider the case that there is no significant pairwise delay variance during our experiment and we report the 99th percentile of the execution time.

Across the board, Flink performs better than the other systems. In Europe, where the pairwise delay is relatively

[2]We use the terms workers and location interchangeably.

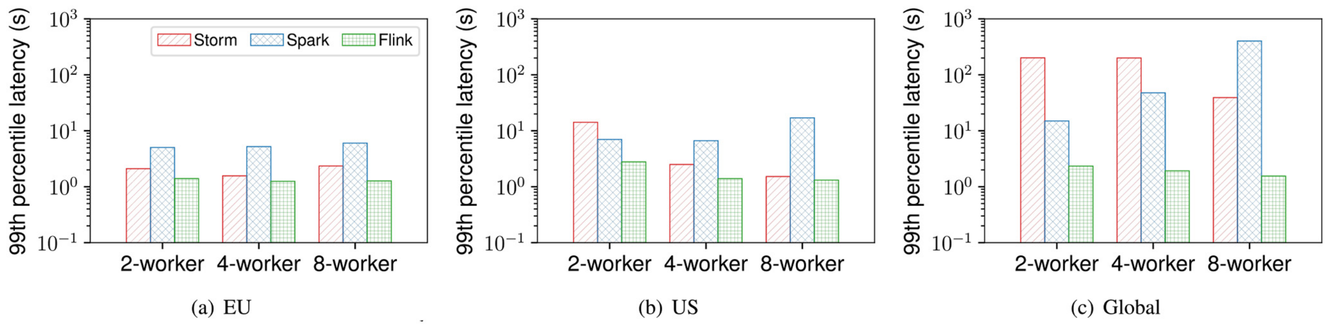


Fig. 6. Geo-distributed worker topologies: 99^{th} percentile query response time of each system with input rate of 22k events/second using 2, 4, 8 workers.

small the performance gap between Flink and the other two systems is smaller (see Fig. 6(a)). It is also worth noting that the impact of the number of workers is in general relatively small. In the U.S. deployment, the increase in the number of nodes has a positive impact on the performance of Storm and Flink, but not on Spark (see Fig. 6(b)). Note also that the average pairwise delay for the set of cities we selected does not go below 30 msecs, thus, the performance of Spark is not expected to improve since we observed this behavior in the results of full mesh topology in Fig. 4. In the global setting, where the pairwise delays are significantly higher than in Europe and the U.S., the latency of Storm and Spark are orders of magnitude higher than this of Flink (see Fig. 6(c)). Again, the increase in the number of workers has a positive impact on Storm and Flink, but not on Spark's performance. The main reason for such results relies on the nature of TCP connections among the workers that suffer from long RTTs. In such scenarios, if the producer and consumer work at the same rate, the consumer slows down the processing due to long RTT since it has to wait for workers to consume the data. Therefore, the buffer of the consumer may crash due to the bounded buffer memory of the consumer or exhausting the memory of workers. Generally, this trend can happen in scenarios in which the producer is faster than the consumer [63]. Furthermore, in the case of Spark, we have two types of buffering, i.e., one comes from the micro-batching and one from consumer buffering, thus, its performance suffers more.

V. Delay-Resistant Architectures

In this section, we discuss the system- and network-related parameters that impact the architecture of DSPs and discuss how these can be tuned towards realizing delay-resistant geo-distributed analytics using these DSPs. We also discuss the possible impact of the tuning of design parameters on the performance of DSPs in WAN scenarios.

A. Deep Dive Into DSP Schedulers

We first discuss one of the fundamental components of all three DSPs, the scheduler. The gained insights will help in tuning the system parameters and understanding the impact of this tuning on the performance of DSPs.

1) Storm Scheduler: Storm partitions the input data into small chunks that have to be processed by the tasks. It executes the streaming tasks using the concept of topology. Each Storm topology consists of a computation graph that determines the logical execution of the input data. Each Storm topology has two components, namely, Spouts and Bolts. The *Spout* is the source of the stream which emits the input data into the topology by reading the data from external resources. All the processing operations like map, aggregations, etc. are executed using the *Bolt* component. Storm offers some built-in stream grouping to group the set of tasks in topology and runs them together on a worker. Depending on the number of Spouts and Bolts in a Storm topology, the inter-connection among the workers varies. There are different threads for each Spout and Bolt component that communicate the execution of a task to the workers of a Storm cluster. However, Storm uses a single TCP port, called virtual port, on each worker for incoming messages. The default scheduler of Apache Storm allocates executors to all available slots evenly, otherwise, it uses round-robin assignment. Placing the workers in WASP scenarios impacts the execution latency of Storm due to the link delay and distribution of Spots and Bolts in different locations.

2) Spark Scheduler: Spark runs the tasks using an instance of `SparkContext`. Each `SparkContext` leverages an independent set of execution processes to complete the tasks. Each Spark application reserves a set of resources to execute the tasks and holds them until the end of that specific task execution. Spark by default runs jobs in First-In-First-Out (FIFO) fashion. Since Spark 0.8, it is possible to configure it in a round-robin manner to fairly use the cluster resources among the jobs.

To run a task, Spark creates an operator DAG from RDD objects which includes reading data from input. The DAG is submitted to the DAG scheduler that splits it into stages of tasks. Spark submits a stage to `TaskScheduler` when it becomes ready for execution. The `TaskScheduler` launches the task via the cluster manager to run on the worker. Each worker has a set of threads to execute the task. Therefore, depending on the number of RDD objects in the cluster and their internode delays in WASP scenarios, the performance can suffer from WAN link delays.

3) Flink Scheduler: Flink uses the concept of pipeline to assign the tasks into task slots offered by each worker. Each pipeline consists of a set of consecutive tasks like MapReduce. Each worker can run multiple parallel tasks. Running pipelines in the workers reduces the number of communications with other workers because, typically, they are running at the same worker. Flink defines `SlotSharingGroup` and `CoLocationGroup` to specify which tasks must be placed

TABLE II
TUNABLE PARAMETERS IN BIG DATA ANALYTICS SYSTEMS

Parameter	**Storm**	**Spark**	**Flink**
Parallelism	✓	✓	✓
Record acknowledgment	✓	✗	✗
Batch interval	✗	✓	✗
#TCP connections	✓	✓	✓

✓annotates support, ✗annotates no-support by the DSP.

in the same worker. A side benefit of this mechanism is that it reduces the communication with other workers placed at geographically distributed locations suffering from link delays. Apache Flink allocates all resources at once, called Eager. However, since version 1.12, Apache Flink leverages *Pipelined Region Scheduling* for the assignment. This strategy finds all the set of tasks that are connected via pipelined data exchanges and runs them together.

Flink uses a single TCP connection if different tasks of a worker should interact with a set of different tasks of another worker. For example, consider the two workers A and B, where each of them has two subtasks interacting with each other. Flink uses a single TCP connection to exchange data between workers A and B. Following this way results in opening fewer TCP connections that may suffer from WAN links delay in geo-distributed scenarios. Flink also leverages credit-based flow control to assure that the receiver can handle whatever the sender transmits. On the sender-side, this credit specifies the availability of network buffers at the receiver. Each communication channel has its own set of exclusive buffers to store the intermediate data.

B. DSP-Specific Parameters

Each DSP implementation has a set of parameters for the features it offers and for tuning the scheduling and execution of tasks. These parameters affect the performance of the system. Table II summarizes the parameters in Storm, Spark, and Flink that are relevant to our study. We briefly explain each of them in the following.

Parallelism: The degree of parallelism specifies how many parallel instances of a task can be executed on the workers of each DSP. For our experiments, we use the same degree of parallelism in all three DSPs depending on the number of worker nodes. By default, all DSPs run the task with single parallelism which may not use all the computing capacity of the workers. Therefore, depending on the number of workers and their capacity, the right parallelism value should be selected. Not that a higher degree of parallelism can degrade the performance due to the overhead introduced by different instances of tasks.

Record Acknowledgment: Storm uses record acknowledgment to guarantee the message passing inside a Storm topology. However, this acknowledgment can be disabled using the framework configuration file before submitting a Storm topology for execution. Note that the record acknowledgment is used for flow control but not for processing guarantees [40]. Since Storm uses `at-least-once` processing semantics, disabling the acknowledgment will not impact the processed data. Other parameters such as the number of workers or executors can be also tuned and we used similar settings in all DSPs [41]. In WASP scenarios, disabling acknowledgment improves the performance of Storm by adding less communication overhead over delayed WAN links without impacting the processed events.

Batch Interval: Spark stream engine generates streaming data using the concept of micro-batch. This parameter can be tuned using the batch interval as input to generate the desired batch of data. The number of tasks in each Spark receiver per batch is approximately equal to $\frac{\text{batch interval}}{\text{block interval}}$. Selecting the right batch interval depends on several parameters such as the number of cores per worker or cluster size [64]. There are no optimal settings for the batch interval, and it has to be obtained empirically depending on the use-cases. In WASP scenarios, the batch interval should be carefully selected due to the additional delay imposed by the WAN links. A small batch interval can lead to the communication overhead and large ones can delay the execution latency.

Number of TCP Connections: Each DSP leverages the Transmission Control Protocol (TCP) for data transmission in a cluster. When placing the cluster of DSP at a geo-distributed location, the inter-node delay of workers in the cluster and the number of TCP connections affect the execution latency of tasks in WASP.

The number of workers has a determinant impact on the performance of real-time stream processing systems like Storm and Flink. Executing the tasks with a high number of workers improves the overall performance of the systems in many scenarios. However, it might add additional overhead due to handling a large number of connections among the workers. Thus, it is imperative to take this trade-off into account when running our experiment.

VI. PERFORMANCE EVALUATION OF DELAY-RESISTANT SYSTEMS

In this section, we outline our goals in designing the experiments and assessing to what extent it is possible to improve the performance of delay-resistant DSPs by properly tuning operational parameters that we described in the previous section (see also Table II). We run the query for again 1,500 seconds and the results are averaged over 5 different runs. We report the 99th percentile execution response time (latency) similar to Section IV.

A. Delay Resistance

The DSPs have been designed to perform the computation of streaming applications in a cluster network located on a rack in a datacenter. Inside a datacenter, delays are low, even sub-second. However, for reasons we highlighted in the intro, namely, volatile data sources and regulation, DSP are becoming increasingly popular for geo-distributed analytics [36]. Thus, the performance of the DSPs in the new environment needs to be carefully examined particularly regarding the impact of wide-area network (WAN) links delay on the execution latency of each system. The goal is to find the breaking point of each DSP when running them at geographically distributed locations where each site is connected via WAN links. To achieve this goal, we first set up a full mesh topology

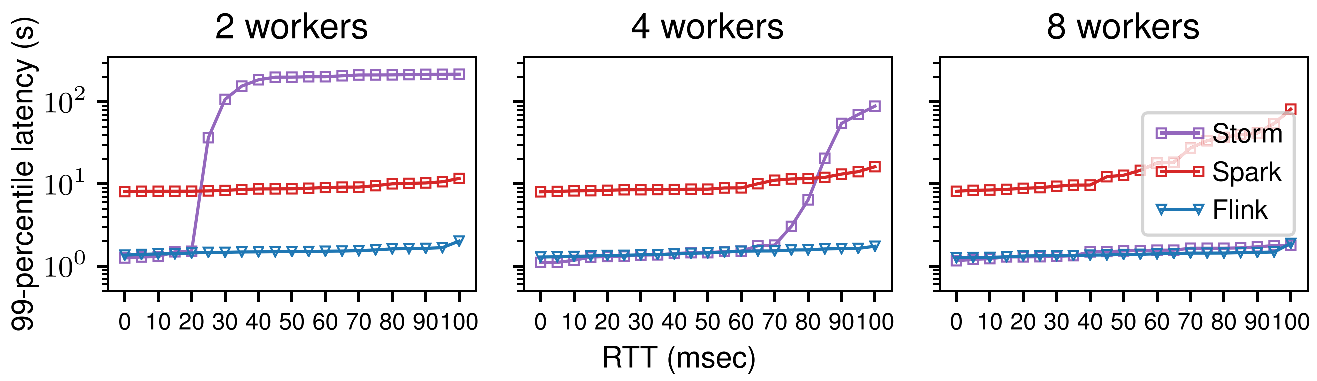


Fig. 7. The 99^{th} percentile latency of all systems using 22k events/second and by varying the RTT and tuning the parameters, i.e., Storm with NoACK and Spark with 5k msecs of batch interval.

and then, gradually increase the inter-node delays to find the performance degradation point for each DSP (Section VI-B). Ideally, we would like to tune the DSPs to be delay resistant in a geo-distributed setting, i.e., achieve throughput and query execution time comparable with this when operating inside a datacenter. Then, we report the results for geo-distributed settings (Section VI-C).

B. Delay-Resistant DSPs on a Full Mesh Topology

We first evaluate the impact of appropriate tuning of parameters towards realizing delay-resistant systems on the full mesh topology.

1) Impact on Delay Sensitivity: We repeat the experiments for the full mesh topology, see Section IV-C, but we now disable record acknowledgment of Storm to eliminate its communication overhead for flow control. It is worth noting that by disabling the acknowledgment the ability of reporting failures is also disabled in Storm. Fig. 7 shows that disabling the acknowledgment improves the performance of Storm for 4 and 8 workers scenarios drastically. For 4 workers the degradation of performance takes place for higher RTTs than 30 msecs. For 8 workers the performance is delay-resilient, even for higher RTTs, similar to trans-continental.

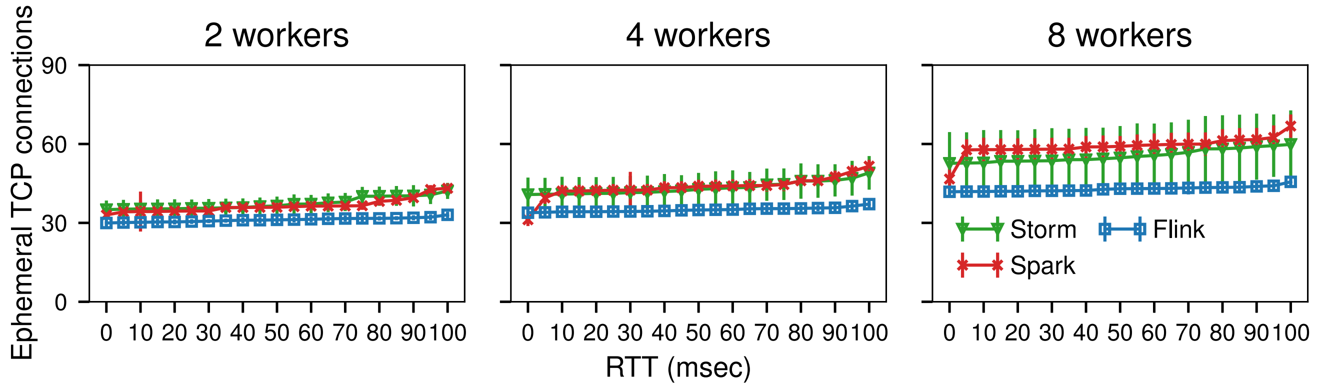
We also notice that the performance of Storm and Flink has a small variance as the RTT and the number of workers increase. Recall that all the results are obtained with 5k msecs batch interval. Our results also provide useful insights on the impact of the batch interval. Increasing the batch interval of Spark from 3k to 5k msecs slightly decreases the percentile latency. We also report the impact on throughput in Section VI-D.

2) Ephemeral TCP Connections: We now measure the number of ephemeral TCP connections for Storm, Spark, and Flink that are established between master and workers, as well as bilaterally between pairs of workers. We use the *socket statistics* tool [65] to count the number of established TCP connections during the experiment and report the average number of connections with standard deviations since they change over the time of running our experiments. Fig. 8(a) shows that the master VM in Flink establishes fewer TCP connections with workers to execute the query, while Storm and Spark establish a similar number of TCP connections. Fig. 8(b) shows that workers in Spark establish, on average, more TCP connections when the RTT delay increases. In the scenario with 2 workers, Spark workers establish two times more TCP connections as compared to Flink and Storm. This adds considerable overhead in communication, see Fig. 8. The reason is that Spark has no built-in TCP server to wait for the producer to buffer data since it works based on polling mechanism for TCP connection using its API libraries [66]. In the case of 2-worker scenarios, it needs to use more TCP connections to process the events. Storm and Flink workers establish the same number of TCP connections to execute the query across different settings we evaluated.

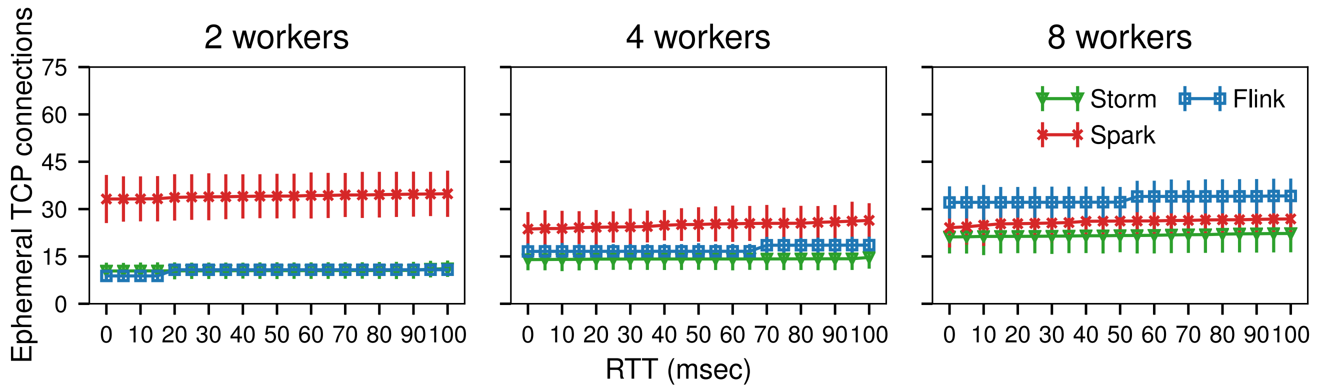
3) Stress Test: We perform stress tests to examine DSPs' performance when increasing input data rate and latencies.

Performance for Load Variation: We first keep the RTT fixed to 0 msecs and increase the input data rate in all DSP systems from 22k to 154k events per second. We measure the execution latency of all DSPs. We run the experiments for scenarios with 2, 4, and 8 workers. The goal is to understand the performance of the systems under higher load in a centralized cluster with zero inter-node delays. If there is an inter-node delay the impact of increased input rate will be even more severe. To perform this experiment we add more Kafka VMs to be able to generate the desired input load when needed. We generate at most 14k events per second on each Kafka VM to guarantee that no events are dropped. Fig. 9 shows that increasing the input data rate has less impact on the execution latency of Flink for various scenarios by changing the number of workers. Indeed, Flink's performance deteriorates only in very high input rates. On the other hand, the performance of Spark and, especially, Storm is very sensitive to the increase of input rate. The increase in the number of workers improves the performance of Spark, and, especially the performance of Storm.

Performance for RTT Variation: We now keep the input rate fixed to 55k events/second (a load that already stress some of the DSPs in some settings) and increase the RTT. Fig. 10 shows the execution latency of all three DSPs for scenarios with 2, 4, and 8 workers. The execution latency of all the systems degrades when the RTT is higher than 25 msecs regardless of the number of workers. All the systems can process the events with a similar latency when the RTT



(a) master



(b) workers

Fig. 8. Average ephemeral TCP connections established by (a) master and (b) workers in Storm, Spark, and Flink with 2, 4, and 8 workers on a full mesh topology.

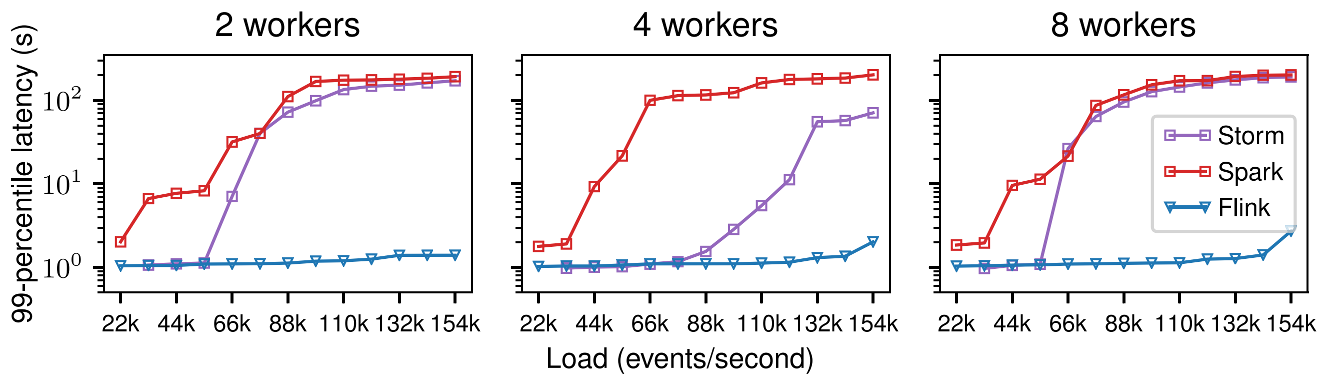


Fig. 9. Stress test: The 99^{th} percentile latency of all systems by varying the input rate and keeping fixed the RTT (zero msecs).

is higher than 50 msecs. However, for lower than 50 msecs, Flink's performance is better, especially when the RTT is below 50 msecs and the number of workers is 8.

4) Resource Usage: We also report on the resource usage of the VMs we utilized in our testbed using Debian vmstat tool [67]. After monitoring the recourse usage at each VM we conclude that the Kafka leader VM is the most resource hungry VM. This is to be expected as it writes the logs into the disk. The I/O operations are reported as the number block/second in which each block has 1KB size. Figs. 11, 12, and 13 report the physical resource usage of Kafka leader VM for Spark, Storm, and Flink, respectively, with an input rate of 22k events/second on the full mesh topology. The Kafka leader VM has very similar resource usage and I/O operations in Storm and Flink and it does not add impact the execution latency of them. Spark has a slightly higher number of I/O which adds additional delay to its execution latency. The resource usage confirms that all the workers have enough compute resources, i.e., CPU cores and memory, to process the incoming data stream on the full mesh topology. However, in WASP scenarios, the WAN

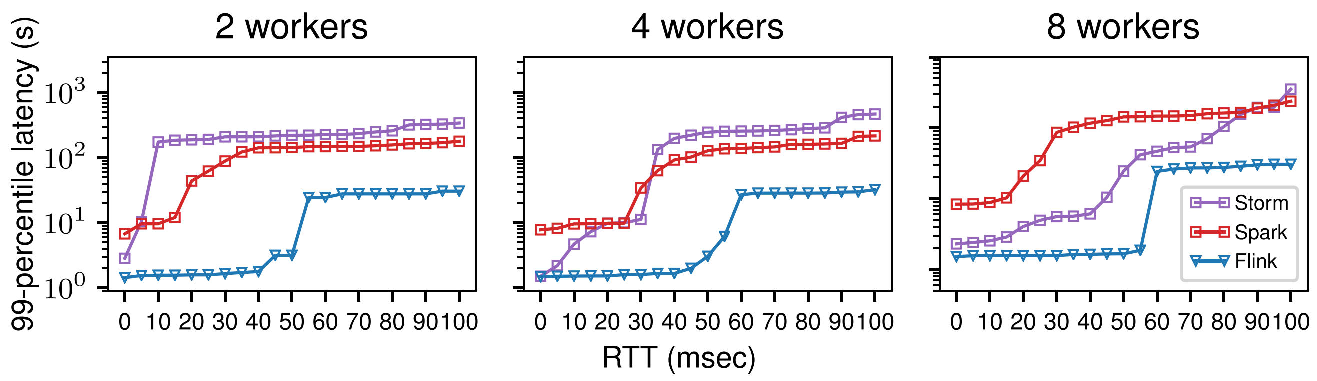


Fig. 10. The 99th percentile latency of all systems using 55k events/second and by varying the RTT; Storm with NoAck and Spark with 5k msecs of batch interval.

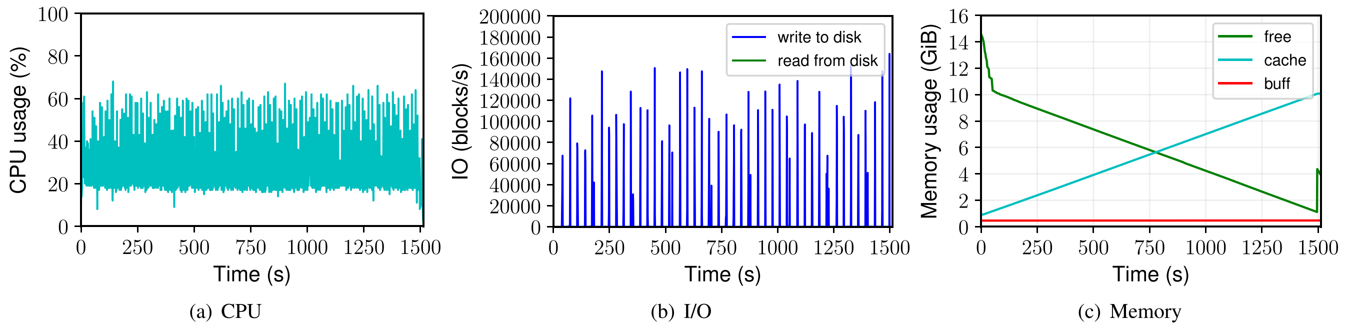


Fig. 11. Storm physical resource on Kafka leader for 22k events/second without ack.

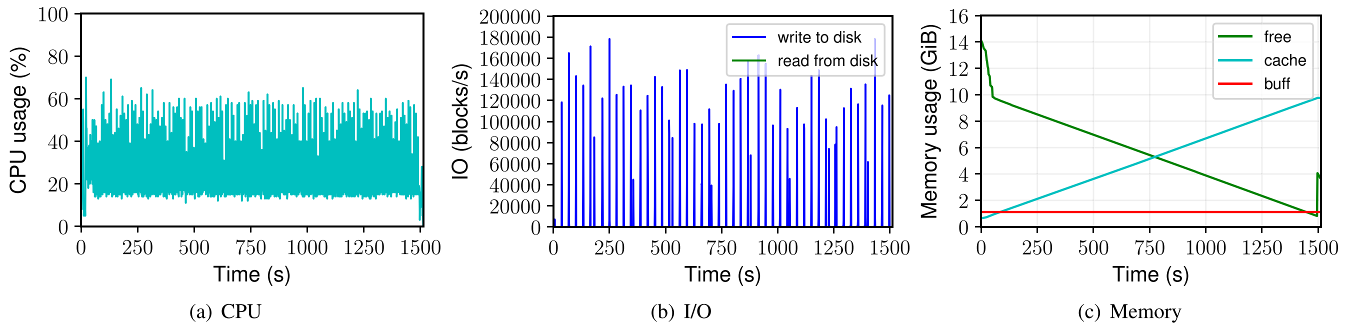


Fig. 12. Spark physical resource on Kafka leader for 22k events/second with batch interval 3k msecs.

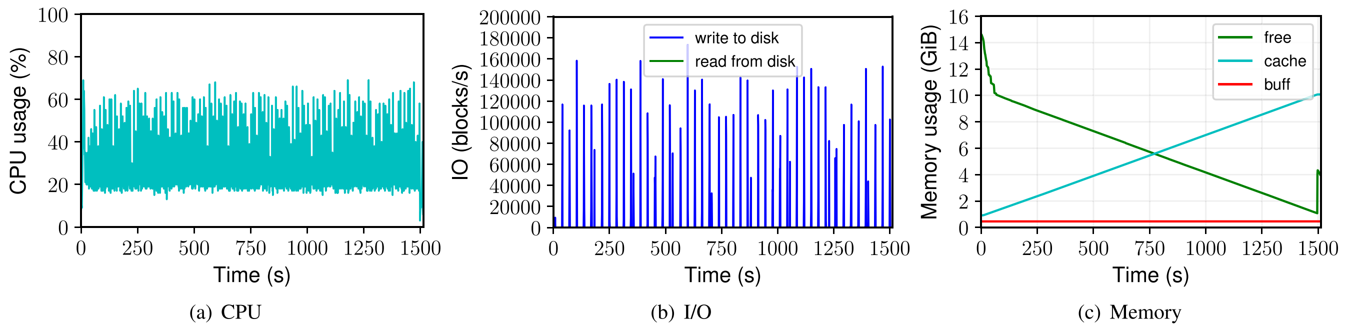


Fig. 13. Flink physical resource on Kafka leader for 22k events/second.

link delays play a determinant role in the response time of the systems. Furthermore, the CPU of the workers is under-utilized in WASP scenarios when the link delay is high and available bandwidth is limited [29]. We investigate this in the next section.

C. *Delay-Resistant DSPs on Geo-Distributed Topologies*

We now repeat the experiments (we performed on the full mesh) on the three realistic topologies, namely, U.S., EU, and Global, by appropriately tuning the three DSP systems.

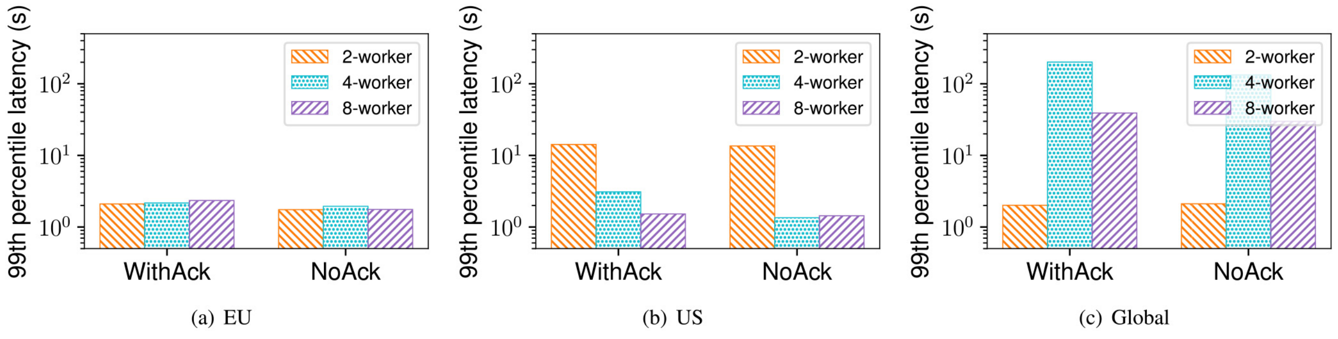


Fig. 14. The 99^{th} percentile latency of Storm on the EU, U.S., and Global topologies with input rate of 22k events/second using 2, 4, 8 workers.

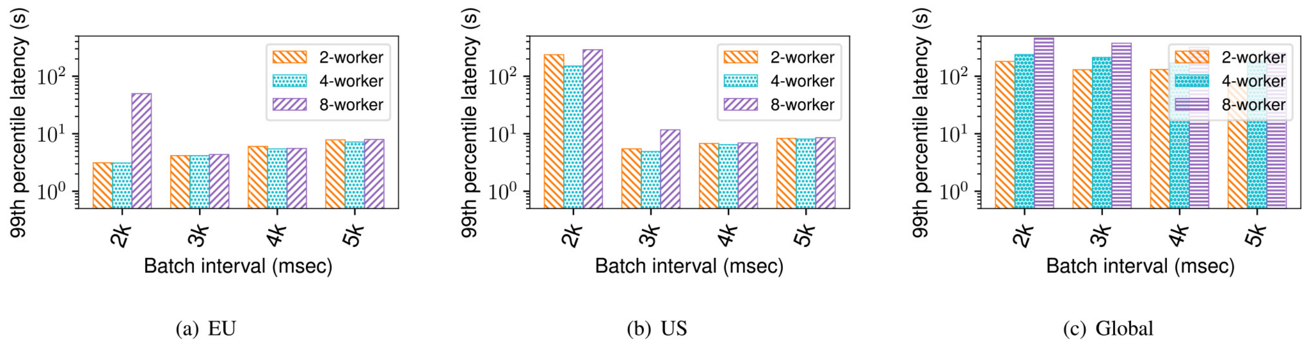


Fig. 15. The 99^{th} percentile latency of Spark by varying the batch interval on the EU, U.S., and Global topologies with input rate of 22k events/second.

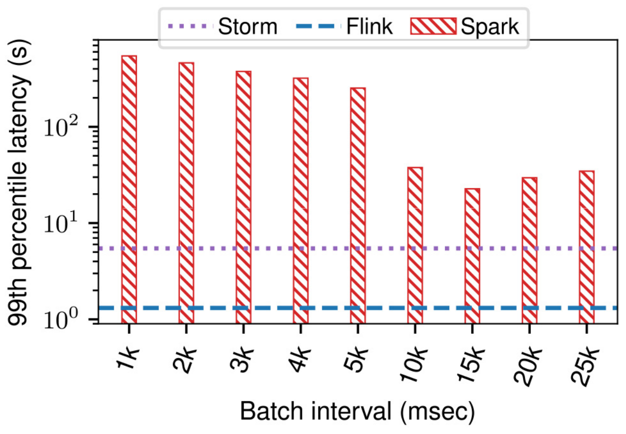


Fig. 16. The 99^{th} percentile latency of Storm and Flink vs. Spark by varying the batch interval of Spark on the Global topology with input rate of 22k events/second using 8 workers.

Storm Tuning Benefits: We repeat the same experiments with Storm to understand the benefits of disabling Storm's record acknowledgment on three topologies with geo-distributed nodes. Fig. 14 shows that disabling the record acknowledgment slightly improves the execution latency of Storm in all topologies. However, it seems that the inter-worker delays have a more profound impact on the Storm's performance and the noAck tuning has only a marginal impact in Fig. 14. In this scenario, the input rate is 22k events/second, and the systems have less load compared to the one with 55k events/second. Therefore, disabling acknowledgment has less impact.

Spark Tuning Benefits: The results of experiments by tuning the batch interval of Spark are presented in Fig. 15. We find that Spark has the best performance on the EU network, i.e., low pairwise RTT, when the batch interval is 3k msecs (see Fig. 15(a)). We also observe that increasing the batch interval of Spark up to 5k msecs results in decreasing the execution latency on the U.S. and global networks where the pairwise RTT is high (see Fig. 15(b) and Fig. 15(c)). Notice that the y-axis is on the logarithmic scale, thus, the improvement is noticeable.

We execute additional experiments to understand the behavior of Spark in the above three realistic topologies. We vary the batch interval from 1k to 25k msecs. We find that by increasing the batch interval from 5k to 25k msecs has a negative effect on the systems that operate on the EU and U.S. topologies. We observe that the performance curve is concave (note that the y-axis is on a logarithmic scale). Indeed, the best performance on the global network, where the pairwise delays are the highest, is achieved when the batch interval is 15k msecs (see Fig. 16). However, even with this optimization, the performance of Spark is not comparable with this of Storm or Flink. When the batch interval exceeds 15k msecs the performance degrades. The reason for that is that the majority of events can be handled when the micro-batch size is large enough. However, the remaining events are processed in the next micro-batches and this behavior may be observable in the next 3 or 4 subsequent batches [40]. In Section VI-E, we provide additional results on our evaluation when the WAN end-to-end delay varies over time.

D. Throughput Measurement

We also study the achieved throughput, i.e., the number of processed events, in the three systems we consider in our

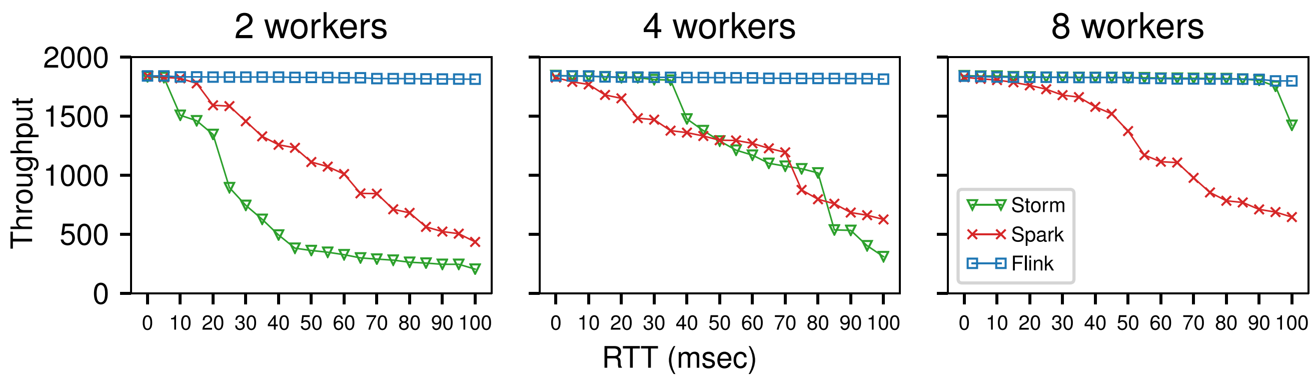


Fig. 17. The moving average throughput for Storm, Spark, and Flink with 2, 4, and 8 workers on a full mesh topology.

study. We report the moving average throughput of the systems for scenarios with the input data rate of 55k events/second and RTT variation, see Fig. 17. The reason to pick such 55k events/second is that all DSPs can process the events without delaying them. Therefore, this input rate is obtained using the capacity of VMs in our testbed when the RTT is 0. The goal is to understand the performance of the DSPs on high loads when the RTTs vary. Across the board, throughput degrades substantially for scenarios with 2 and 4 workers for Storm and Spark as RTT increases. Apache Storm is particularly affected (exponential decrease) when the number of workers is small (2). The performance of Storm and Flink is very similar in scenarios with 8 workers.

We also report the exponential moving average of the throughput for a sample scenario with 4 workers, and the RTT equals 50 msec in Fig. 18. We show this figure to show how the throughput of each system varies throughout an experiment. Across the board, Flink has higher throughput compared to Storm and Spark. Furthermore, Spark has a better throughput compared to Storm.

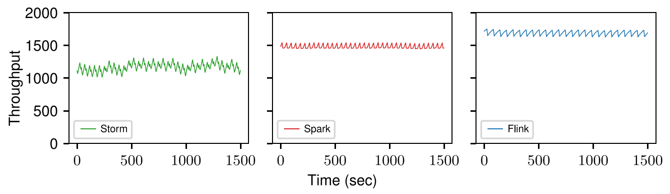


Fig. 18. The moving average throughput for scenarios with 4 workers and RTT=50msec.

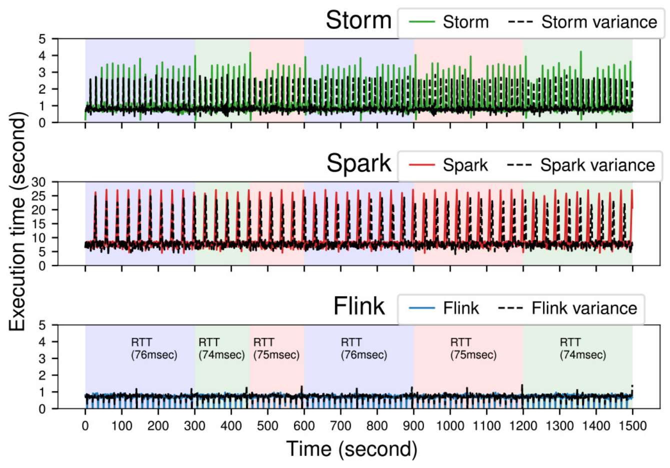


Fig. 19. Storm, Spark, and Flink performance in the RTT variation scenario.

E. Delay Variance Results

Recent studies [37], [38] show that the WAN link delays vary over time. Moreover, network resources in different cloud providers may vary. For example, the network operators should monitor the available token buckets when running experiments on Amazon AWS [68] since the lack of them can add extra delay to the network. To better check the effect of this variance, we conduct a set of experiments by varying the link delay according to the RTT measurements among Amazon VMs taken from [38]. The experiments last 1,500 seconds, and the RTT varies between 74 and 76 msec and we run them for scenarios with 22k events/second. Fig. 19 shows that few msec RTT variations have no impact on the performance of the frameworks confirming that all the frameworks can tolerate a few milliseconds of delay during the query execution time. We used a different color to differentiate the delay variance during the experiment time.

F. Summary of Insights

Flink seems to be the best option across all settings in our experiments and does not require too much tuning. For Spark, the batch interval plays an important role in tuning Spark. The batch interval is the interval in which data is partitioned into blocks before storing them in Spark receiver [64]. We find that Spark should be carefully tuned according to the inter-node delays. For low pairwise delays, e.g., in the EU topology, the best performance is achieved when the batch interval is relatively low, i.e., 3k msecs. However, Spark performs better when the batch interval is higher at 5k msecs for the U.S. network. For a global network, we got the best performance when the batch interval was 15k msecs. Moreover, Spark should be carefully examined for the batch interval depending on the WAN links delay. Overall, higher batch intervals for Apache Spark are suitable for higher WAN links latency. Apache Storm seems to be the more difficult DSP to tune correctly in the WAN environment, and in many settings, even its best performance is not comparable with this of Flink. We conclude that real-time stream processing systems have better performance than micro-batch processing ones. The optimal settings depend on several parameters such as the input rate, the available computation resources, and WAN links properties. We plan to automate the optimal settings in the future.

VII. Conclusion

Today, stream analytics involve geographically distributed processing sites due to the increasing complexity or volume of online products, and privacy protection legislation. In this setting, data streams are processed in-situ and intermediate results have to be exchanged among all sites using a wide-area network. Thus, pairwise site delays are typically way higher than those in a datacenter environment. In this paper, we show that delays of tens of milliseconds have a significant negative impact on data stream processing systems' performance. After evaluating three of the most popular systems for various settings, we conclude that there is not a clear winner in all settings. Nevertheless, Flink seems to be more resistant to network delay because its credit-based flow buffering reduces the impact of WAN delays when transmitting data over TCP connections. Spark can be tuned to be tolerant in many scenarios, including those with high network delay and load. Apache Storm is the most challenging to configure, and its performance is not comparable with these of the other systems in many settings we investigated. The insights gained in this study provide best practices in configuring popular data stream processing systems when operating using wide-area networks. Indeed, we show that it is possible to improve the performance of all these popular big data analytics systems significantly amid even transcontinental delays (where the inter-node delay is more than 30 milliseconds) and achieve performance comparable to this within a datacenter, at least for a low data rate. We believe that the gained insights can also inform decision-making on installing or expanding sites around the globe to better process data streams in the wild.

References

[1] "Apache storm." 2020. [Online]. Available: https://storm.apache.org/

[2] A. Toshniwal *et al.*, "Storm@twitter," in *Proc. ACM SIGMOD*, 2014, pp. 147–156.

[3] "Apache spark." 2020. [Online]. Available: https://spark.apache.org/

[4] M. Zaharia *et al.*, "Spark: Cluster computing with working sets," in *Proc. HotCloud*, vol. 10, 2010, p. 95.

[5] "Apache Flink." 2020. [Online]. Available: https://flink.apache.org/

[6] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bull. IEEE Comput. Soc. Tech. Committee Data Eng.*, vol. 36, no. 4, pp. 28–38, 2015.

[7] "Keystone real-time stream processing platform." 2021. [Online]. Available: https://bit.ly/3ngPoeF (Accessed: Jan. 6, 2021).

[8] "Video access log processing with apache Flink." 2021. [Online]. Available: https://bit.ly/3ol4Vvz (Accessed: Jan. 6, 2021).

[9] "2020 identity fraud study: Genesis of the identity fraud crisis." 2021. [Online]. Available: https://bit.ly/3hSTkBn (Accessed: Jan. 6, 2021).

[10] "The state of online retail performance, spring 2017, Akamai." 2021. [Online]. Available: https://bit.ly/3hSVp09 (Accessed: Jan. 6, 2021).

[11] "Data protection in the EU, the general data protection regulation (GDPR); regulation (EU) 2016/679." 2016. [Online]. Available: http://bit.ly/3qdVUVo

[12] L. Kalman, "New European data privacy and cyber security laws: One year later," *Commun. ACM*, vol. 62, no. 4, p. 38, 2019.

[13] S. Greengard, "Weighing the impact of GDPR," *Commun. ACM*, vol. 61, no. 11, pp. 16–18, 2018.

[14] State of California. "California consumer privacy act—Assembly bill no. 375." 2018. [Online]. Available: http://bit.ly/2K5qOjo

[15] Office of the Privacy Commissioner of Canada. "Amended act on the personal information protection and electronic documents act." 2018. [Online]. Available: https://bit.ly/3oPDSJ7

[16] The Privacy Protection Authority of Israel. "Protection of privacy regulations (data security) 5777-2017." 2018. [Online]. Available: https://bit.ly/2LMwcIA

[17] Personal Information Protection Commission, Japan. "Amended act on the protection of personal information." 2017. [Online]. Available: https://www.ppc.go.jp/en/

[18] Office of the Australian Information Commissioner. "Australian privacy principles guidelines; Australian privacy principle 5—Notification of the collection of personal information." 2018. [Online]. Available: https://bit.ly/38BBrUP

[19] 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. NSDI*, 2014, pp. 275–288.

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

[21] R. Viswanathan, G. Ananthanarayanan, and A. Akella, "CLARINET: WAN-aware optimization for analytics queries," in *Proc. OSDI*, 2016, pp. 435–450.

[22] Q. Pu *et al.*, "Low latency geo-distributed data analytics," in *Proc. ACM SIGCOMM*, 2015, pp. 421–434.

[23] C.-C. Hung, G. Ananthanarayanan, L. Golubchik, M. Yu, and M. Zhang, "Wide-area analytics with multiple resources," in *Proc. EuroSys*, 2018, pp. 1–16.

[24] W. Xiao, W. Bao, X. Zhu, and L. Liu, "Cost-aware big data processing across geo-distributed datacenters," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 11, pp. 3114–3127, Nov. 2017.

[25] A. Vulimiri, C. Curino, B. Godfrey, K. Karanasos, and G. Varghese, "WANalytics: Analytics for a Geo-distributed data-intensive world," in *Proc. CIDR*, 2015, pp. 1–7.

[26] D. Kumar, J. Li, A. Chandra, and R. Sitaraman, "A TTL-based approach for data aggregation in Geo-distributed streaming Analytics," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 3, no. 2, pp. 1–27, 2019.

[27] W. Li, D. Niu, Y. Liu, S. Liu, and B. Li, "Wide-area spark streaming: Automated routing and batch sizing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 6, pp. 1434–1448, Jun. 2019.

[28] A. Jonathan, A. Chandra, and J. Weissman, "Multi-query optimization in wide-area streaming analytics," in *Proc. SoCC*, 2018, pp. 412–425.

[29] F. Lai, J. You, X. Zhu, H. V. Madhyastha, and M. Chowdhury, "Sol: Fast distributed computation over slow networks," in *Proc. NSDI*, 2020, pp. 273–288.

[30] B. Heintz, A. Chandra, and R. K. Sitaraman, "Optimizing timeliness and cost in Geo-distributed streaming Analytics," *IEEE Trans. Cloud Comput.*, vol. 8, no. 1, pp. 232–245, Jan.–Mar. 2020.

[31] A. Jonathan, A. Chandra, and J. Weissman, "Rethinking adaptability in wide-area stream processing systems," in *Proc. HotCloud*, 2018, pp. 1–9.

[32] H. Mostafaei, S. Afridi, and J. Abawajy, "Network-aware worker placement for wide-area streaming analytics," *Future Gener. Comput. Syst.*, vol. 136, pp. 270–281, Nov. 2022.

[33] L. Chen, S. Liu, and B. Li, "Optimizing network transfers for data analytic jobs across Geo-distributed Datacenters," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 2, pp. 403–414, Feb. 2022.

[34] Y. Chen, L. Luo, D. Guo, O. Rottenstreich, and J. Wu, "SDTP: Accelerating wide-area data analytics with simultaneous data transfer and processing," *IEEE Trans. Cloud Comput.*, early access, Oct. 15, 2021, doi: 10.1109/TCC.2021.3119991.

[35] K. Beedkar, D. Brekardin, J.-A. Quiané-Ruiz, and V. Markl, "Compliant geo-distributed data processing in action," *Proc. VLDB Endow.*, vol. 14, no. 12, pp. 2843–2846, Jul. 2021.

[36] A. Jonathan, A. Chandra, and J. Weissman, "WASP: Wide-area adaptive stream processing," in *Proc. ACM/IFIP/USENIX Middleware*, 2020, pp. 221–235.

[37] Y. Jin *et al.*, "Zooming in on wide-area latencies to a global cloud provider," in *Proc. SIGCOMM*, 2019, pp. 104–116.

[38] W. Reda *et al.*, "Path persistence in the cloud: A study of the effects of inter-region traffic engineering in a large cloud provider's network," *SIGCOMM Comput. Commun. Rev.*, vol. 50, no. 2, pp. 11–23, 2020.

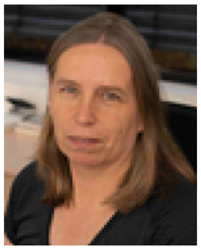
[39] C.-Y. Hong *et al.*, "Achieving high utilization with software-driven WAN," in *Proc. ACM SIGCOMM*, 2013, pp. 15–26.

[40] S. Chintapalli *et al.*, "Benchmarking streaming computation engines: Storm, Flink and spark streaming," in *Proc. IPDPSW*, 2016, pp. 1789–1792.
[41] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl, "Benchmarking distributed stream data processing systems," in *Proc. ICDE*, 2018, pp. 1507–1518.
[42] M. A. Lopez, A. G. P. Lobato, and O. C. M. B. Duarte, "A performance comparison of open-source stream processing platforms," in *Proc. GLOBECOM*, 2016, pp. 1–6.
[43] "Apache Zookeeper." 2020. [Online]. Available: https://zookeeper.apache.org/
[44] "Netty." 2020. [Online]. Available: https://netty.io/
[45] "Apache Hadoop." 2020. [Online]. Available: https://hadoop.apache.org/
[46] "What is/are the main difference(s) between Flink and storm?" 2015. [Online]. Available: https://bit.ly/3qiaQSF
[47] S. Saxena and S. Gupta, *Practical Real-Time Data Processing and Analytics: Distributed Computing and Event Processing Using Apache Spark, Flink, Storm, and Kafka*. London, U.K.: Packt, 2017.
[48] "Trident API overview." 2021. [Online]. Available: https://bit.ly/3win8NC
[49] S. Zeuch *et al.*, "Analyzing efficient stream processing on modern hardware," *Proc. VLDB Endow.*, vol. 12, no. 5, pp. 516–530, 2019.
[50] "Extending the Yahoo streaming benchmarks." 2020. [Online]. Available: https://github.com/dataArtisans/yahoo-streaming-benchmark
[51] "Apache kafka." 2020. [Online]. Available: https://kafka.apache.org/
[52] "Redis." 2020. [Online]. Available: https://redis.io/
[53] J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM*, vol. 56, no. 2, pp. 74–80, 2013.
[54] "Traffic control (TC)." 2020. [Online]. Available: https://wiki.debian.org/TrafficControl
[55] "The state of the Internet." 2020. [Online]. Available: https://bit.ly/39gKAS4
[56] "Apache storm: Performance tuning." 2021. [Online]. Available: https://bit.ly/3bLzYfM
[57] "Equinix." 2020. [Online]. Available: https://www.equinix.com/data-centers/
[58] "Amazon global infrastructure." 2020. [Online]. Available: http://amzn.to/38zsFq4
[59] "Data residency in azure." 2020. [Online]. Available: http://bit.ly/3qdWimV
[60] "Google Datacenters." 2020. [Online]. Available: https://about.google/locations/
[61] "AT&T network delay." 2020. [Online]. Available: https://ipnetwork.bgtmo.ip.att.net/pws/network_delay.html (Accessed: Apr. 11, 2020).
[62] "Global ping statistics: Ping times between Wonder Network servers." 2020. [Online]. Available: https://wondernetwork.com/pings(Accessed: Sep. 10, 2020).
[63] "Analysis of network flow control and back pressure: Flink advanced tutorials." 2020. [Online]. Available: https://bit.ly/340lvZ7
[64] "Spark streaming programming guide." 2020. [Online]. Available: http://bit.ly/3bw3bfb (Accessed: Dec. 28, 2020).
[65] "Socket statistics (SS)." 2020. [Online]. Available: http://bit.ly/39nj2dm
[66] "Can apache spark use TCP listener as input?" 2019. [Online]. Available: https://bit.ly/3bJIUSV
[67] "vmStat—Report virtual memory statistics." 2020. [Online]. Available: http://bit.ly/38BlTAd
[68] A. Uta *et al.*, "Is big data performance reproducible in modern cloud networks?" in *Proc. NSDI*, 2020, pp. 513–527.

Habib Mostafaei (Member, IEEE) received the Ph.D. degree in computer science and engineering from Roma Tre University in 2019. He was a Postdoctoral Researcher with Technische Universität Berlin where he was involved in the BIFOLD-BBDC Project from 2019 to 2022. He worked as a Full-Time Faculty Member with Computer Engineering Department, Azad University from 2009 to 2015. He is currently an Assistant Professor of Computer Science with the Eindhoven University of Technology. His current main research fields include networked systems, network management, and distributed systems. He is a member of ACM. For additional information: https://mostafaei.bitbucket.io

Georgios Smaragdakis (Senior Member, IEEE) received the Ph.D. degree in computer science from Boston University in 2009 and the Diploma degree in electronic and computer engineering from the Technical University of Crete. He is a Full Professor and the Chair of Cybersecurity with the Faculty of Electrical Engineering, Mathematics, and Computer Science, Delft University of Technology. He has been a Researcher with the Max Planck Institute for Informatics since 2019 and a Principal Investigator and a Fellow with the Berlin Institute for the Foundations of Learning and Data since 2017. He was a Professor with TU Berlin from 2017 to 2021 and a Research Collaborator with Akamai Technologies from 2014 to 2021. From 2014 to 2017, he was a Marie Curie Fellow with the Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology (MIT) and from 2015 to 2018 he was a Research Affiliate with the MIT Internet Policy Research Initiative. From 2008 to 2014, he acted as a Senior Researcher with Deutsche Telekom Laboratories. In 2008, he was a Research Intern with Telefonica Research. His research brings a data- and measurement-driven approach to the study of the Internet's security, resilience, state, and performance, as well as to the enhancement of Web privacy and security. His research was recognized with a European Research Council Starting Grant Award (2015), a Marie Curie International Outgoing Fellowship (2013), the Best Paper Awards with ACM SIGCOMM (2021), ACM IMC (2018, 2016, and 2011), ACM CoNEXT (2019 and 2015), IEEE INFOCOM (2017), three IETF/IRTF Applied Networking Research Prizes (2022, 2020, and 2019), and the "Best of ACM SIGCOMM Computer Communication Review" (2019) and selected for Communications of the ACM (CACM) Research Highlights in 2021. He has served as the Technical Program Chair of TMA 2020, ACM CoNEXT 2019, and PAM 2018. He has also been involved in the organization and technical program committees of many conferences, including, ACM SIGCOMM, ACM IMC, ACM SIGMETRICS, ACM CoNEXT, ACM HotNets, IEEE EuroS&P, IEEE Infocom, IEEE HotWeb, USENIX ATC, PETS, and ESORICS. He is a Senior Member of ACM.

Thomas Zinner (Member, IEEE) received the Ph.D. degree in computer science from the University of Würzburg in 2012. He has been an Associate Professor with the Department of Information Security and Communication Technology, NTNU, Norway, since 2019 and currently leads the Networking Research Group. He was a Visiting Professor and the Head of the Research Group INET with TU Berlin from 2018 to 2019. From 2013 to 2018, he was the Head of the Research Group on Next Generation Networks with the Chair of Communication Networks, University of Würzburg. His research interests cover cognitive network management and network softwarization with particular focus on performance and security aspects. He was a recipient of several best paper awards, the DASH-IF "Excellence in DASH Award" (2020) and the ITC Rising Scholar Award (2019). He has served as the Technical Program Chair for ITC 2018. He has been involved in the organization and technical program committees of many conferences and workshops, including ITC, Netsoft, IM/NOMS, CNMS, and ACM CoNEXT. He is a member of ACM.



Anja Feldmann received the master's degree from Universität Paderborn, Germany and the Ph.D. degree from Carnegie Mellon University. In the next four years, she did research work with AT&T Labs Research, before taking Professor positions with Saarland University, the TU Munich, and the TU Berlin. Since 2018, she has been the Director with the Max Planck Institute for Informatics, Saarbrücken, Germany. Her current research interests include Internet measurement, traffic engineering and traffic characterization, network performance debugging, and network architecture. She was the Co-Chair of ACM SIGCOMM 2003 and ACM IMC 2011 and the Co-PC-Chair of ACM CoNext 2020, ACM SIGCOMM 2007, ACM IMC 2009, and ACM HotNets 2014.