Reconfigurable Streaming for the Mobile Edge

Abhishek Tiwari
Department of Computer Science
University of Toronto
atiwari@cs.toronto.edu

Brian Ramprasad
Department of Computer Science
University of Toronto
brianr@cs.utoronto.ca

Seyed Hossein Mortazavi
Department of Computer Science
University of Toronto
mortazavi@cs.toronto.edu

Moshe Gabel
Department of Computer Science
University of Toronto
mgabel@cs.toronto.edu

Eyal de Lara
Department of Computer Science
University of Toronto
delara@cs.toronto.edu

ABSTRACT

Deploying stream computing applications on edge networks brings a new set of challenges including frequent reconfigurations due to client mobility and topology changes, geographical constraints from application semantics, state management over wide-area networks, and more. Current stream processing frameworks do not adequately address these challenges since they are designed for use inside data centers and rely on global coordination between participating nodes. Merlin is a new stream processing framework designed from the ground up for stream processing on the edge. Merlin supports fast reconfiguration without disrupting applications by decoupling data delivery from data processing and removing the need for global coordination.

CCS CONCEPTS

• **Information systems → Stream management**.

KEYWORDS

Edge Computing; Streaming; Dynamic Reconfiguration

ACM Reference Format:
Abhishek Tiwari, Brian Ramprasad, Seyed Hossein Mortazavi, Moshe Gabel, and Eyal de Lara. 2019. Reconfigurable Streaming for the Mobile Edge. In *The 20th International Workshop on Mobile Computing Systems and Applications (HotMobile '19), February 27–28, 2019, Santa Cruz, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3301293.3302355

1 INTRODUCTION

Our ability to support the acquisition, processing, and storage of huge volumes of data has transformed business and culture, technology and science [7]. Yet, if the technical challenge of last decade was the exponential increase in the volume of data, then the challenge of the next decade is the proliferation of geographically distributed data. The advent of the Internet-of-Things (IoT), mobile devices, and sensor networks is expected to lead to an exponential increase in the number of distributed data sources, with a predicted 27 billion connected devices by 2021 [4]. The prevailing cloud computing paradigm is not a good match for this flood, and scaling computational resources in the cloud will not help if we cannot get the data into the data center in the first place, due to bandwidth and latency constraints.

Edge or fog computing is an emerging alternative [13, 14] that uses nearby resources, as well as nodes on the path to the cloud, to provide computation and storage over very large number of geographically distributed sites. Such multi-tier hierarchies are beneficial when latency requirements are stringent enough to render cloud processing unviable, where client devices are too weak, and when the computation requires both locally and globally relevant information. Processing data near to the point of creation has several benefits: it reduces wide-area network bandwidth demands by aggregating data locally, it lowers the latency to compute results that depend on local data, and it helps limit the geographic area where sensitive data propagates.

Unfortunately, existing distributed stream processing frameworks [1, 3, 6, 8, 17] are a poor match for edge computing. They were designed to work inside data centers on nodes that connect over high capacity, low-latency links. Moreover, these designs assume a mostly static environment where data is produced by a comparatively small and stable set of data sources, and a network topology that changes infrequently. In contrast, edge deployments are characterized by geographically distributed nodes connected over links that (relative to the data center) have low bandwidth and high latency. Moreover, in edge deployments, data is produced at a large number of geographically distributed sources (e.g., smart phones, intelligent transportation), and client mobility results in frequent changes in network topology as clients move between edge nodes.

We propose a new stream processing framework for running applications on edge data centers. We assume a hierarchy of edge data centers that has a traditional wide-area cloud data center at its root, and additional layers of data centers that become progressively smaller as we approach the edge of the network. We assume that the data center hierarchy is shared infrastructure that will run a large number of data streaming applications. Moreover, since most data centers in the hierarchy have limited capacity, application components have to be dynamically removed or deployed in response to changes in usage patterns.

Consider for example a smart city traffic monitoring system deployed over a wide area. The system collects streams of data from

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
HotMobile '19, February 27–28, 2019, Santa Cruz, CA, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6273-3/19/02…$15.00
https://doi.org/10.1145/3301293.3302355

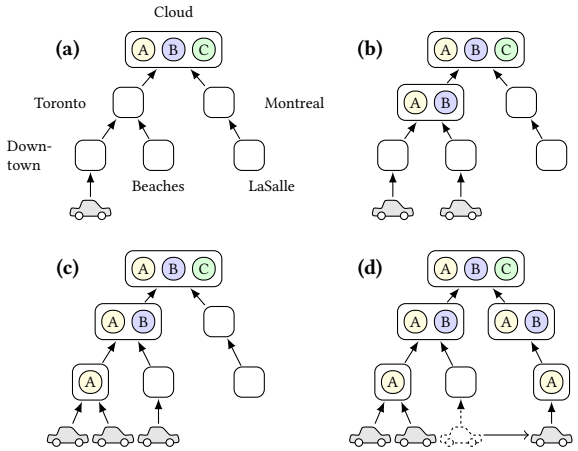

Figure 1: Reconfiguration in a traffic monitoring application: (a) initially all components run in the cloud; (b) additional instances of A and B started in Toronto to handle increase in load; (c) extra instance of A added to Downtown data center to handle load spike; (d) reconfigure deployment as traffic shifts to a new city.

road sensors, GPS-equipped vehicles, environmental monitors, and other sources and aggregates them at multiple levels to manage traffic. This huge volume of data could overwhelm network capacity at the cloud data center, yet correct operation requires aggregating data from multiple streets, neighborhoods, and cities. An edge architecture provides an elegant solution, as data can be aggregated locally and put to use exactly where it is needed, without flooding network links at the upper layers.

Figure 1 illustrates how the deployment of the monitoring system's components evolves over time on a hierarchical data center network with three layers. Initially, all the application's components (A, B, C) are only running at the cloud data center. At this stage there is only a modest load and it is most convenient to send all data for processing to the cloud. In the second stage, additional instances of A and B are started on the Toronto data center to handle an increase in data originating from this city. In stage three, an instance of A is started on the Downtown data center to address a sudden increase in traffic concentration. Finally, as traffic shifts from Toronto to Montreal, new bolts are instantiated along this route.

Dynamic deployment of streaming applications on a hierarchical data center network, however, presents several important challenges: the framework needs to be able to support the addition and removal of computational elements with minimal disruption; the state of the application needs to be rebalanced; developers must be able to specify geographical constraints on placement; and the system needs to be able to handle failure and intermittent connectivity.

In this paper we take an initial step toward addressing the challenge of reconfiguration. We first introduce our vision for stream processing on a hierarchical network of data centers, and discuss key challenges.We then discuss the limitations of existing stream processing architectures and illustrate the detrimental effects of reconfiguration with experiments using the popular Apache Storm [6] framework. We then introduce Merlin, our dynamic hierarchical streaming prototype and show that it enables frequent dynamic application reconfiguration without any stoppage time. Merlin achieves this by decoupling data delivery from data processing and by removing the need for global coordination.

2 VISION AND CHALLENGES

We next describe our vision for enabling dynamic stream processing on a hierarchical network of edge data centers.

We assume the common operator graph model: stream processing applications are represented as directed acyclic graphs (DAG) consisting of *nodes* that originate or process streams of data. A stream is made of individual data units called *tuples*, and nodes in the graph are either *spouts* that emit tuples, or *bolts* that consume and process tuples, and (optionally) emit new tuples[1].

A developer parallelizes their application by decomposing their algorithm into a collection of connected bolts and spouts, called a *logical plan*, which can process tuples concurrently. The framework then compiles a *physical plan* that determines the number of instances of each bolt and spout to create and their placement on the cluster. The logical plan for the sample traffic monitoring application consists of: a spout that generates tuples with traffic, GPS, and environmental data; bolt-A, which aggregates traffic by neighborhood and emits new tuples with traffic predictions and per-neighborhood summaries; bolt-B, which aggregates data by city, provides traffic management, and emits a new tuple every few seconds with the per-city summaries; and finally, bolt-C, which aggregates the data for the entire country. When deployed on a cloud data center, road sensors and vehicles upload information to a central database on the cloud, from which they can be retrieved by the spout.

Our goal is to enable the deployment of stream processing applications on a hierarchical network of data centers. We assume that the data center hierarchy is organized as a tree that has a traditional wide-area cloud data center at its root, and an arbitrary number of additional layers of data centers that get progressively smaller as we approach the edge of the network. We also assume that the data center hierarchy is shared infrastructure that will run a large number of data streaming applications, and that data centers that are closer to the edge have limited capacity. For example, Figure 1(a) illustrates a hierarchical network consisting of three levels: a traditional wide-area cloud data center, two city-scale data centers, and several neighborhood-scale data centers.

We further assume that an application's logical plan consists of nodes that can all run at the cloud data center at the root of the hierarchy. In addition, we assume that some (and potentially all) of the nodes can also be instantiated on data centers closer to the edge of the network, subject to developer-provided constraints that indicate how close to the edge it is safe to place a node without altering application semantics. In our traffic monitoring example, bolt-A (neighborhood aggregation) can run correctly at any level of the data center hierarchy. In contrast, bolt-B (city aggregation and management) can only run at the root on a city-level data

[1]For simplicity, we adopt the nomenclature of Apache Storm [6].

center. Running this bolt at a neighborhood-level data center will result in an inaccurate statistics for the city as traffic reports from other neighborhoods will be missed. In this scenario, sensors and cars operate as spouts emitting reports (i.e., tuples) to the closest neighborhood-level data center. Tuples are then either processed by a locally-installed bolt, or get propagated up towards the root of the hierarchy.

By default, an application is initially deployed on the cloud data center. As the data stream starts to flow through the data center hierarchy, additional instances of nodes can be dynamically added progressively closer to the edge of the network, subject to developer constraints and the load on the shared network. Figure 1 shows how the deployment of our sample application evolves over time. Initially, all three application bolts are only installed at the cloud data center. At this stage the application is only experiencing modest load and it is most convenient to send all reports to the cloud. In the second stage, additional instances of bolt-A and bolt-B are started on the Toronto data center to handle an increase in reports emanating from this city. In stage three, an instance of bolt-A is started on the Downtown data center to address a sudden increase in traffic concentration in this neighborhood. In our traffic monitoring application, placing bolts close to the edge of the network dramatically reduces the number of tuples that must be sent to the the wide-area cloud data center, as the majority of the tuples get processed locally, and only a much smaller number of updates will propagate to the city-wide and wide-area cloud data centers. This approach also reduces the latency for calculating neighborhood and city-level statistics, as there is no need to wait for the data to propagate all the way to the cloud and back. Finally, the approach is better for privacy as individual reports are no longer available on the cloud.

Stream processing on a multi-tier hierarchy of edge data centers, however, introduces a new set of unique challenges:

- Frequent reconfiguration. Since data centers (particularly those close to edge of the network) have limited resources, it is unreasonable to assume that all applications will be pre-installed and run all the time at all locations. Instead, the physical map of an application will have to change dynamically as users move between edges, or as the need to run different applications on a given edge arises. This requires mechanisms for reconfiguration without disruption as well as policies that optimize system performance.
- Programming interface. The developer needs to have a way to instruct the streaming framework where to place elements. Specifying the parallelism factor for a bolt is not sufficient as the number and location of data sources (e.g., road sensors and vehicles) changes over time. In addition, the structure of the data centre hierarchy may not be known to the developer, and the hierarchy may not be uniform. (i.e., different parts of the hierarchy may consist of different number of levels or may vary in their geographic coverage).
- Support for failures and disconnection. Edge deployments may involve large networks with a large number of links that may experience intermittent failures. In addition, some scenarios may require support for disconnected operation.

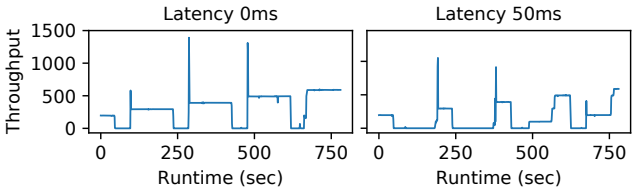

Figure 2: Storm's throughput drops to zero at the start of the reconfiguration operation, and can take many seconds to recover. Latency exacerbates this affect.

- Optimal resource allocation on a hierarchical data centre network is challenging as there is a need to balance the multiple application requirements in the face of heterogeneity in data center resource and application demands. For example, the decision to deploy a new instance of bolts A and B in the neighborhood data center must be taken in the face of performance, load management, and geographical constraints.

In the rest of this paper, we take an initial step to address the first of these challenges – the need for frequent reconfiguration. Addressing the other challenges is left for future work. Before we introduce our new approach, however, we present results from a simple experiment with Apache Storm that shows that existing cloud-based streaming systems are poorly suited to the task of handling frequent reconfiguration requests. While Apache Storm is just one of many existing cloud-based streaming systems, we argue that it is representative of a broad class cloud-based streaming frameworks that would be expected to perform in a similar way.

3 APACHE STORM

We next explore the suitability of using an existing cloud stream processing system on a hierarchical edge network. We focus our evaluation on Apache Storm [6] because it is a widely-used framework; however, we argue that our evaluation is representative of a wide class of cloud-based stream processing systems, such as Twitter Heron [8], Apache Flink [3], Spark [17], and Drizzle [15]. In all these systems application components are tightly coupled: the element that emits a tuple also directly delivers it to the element that is meant to consume it. All these frameworks also rely on global coordination, and require the topology to stop and restart in order to add or remove nodes.

In our experiment, we deploy the spout and first two bolts of an example aggregation application on an emulated two-tier hierarchical edge network composed of one wide-area cloud data center and nine edge data centers connected over network links with 50 msec of latency.

We configured the cloud data center to run the Storm Master (i.e., Nimbus), Zookeeper, and a single Storm worker, which ran bolt-B for the duration of the experiment. We configure each of the edge data centers to run two Storm workers: a spout and a bolt. At the start of the experiment, we run a single instance of bolt-A and a single spout on one of the edge data centers. After three minutes, we use the Storm *rebalance* command to start an extra instance of bolt-A and a new spout on one of the idle edge data centers. We

repeated this process every 3 minutes until we ran out of idle edge data centers.

Figure 2 depicts the throughput measured at the cloud data center for the first 10 minutes of the experiment. The figure shows that the throughput drops to zero at the start of every reconfiguration operation, and can take many seconds to recover. To be fair, reconfiguration is very rare in the cloud environments for which Storm was designed. It is therefore not surprising that Storm experiences multi-second disruption times when adding new components to the topology. While this may not be a significant issue for traditional cloud deployment where reconfiguration is indeed rare, it will significantly disrupt the operation of an edge network where reconfiguration is expected to occur frequently.

Reconfiguration in Storm is a high latency operation that requires the complete network topology to synchronize before making any changes. Upon receiving the rebalance command, Storm halts all spouts and bolts and allows them to flush their data. Storm coordinates the entire process by communicating via Zookeeper and this communication is based on a polling mechanism where the Storm slave workers poll the Zookeeper server. Essentially the 'stop the world event' causes the topology to be down for a significant amount of time, making dynamic topology changes an extremely expensive operation. The early-binding or tightly coupled design of the framework where the component emitting data is directly connected with the component processing the data is good for performance; however, it makes reconfiguration hard. We conclude that current stream processing engines are a poor match for the edge computing setting due to their reliance on global coordination between participating nodes.

4 MERLIN

Merlin is a new framework for stream processing on a hierarchical network of data centers. Merlin supports dynamic application deployment and enables the frequent reconfiguration of an application's physical plan without requiring global coordination. Merlin's design in based on two key principles: decoupling tuple delivery from processing, and the use of only local knowledge for routing tuples. Merlin assumes that each data center in the hierarchy runs four components: a local data store, one or more executors, a pusher, and a router module. The data store provides reliable persistent storage for tuples and can be locally replicated for performance and durability. The executors run instances of locally installed application bolts. In Merlin, bolts do not communicate directly with each other. Instead, the Merlin *router* reads tuples from the data store, and decides for each tuple whether to pass it to one of the *executors* for processing by a local bolt, or leave it for the *pusher*, which will propagate it to the next level of the hierarchy by inserting the tuple into the data store of its parent. New tuples emitted by a local bolt are written to the local data store and are subsequently handled by the router.

For example, in the scenario shown in Figure 1 (c), tuples created by road sensors in the Beaches neighborhood are initially stored in the local store of the Beaches data center. The router running on this data center reads the tuples and copies them to the Toronto data center as there are no locally installed bolts that could process them. In turn, the router running on the Toronto data center reads

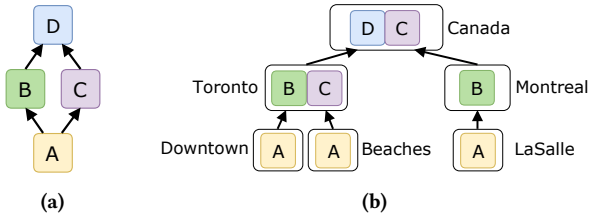

Figure 3: Application logical plan (a) and different mappings on to the edge network (b).

tuples from its local data store and forwards them to the locally installed instance of bolt-A. Tuples emitted by bolt-A are initially stored on the local data store of the Toronto data center where the router will forward them to the locally installed instance of bolt-B. Finally, tuples emitted by bolt-B are copied to the cloud data center where they will be eventually passed to bolt-C.

The above design makes it possible for Merlin to handle the addition and removal of bolts from a data center as a local operation. Adding or removing a bolt only requires updating information on the local router, which can then start forwarding compatible tuples for local processing. Similarly, changes made to a data center do not affect the routing decisions made by the rest of the data centers in the network. Individual data centers do not know the global placement of bolts, and their local router module only needs to make a simple decision: consume the tuple locally or push it to the parent. To ensure that it is always safe for the Merlin router to propagate tuples towards the root of the data center hierarchy, Merlin assumes that all application bolts are installed by default on the root data center (Section 2). Merlin's store-and-forward approach is also a good match for applications that experience intermittent connectivity, as tuples meant to be propagated up the hierarchy can be kept in local storage until the connection to the parent is re-established.

While Merlin assumes a simple hierarchy in the shape of a tree to forward tuples, the framework supports more complex streaming application topologies and allows bolts to emit tuples for multiple destinations. For example, Figure 3(a) shows a logical plan consisting of 4 bolts. Bolt A emits two types of tuples, which get consumed by either bolt B or bolt C. Figure 3(b) shows a possible deployment of this application on Merlin. The example assumes that the application developer has indicated that application correctness would not be impacted by running bolt A and bolt B, on neighborhood and city-level data centres, respectively.

4.1 Prototype

We implemented a prototype of Merlin that provides *at-least once* and *out-of-order* processing of data streams on a hierarchy of data centers. Figure 4 illustrates the various components that Merlin deploys on each data center. The prototype uses Cassandra DB [9] as its local data store. As the figure shows, each data center deploys an independent Cassandra ring. Every Merlin data center also runs a *router* that reads tuples from its local Cassandra ring and either delivers them to an *executor* for consumption by an application bolt or to the *pusher*, which will transfer the tuple to the Cassandra ring of the parent data center. The design provides at-least once

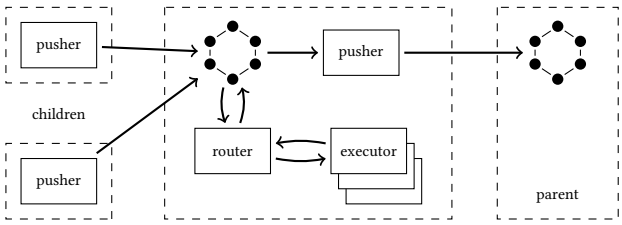

Figure 4: Merlin architecture.

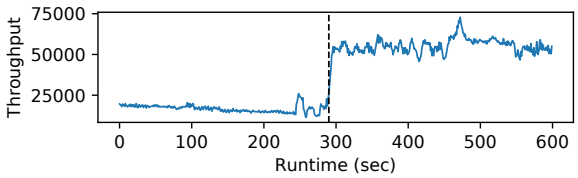

Figure 5: Merlin throughput over 10 minutes. A new edge data center was added after 5 minutes.

processing guarantee: in event of a failure, the data is available in the Merlin node's local data store; on recovery, the router will pick up undelivered tuples and route them again. Merlin implements a simple flow control algorithm that throttles pushers when available free space on the parent drops below a configurable threshold. Bolts in Merlin are Java objects that adhere to an API for consuming and emitting tuples.

The prototype does not support the replication or migration of application-specific bolt state, which is required to handle the addition or removal of bolts that depend on application state to operate. For example, adding an instance of bolt-A (neighborhood-aggregation) to the Downtown data center (see Figure 1(c)) requires replicating information about all Downtown streets that was up to this point kept by the instance of bolt-A running on the Toronto data center. We plan to explore the use of CloudPath [11] to run bolts as *stateless* functions that rely on a hierarchical distributed database (such as PathStore [10]) for application state management.

4.2 Preliminary Evaluation

Figure 5 presents experimental results for a topology that starts with one Merlin edge data center and one Merlin data center. At the 300 second mark, we add another Merlin edge data center to the topology. As the figure depicts, the new Merlin data center assimilates into the topology and starts functioning with no disruption or delay. The reconfiguration would be similarly immediate if an edge data center were to leave the topology. By design, Merlin is a loosly coupled system; Merlin nodes maintain no knowledge of the global topology and only know their parent node. The tuples are not emitted directly to the destination bolt for consumption but are rather inserted into the local store of the parent node, which routes them upstream. All a node has to do to join the topology is to get the parent node information and start writing to it via a Cassandra session. This loosely coupled design is well suited for a dynamic edge architecture and IoT streaming applications as it does not require stopping and synchronizing the entire topology in order to reconfigure it.

5 RELATED WORK

Existing work on edge stream processing focuses on resource allocation, task placement, and effective parallelization. For example, work on video streaming applications deals with optimizing for lag-sensitive applications or pushing computation towards the edge [16, 18, 19]. These works assume a static network topology, and also use global optimization.

Most existing stream processing engines require *global coordination*. They are designed for large cloud data centers [1–3, 6, 8, 17], and assume that data is first propagated to the cloud and that all nodes in the cluster communicate over low latency and high bandwidth links. For example, adding new workers to a running Apache Storm [6] topology requires checkpointing and stopping the entire topology, reconfiguring it, and starting it again. Similarly, Apache Flink [3] requires global coordination of the dataflow [5] for failure recovery and for reconfiguration.

Kafka Streams similarly relies on global coordination (for example to provide end-to-end message processing guarantees) and low-latency communication between all nodes and the Kafka brokers. Drizzle [15] introduces group scheduling which reduces reconfiguration to within several seconds for small groups, but still requires 10 to 100 seconds when adding new nodes, and requires centralized scheduling which is infeasible across data centers. SpanEdge [12] allows placing of processing tasks at the edge for applications that require low latency and to reduce bandwidth usage on the WAN. It is built on top of Storm which suffers from the same restart delay when reconfiguration of the topology is required. To maintain their low latency goals, only a static topology can be used. In contrast, Merlin is designed from the ground up to enable fast and frequent reconfiguration by avoiding global coordination and by decoupling tuple delivery from processing.

6 CONCLUSION AND FUTURE WORK

This paper lays the groundwork for a new kind of streaming platform designed for edge computing and mobile settings. Unlike existing stream computing engines such as Storm and Flink, Merlin's loosely-coupled design allows quick and non-disruptive reconfiguration – regardless of network size. There are many research and engineering challenges that we plan to tackle including state management for stateful operators, exactly-once processing semantics, in order execution semantics, an Application Programming Interface (API) for user code, a higher level query processing engine, a mechanism to specify geographic constraints for deployment, load balancing and more sophisticated flow control, dynamic scaling of routers on the nodes, and generation of a physical plan from the logical plan.

REFERENCES

[1] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. 2013. MillWheel: Fault-tolerant Stream Processing at Internet Scale. *Proc. VLDB Endow.* 6, 11 (2013), 1033–1044.

[2] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. 2015. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-scale, Unbounded, Out-of-order Data Processing. *Proc. VLDB Endow.* 8, 12 (2015), 1792–1803.

[3] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* 38 (2015), 28–38.

[4] Cisco VNI. 2017. Cisco Visual Networking Index: Forecast and Methodology, 2016–2021. (2017).

[5] Apache Software Foundation. [n. d.]. Apache Flink: Data Streaming Fault Tolerance. https://ci.apache.org/projects/flink/flink-docs-master/internals/stream_checkpointing.html. ([n. d.]).

[6] Apache Software Foundation. 2015. Apache Storm. http://storm.apache.org/index.html. (2015).

[7] Rob Kitchin. 2014. *The Data Revolution: Big Data, Open Data, Data Infrastructures and Their Consequences.* SAGE Publications Ltd.

[8] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. 2015. Twitter Heron: Stream Processing at Scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. 239–250.

[9] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.* 44, 2 (2010), 35–40.

[10] Seyed Hossein Mortazavi, Bharath Balasubramanian, Eyal de Lara, and Shankaranarayanan Puzhavakath Narayanan. 2018. Toward Session Consistency for the Edge. In *USENIX Workshop on Hot Topics in Edge Computing (HotEdge 18)*.

[11] Seyed Hossein Mortazavi, Mohammad Salehe, Carolina Simoes Gomes, Caleb Phillips, and Eyal de Lara. 2017. CloudPath: A Multi-Tier Cloud Computing Framework. In *2nd ACM/IEEE Symposium on Edge Computing (SEC)*.

[12] H. P. Sajjad, K. Danniswara, A. Al-Shishtawy, and V. Vlassov. 2016. SpanEdge: Towards Unifying Stream Processing over Central and Near-the-Edge Data Centers. In *2016 IEEE/ACM Symposium on Edge Computing (SEC)*. 168–178.

[13] M. Satyanarayanan. 2017. The Emergence of Edge Computing. *Computer* 50, 1 (2017), 30–39. https://doi.org/10.1109/MC.2017.9

[14] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. 2016. Edge Computing: Vision and Challenges. *IEEE Internet of Things Journal* 3, 5 (2016), 637–646.

[15] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J. Franklin, Benjamin Recht, and Ion Stoica. 2017. Drizzle: Fast and Adaptable Stream Processing at Scale. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. 374–389.

[16] Shanhe Yi, Zijiang Hao, Qingyang Zhang, Quan Zhang, Weisong Shi, and Qun Li. 2017. LAVEA: Latency-aware Video Analytics on Edge Computing Platform. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing (SEC '17)*. Article 15, 13 pages.

[17] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM* 59, 11 (2016), 56–65.

[18] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodik, Matthai Philipose, Paramvir Bahl, and Michael J. Freedman. 2017. Live Video Analytics at Scale with Approximation and Delay-Tolerance. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 377–392.

[19] Tan Zhang, Aakanksha Chowdhery, Paramvir (Victor) Bahl, Kyle Jamieson, and Suman Banerjee. 2015. The Design and Implementation of a Wireless Video Surveillance System. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking (MobiCom '15)*. 426–438.