Data-driven Stream Processing at the Edge

Eduard Renart, Javier Diaz-Montes, Manish Parashar
Rutgers Discovery Informatics Institute, Rutgers University, Piscataway, NJ, 08854, USA
contact author: egr33@scarletmail.rutgers.edu

***Abstract*—The popularity and proliferation of the Internet of Things (IoT) paradigm is resulting in a growing number of devices connected to the Internet. These devices are generating and consuming unprecedented amounts of data at the edges of the infrastructure, and are enabling new classes of data-driven applications, however, current approaches typically rely on cloud platforms located at the core of the infrastructure to process data. As the number of devices and the amount of data they generate and consume increases, such core-centric approaches are becoming increasingly inefficient as they need to transfer data back and forth between the edge and the core. Furthermore, the latencies associated with such data transfer may not be able to support applications involving time-critical data-driven decision making. In this paper, we propose an edge-based programming framework that allows users to define how data streams are processed based on the content and the location of the data. This enables the definition of data-driven reactive behaviors that can effectively exploit data patterns to dynamically drive stream processing, leveraging resources located at the edges of the infrastructure. We have implemented a prototype of the proposed approach and performed several experiments to evaluate its scalability and efficiency against a more typical single-cloud approach. Using a smart-city application usecase, we illustrate that the presented programming system can support data-driven stream processing using edge resources. In terms of scalability, our experiments show that the system can scale to hundreds of nodes while keeping overheads low. Our experiments also show that our approach can perform up to 56% better than a single cloud approach that does not consider data and user locality.**

Keywords*—*Content-based processing, Stream Processing, Edge Computing, Multi-Cloud

I. Introduction

The Internet of Things (IoT) is an emerging paradigm that is fostering the connection of massive numbers of devices to the network. It is predicted that by 2020 there will be 50 to 100 billion devices connected to the Internet [21]. These devices can enable new classes of data-driven applications, such as those involving time-critical monitoring and control. Nonetheless, these devices are generating enormous amounts of data at the edges of the infrastructure that need to be seamlessly processed in an efficient and timely manner. Stream processing frameworks (SPFs) have proven to be very effective at processing large amounts of data at near-real time, especially when combined with the elasticity and scalability of the cloud. Nonetheless, existing solutions were developed keeping in mind Big Data streams generated at the core of the infrastructure, such as those associated with web analytics. As a result, applying these solutions to IoT data stream requires transferring data from the edges to a data center located at the core of the infrastructure for processing.

As the number of IoT devices continues to grow, it will become increasingly challenging to support low-latency data processing using models that rely on moving all data between the edge and the core of the infrastructure. Consequently, emerging computational paradigms [5, 23, 24] are exploring different ways of exploiting computational capabilities available at the edges of the infrastructure to better support application needs while reducing the amount of data that needs to be transferred through the network between the edge and the core. Emerging infrastructures are composed of highly distributed and heterogeneous resources with varying service offerings, capabilities, and capacities. Additionally, the characteristics of these resources can vary over time, not only because of the dynamics of the resources due to their changing workloads, but also due to the location of the client – e.g., network latency can vary significantly depending on the geographic location of a client and a specific resource. Furthermore, data processing requirements often depend on the data itself. Understanding how to dynamically discover and aggregate resources and use them to support data-driven processing of data streams is essential for developing a solution that can effectively expand the boundaries of the cloud towards the edges of the infrastructure and enable an efficient end-to-end solution that can process and deliver increasing volumes of data while ensuring quality of service (QoS) and supporting emerging data-driven applications.

In this paper, we propose an edge-based programming framework that allows users to define how data streams are processed based on the content and the location of the data. This enables the definition of data-driven reactive behaviors that can effectively exploit data patterns to dynamically drive stream processing, leveraging resources located at the edges of the infrastructure.

The programming framework builds on a novel architecture that implements a locality-aware stream-processing overlay network, which is able to coordinate the execution of streaming application workflows across geographically distributed stream-processing infrastructures. This overlay integrates a content-based publish/subscribe messaging system that decouples senders from receivers and allows content-based interactions (e.g., discover capabilities, announce resource properties). By decoupling senders from receivers, our approach is able to discover computational resources, data producers, and data consumers at runtime. Content-based interactions are asynchronously coordinated in a distributed manner, improving the scalability and resilience to failures of the overlay. These characteristics make our overlay naturally suited for large-scale, distributed, and highly dynamic systems. Additionally, we propose a locality-aware protocol to seamlessly and effi-

978-1-5090-3047-7/17 $31.00 © 2017 IEEE
DOI 10.1109/ICFEC.2017.18

ciently allocate streaming computations along the data path, from data source to data consumer, allowing us to achieve the desired latency, computation, and throughputs required by data-driven application workflows. Appropriate monitoring and actuation mechanisms ensure the QoS of streaming application workflows during the life of a stream by identifying and adapting to changes in the infrastructure and location of the data producer or data consumer.

We have implemented a prototype of the presented framework and have performed several experiments to evaluate its scalability and efficiency against a more typical single-cloud approach. Using a smart-city application usecase, we illustrate that the presented programming system can support data-driven stream processing using edge resources. In terms of scalability, our experiments show that the system can scale to hundreds of nodes while keeping overheads low. Our experiments also show that our approach can perform up to 56% better than a single cloud approach that does not consider data and user locality.

The main contributions of this work are as follows:

- An edge-based programming framework that allows users to define data-driven reactive behaviors that can effectively exploit data content and location to dynamically and autonomously decide the way a stream is processed.
- A novel architecture for the programming framework that can consider the location of the client and data sources to seamlessly and efficiently allocate streaming computations at the edges of the infrastructure.
- An implementation that creates an overlay network that integrates a content-based publish/subscribe messaging system. This messaging layer decouples senders from receivers, allowing content-based interactions and eases the development of applications by hiding low-level details such as setting up communications between data producers and consumers.
- An experimental validation of the proposed framework using a smart infrastructure scenario demonstrating effectiveness, performance and scalability.

The rest of the paper is structured as follows: Section II describes our driving use case. Section III presents the related work. Section IV presents the design and implementation of our approach. Section V presents a visual example of the orchestration mechanism. Section VI presents an experimental evaluation of our approach. Section VII concludes the work and suggests some future work for this paper.

II. A Motivating Smart City Use Case

Large cities are difficult to navigate, especially for people with special needs such as those with visual impairment, Autism Spectrum Disorder (ASD), or simply those with navigational challenges. The primary objective of this application usecase is to explore the use of IoT capabilities to transform cities around the world into smart cities capable of providing location-aware services (e.g., finding buildings and streets, improving travel experience, obtaining security alerts) [11]. In order to create smart cities that can support reliable navigation services to people with special needs, researchers are creating complex workflows integrating a number of novel IoT elements, including video analytics, Bluetooth beacons, mobile computing, and LiDAR-scanned 3D semantic models. For example, we may have a streaming application workflow that analyzes video feeds from the surveillance cameras of the streets in real-time to evaluate the density of crowds in different parts of the city to help select path choices. Specially, ASD individuals may prefer to choose paths that have less dense crowds due to psychological factors; people with visual impairment try to avoid large open spaces due to the difficulty of finding references for localization; and people in wheelchairs can navigate along paths with fewer crowds far more conveniently than along those with large crowds. This information is then combined with a 3D model and the location of the user to calculate the best path to reach the desired destination. Additionally, we need to continuously monitor the user (e.g., using the Bluetooth beacons), and the streets (e.g., using surveillance cameras) to adapt to changes.

Data-driven workflow, such as the one described above, are very latency sensitive. In our use case, the navigation path needs to be computed in a timely manner to improve the quality of experience and allow users to meet planned schedules (for example, arrive in time to take a specific bus). In some cases we might need to adapt the path based on users' feedback. For example, if an ASD user gets stuck and panics at a certain location, the data-streaming application has to react following pre-defined or learned strategies such as re-route the path to avoid a current crowd, or move them to certain intermediate location to make them wait until the crowd passes.

Supporting workflows that require analyzing real-time video analytics from a public space to a cloud-centric approach require the need of transferring all the raw data to the cloud. This can lead to extra latencies that can affect users' quality of experience and may also result in privacy concerns.

This usecase is based on a real project at Rutgers University titled "Building Smart Transportations Hubs with Internet of Things to Improve Services to People with Special Needs", which helps people with special needs navigate through large transportation hubs. We have extended it using IoT capabilities so as to assist people with special needs in larger spaces such as cities, and then we used it to evaluate the effectiveness and performance of our approach.

III. Related Work

Distributed stream-processing systems have been explored in literature as a way of collecting, processing, and aggregating data across large numbers of real-time data streams. Pietzuch et al. [22] proposed the approach of a stream-based overlay network (SBON) to efficiently place computation across geographically distributed resources considering knowledge of stream, network, and node conditions, and to continuously optimizes placement without global knowledge of the system. Other approaches such as [4, 9] optimize the placement of computation by balancing the load through the infrastructure. Borealis [1] relies on a static placement of services in order

to use distributed resources. Ahmad et al. [2, 3] extended Borealis to enable dynamic placement of services considering the bandwidth usage of a query. In general, these approaches rely on pre-existing knowledge of the infrastructure and on statically defined data pipelines to perform efficient workload allocation across geographically distributed resources.

In the context of sensor networks, there has been extensive work on enabling dynamic computation of real-time streams in a distributed manner. Madden et al. [20] proposed TinyDB to address acquisitional issues in sensor networks of when, where, and in what order to sample and which samples to process must be considered. Other approaches focus on enabling in-network processing within sensor network devices [13, 19].

Frameworks such as Heron [18], Storm [25], Neptune [6], Flink [7], and Spark [26] allow the processing of large amounts of data very efficiently. Solutions created based on these approaches typically consider a single data center and statically defined workflows and they are complementary to our work.

Unlike previous approaches, we propose a framework that allows users to specify the way streams are processed using the content of the data. Moreover, our framework creates an abstraction using a publish/subscribe overlay network that can discover available computational resources and allocate the computation in the most appropriated one (e.g., closest to the data source and client) at runtime. These computational resources can involve other distributed stream-processing frameworks such as those described in this related work (e.g., Storm, Spark).

Rendezvous-based models (AR) [14] enable programmable reactive behaviors at RPs using the action field within a message. Interactions in the AR model are symmetric, allowing participants to simultaneously be information producers and consumers. Other content-based decoupled interaction systems are based on publish-subscribe-notify models (PSN)[10] and include implementations such as Sienna [8] and Gryphon [12]. In this work we build on top of Meteor [14] and adapt its primitives to our model.

IV. Data-driven Stream Processing Framework

In this work we propose a framework that allows users to programmatically define stream-processing workflows as reactive behaviors based on the content of the streaming data. As such, data streams are evaluated at runtime to decide how and where to process their data. Our approach is also location-aware and can leverage resources located at the edges of the infrastructure as a way of reducing latencies in processing the data. Specifically, it can provision geographically distributed resources based on user objectives, service availability, and data locality. Figure 1 presents a schematic overview of the system. The key layers are described below.

Infrastructure layer: The infrastructure layer is composed of data and computational resources. The data can be generated by various streaming sources, including IoT devices (e.g., cameras, smart watch, and smart infrastructure), traffic information, weather data, etc. The computational resources can also be heterogeneous and distributed through the infrastructure, from the core to the edges. These computational resources can be part

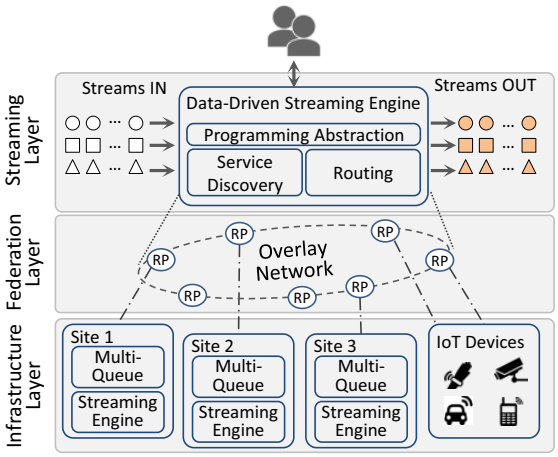


Fig. 1: Schematic overview of the system architecture.

of public and private clouds, network data centers, cloudlets, and edge clouds. In our approach, each data center or site has a multi-queue system to classify and isolate different types of streams (e.g., Apache Kafka [17]) and a stream-processing engine that processes data elements from the different queues (e.g., Apache Kafka Streams [1], Apache Storm [17]). Stream-processing engines allow some applications to easily exploit a limited form of parallel processing. Given a sequence of data (a stream), a series of operations (kernel functions) is applied to each element of the stream; these types of operations are typically represented by a direct acyclic graph (DAG) and define the application workflow. These workflows are also known as topologies.

Federation layer: The federation layer is responsible for orchestrating the geographically distributed resources composing the infrastructure. This layer is built using two main components. First, it creates a peer-to-peer overlay network that interconnects sensor/actuator services, computational resources, and data. In our per-to-peer overlay network we have four distinct roles: Consumers, Producers, Users and RP. Consumers are responsible for consuming data (Storm or Kafka Streams) and they are located at the RP nodes. The Producers are sensors geographically distributed and they produce some type of data. Users can be producers and consumers at the same time. Finally, RP are geographically distributed physical nodes that are in charge of performing the computations. On top of this overlay, we implement an associative rendezvous messaging substrate (ARMS), which implements the Associative Rendezvous (AR) interaction model. AR is a paradigm for content-based decoupled interactions with programmable reactive behaviors [15, 16]. Rendezvous-based interactions provide a mechanism for decoupling senders and receivers in both space and time. In this way, we can discover available resources, data sources, and services.

To enable these interactions, each RP has two components: the profile manager and the matching engine. The matching engine component is essentially responsible for matching pro-

[1] http://kafka.apache.org/documentation/streams

files. An incoming message profile is matched against existing interest and/or data profiles depending on the desired reactive behavior. If the result of the match is positive, then the action field of the incoming message is executed first followed by the evaluation of the action field in matched profiles. The profile manager manages locally stored profiles and monitors message credentials and contexts to ensure that related constraints are satisfied. For example, a client cannot retrieve or delete data that she/he is not authorized to access. Section IV-A describes in detail our specific ARMS implementation.

Streaming layer: The streaming layer provides users and applications with efficient data-driven access to federated resources. This layer is composed of three components: a programming abstraction, a data-driven service discovery, and a routing engine. The programming abstraction allows users to define content-based reactive actions to decide at runtime how to process a data stream. In this way, users can dynamically express which workflow or topology is required to process a stream by looking at the content of the data, and it can also trigger additional processing workflow upon the detection of a specific data pattern. This programming abstraction uses the mechanisms provided by the ARMS substrate to trigger a processing event. The next component is a data-driven service discovery, which allows us to identify services, data sources, and computational resources that are relevant to a request placed by a user. This is performed by taking the user's information (e.g., location, requested operation, preferences) and querying the federation using ARMS mechanisms. Once all relevant services are discovered, the routing component combines the user's information with knowledge of the infrastructure (i.e. computational resources and data sources) to transparently decide where to perform the required computation. Section V describes in detail all the steps involved in this process.

A. Semantics of the Programming Abstraction

In this section, we describe the semantics and mechanisms of our programming abstraction, which enable users to define content-based interactions. This abstraction is based on the associative rendezvous model. We describe the three main elements realizing our implementation of the associative rendezvous model, namely Messages, Associative Selection, and Reactive Behaviors. We have specifically designed these elements to enable the orchestration of location-aware computations.

AR Messages: An AR message is defined as the quintuplet: (header, action, data, location, topology). The data, location, and topology fields may be empty or they may contain a message payload with the location of the user or the topology to be uploaded. The header includes a semantic profile in addition to the credentials of the sender. A profile is a set of attributes and/or attribute-value pairs that define not only properties but also the recipients of the message. Attribute fields must be keywords from a defined information space, while value fields may be keywords, partial keywords, wildcards, or ranges from the same space. At rendezvous points, profiles are classified as data profiles or as interest profiles, depending on the action field of its associated message. The action field of a message defines its reactive behavior when a matching occurs at a rendezvous point.

TABLE I: Reactive behaviors

Actions	Semantics
store	Store data in rendezvous point queue.
retrieve	Match message profile with existing data profiles; Send data corresponding to each matching data profile to the sender.
statistics	Notify sender the current rendezvous workload.
profiles	Notify sender all the interest_profiles stored in that rendezvous point.
update	Notify receiver the new RP that needs to start communicating.
submit_topology	Store the new topology in RP and notify required RP nodes.
notify_data	Match message profile with existing data/interest profiles.
notify_interest	Notify sender if there is at least one match.
delete_data	Match message profile with existing data/interest profile.
delete_interest	Notify sender if there is at least one match. Remove all matching data/interest profiles from the system.

(a) (b)

Fig. 2: Sample message profiles: (a) a data profile for a sensor; (b) an interest profile for a client.

A sample data profile used by a sensor to publish data is shown in Figure 2a, and a matching interest profile is shown in Figure 2b. Note that the number or order of the attribute/attribute-value pairs in a profile is not restricted however, our current prototype requires that the maximum possible number of attribute/attribute-value pairs must be predefined. The action field of the AR message defines the reactive behavior at the rendezvous point and is described later in this section.

The AR interaction model defines a single symmetric post primitive. To send a message, the sender composes a message by appropriately defining the header, action, data, location, and topology fields, and invokes the post primitive. The post primitive resolves the profile of the message and delivers the message to relevant rendezvous points. The profile resolution guarantees that all rendezvous points that match the profile will be identified. Nonetheless, the actual delivery relies on existing transport protocols.

Associative Selection: Profiles are represented using a hierarchical schema that can be efficiently stored and evaluated by the selection engine at runtime. A profile p represents a path in the hierarchical schema, $[e_0 \triangle ... e_k]$, where e_i is an element operand and $\triangle$ can be a parent-child ("/") operator (i.e. at adjacent levels) or an ancestor-descendant ("//") operator (i.e. separated by more than one level). Within a level, the profile defines a propositional expression where $\triangle$ represents propositional operators, such as $\wedge$ and $\vee$ between elements at the same level. Note that the propositional expression at a level must evaluate to TRUE for the evaluation to continue to

the next level. The elements of the profile can be an attribute, e_i : (a_i), or an attribute-value pair e_i : (a_i, v_i), where a_i is a keyword and v_i may be a keyword, partial keyword, wildcard, or range. The singleton attribute a_i evaluates to true if and only if p contains the simple attribute a_i. The attribute-value pair (a_i,u_i) evaluates to true with respect to a profile p, if and only if p contains attribute a_i and the corresponding value v_i satisfies u_i, e.g. v_i = computer and u_i = comp*. For example, in Figure 2, the profile 2a is associatively selected by the profile 2b, since (1) both have matched singleton attribute videocamera; (2) for attribute location, 100 - 105, satisfies the range relation; (3) for attribute type, *Public* matches the wildcard *; and (4) quality=720p satisfies the request quality>=720p. A key characteristic of the selection process is that it does not differentiate between interest and data profiles. This allows all messages to be symmetric where data profiles can trigger the reactive behaviors of interest messages and vice versa. The matching system combines selective information dissemination with reactive behaviors.

Reactive Behaviors: The action field of a message defines its reactive behavior at a rendezvous point. Basic reactive behaviors currently supported include *store, statistics, profiles, update, submit topology, notify*, and *delete*, as shown in Table I. The *notify* and *delete* actions are explicitly invoked on a data or an interest profile. The *store* action stores data and data profiles in the appropriated rendezvous point queue. It also causes the message profile to be matched against existing interest profiles and associated actions to be executed in case of a positive match. The *statistics* action queries the system to retrieve detailed information about the characteristics and the status of the computational resources. The *profiles* action matches its associated message profile against existing interest profiles and reports the results to the sender. The *update* action requests to start streaming to a new RP. The *notify* action matches the message profile against existing interest/data profile and notifies the sender if there is at least one positive match. The *submit_topology* action allows users to submit and store topologies into a select number of RPs, allowing us to have a distributed store where users can share and discover existing topologies previously uploaded by other users. This avoids the need of rewriting the same topology multiple times and facilitates the reproducibility of experiments. Finally, the *delete* action deletes all matching interest/data profiles. Actions are only executed when the message header contains an appropriate credential. In the case of multiple matches, the profiles matching are processed in a random order. By default, all matched profiles are returned. The programmable reactions can be used to define other behaviors: for instance, any one of the matched profiles is returned.

B. Implementation Overview

The current implementation builds on Project JXSA, the open source Java version of the JXTA [2] protocol, a general-purpose peer-to-peer framework that provides a set of open protocols and platforms to build new services and applications. JXTA defines concepts, protocols, and a network architecture. JXTA concepts include peers, peer groups, advertisements, modules, pipes, rendezvous, and security. JXTA defines protocols for (1) discovering peers (Peer Discovery Protocol, PDP), (2) binding virtual end-to-end communication channels between peers (Pipe Binding Protocol, PBP), (3) resolving queries (Peer Resolver Protocol, PRP), (4) obtaining information on a particular peer, such as its available memory or CPU load (Peer Information Protocol, PIP) (5) propagating messages in a peer group (Rendezvous protocol, RVP), (6) determining and routing from a source to a destination using available transmission protocols (Endpoint Routing Protocol, ERP). Note that JXTA is an application-level technology and does not introduce any limitations on the underlying infrastructure or the routing protocols.

The overall operation of the overlay consists of two phases: bootstrap and running. During the bootstrap phase (or join phase), messages are exchanged between a joining RP and the rest of the group. During this phase, the joining RP attempts to discover RPs already existing in the system to build its routing table. The joining RP sends a discovery message to the group. If the message is unanswered after a set duration (in the order of seconds), the RP assumes that it is the first in the system. If an RP responds to the message, the joining RP and the rest of the RPs update their routing tables.

The running phase consists of stabilization and user modes. In the stabilization mode, an RP responds to queries issued by other RPs in the system. The purpose of the stabilization mode is to ensure that routing tables are up to date and to verify that other RPs in the system have not failed or left the system. In the user mode, each RP interacts at the ARMS layer. The ARMS matching engine at each RP is based on MongoDB [3], a NoSQL database.

Once the system is operating in user mode, RPs allow external entities to use the ARMS layer to communicate with each other to offer and request data and computation. These entities can include: a) users that might want to retrieve specific data or perform certain computation over data found using a query; b) IoT devices that can produce and consume data based on specific interests; and c) computational resources, such as data analytics and streaming platforms, clouds, or high performance computing clusters that offer their computational capabilities.

V. Realizing Data-driven Edge Stream Processing

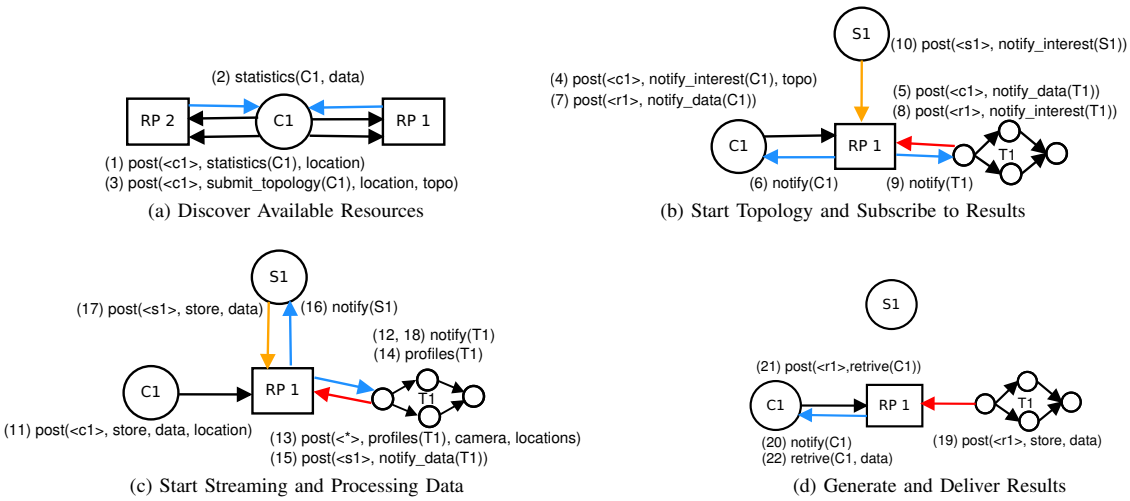
This section describes the steps involved in the orchestration mechanisms required to realize our vision of data-driven edge stream processing.

A. Basic Functionality

In this section we use the smart city scenario to detail the basic functionality of our system. We consider that a user installs our client in her/his phone and provides personal preferences. Moreover, the user has to indicate her/his destination.

[2]https://jxse.kenai.com/

[3]http://www.mongodb.com



(a) Discover Available Resources

(b) Start Topology and Subscribe to Results

(c) Start Streaming and Processing Data

(d) Generate and Deliver Results

Fig. 3: Example Illustrating the Basic Operations of Our System using Associative Rendezvous.

Next, the application interacts with the rest of the framework transparently. Figure 3 depicts the interactions required for this scenario. In Figure 3, client C1 is a user with visual impairment (C1 acts as producer and consumer at the same time) that is interested in finding a route from her/his location to a specific destination. We describe the steps involved in computing a client's request. Note that all the steps described below are performed in an asynchronous manner to facilitate the scalability of the system.

Figure 3(a) depicts the steps involved in a location-based discovery of available resources. First, client C1 sends a message indicating that she/he wants to be notified with the resource information of all Rendezvous Points (RPs) that meet her/his interest profile $\langle c1 \rangle$, which in this case involves his/her location (Figure 3(a) message (1)). Next, in an asynchronous manner, any RP that meets the criteria specified in $\langle c1 \rangle$, are notified and consequently all notified RPs send their resource information to client C1 (message (2)) – resource information can include current resource availability, statistical performance information, and hardware details. Once client C1 has received a certain number of answers, the system decides which RP is the best fit to use for processing her/his workload. This decision can be made considering different metrics, such as closest RP, highest performance, and lowest failure rate. Additionally, the client can perform an optional step, which involves registering a topology and its associated metadata to process her/his workload in a customized manner. Once a topology is registered in the system, any client can discover it and use it by simply querying the system.

Next, Figure 3(b) shows how the client starts a topology and subscribes to receive results generated by such topology. Client C1 sends a request with profile $\langle c1 \rangle$ to start a topology in the stream engine associated with the previously selected RP (message (4)). This message also allows the client to subscribe to updates from the topology. After topology T1 is deployed and ready to accept workload, the topology notifies its associated RP (message (5)). Since message (5) matches the profile $\langle c1 \rangle$, client C1 is notified that topology T1 is ready to start processing data and generating results (message (6)). Then, client C1 sends another request with data profile $\langle r1 \rangle$, requesting to be notified if there are results (i.e. navigation data) stored in the system matching her/his $\langle r1 \rangle$ profile (message (7)). C1's interest profile is stored in the system and matched against existing profiles as well as the ones arriving in the future. Topology T1 sends a message requesting to be notified if anyone is interested in receiving the results that it generates (message (8)). Since this interest matches the profile $\langle r1 \rangle$, T1 is notified that client C1 is subscribed to its results and therefore it can send data to the system. Note that the order in which messages (7) and (8) reach the system is irrelevant, as the subscriptions are persistent in time.

Additionally, Figure 3(b) shows a Camera sensor S1, which can produce data described by the profile $\langle s1 \rangle$. S1 publishes data in the system only if there are consumers that request it. Hence, sensor S1 sends a message with interest profile $\langle s1 \rangle$, requesting to be notified if there are clients interested in the type of data it produces (message (10)). S1's interest profile is stored in the system, and matched against existing profiles as well as any profile arriving in the future.

Figure 3(c) shows the steps involved in streaming and processing data. After C1 is notified that T1 is ready to accept data for processing (Figure 3(b) message (9)), C1 starts streaming data in the system (message (11)). This data can involve location, destination, special needs, preferences, etc. The data published by C1 match T1's profile, whose action filed is executed, resulting in a notification being sent to T1 (message (12)). The streaming topology processes the data coming for the client, resulting in a set of valid paths that can lead client C1 to her/his destination. This information can also be used to identify which camera sensors contains relevant data in order to select the best path matching client's preferences. Hence, topology T1 sends a message requesting

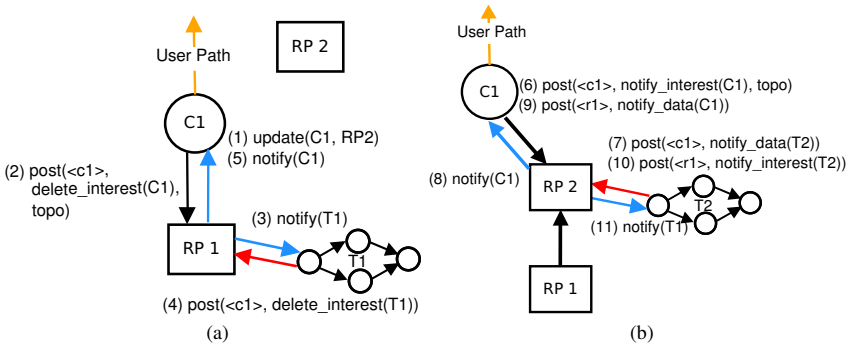


Fig. 4: Example Illustrating Location-aware Operations.

all interest profiles of type camera that are located along the previously identified paths (message (13)). The system finds camera sensors that overlap with the specific interests and sends them to T1 (message (14)). Next, T1 sends notify messages to subscribe to data of all cameras of interest, e.g., T1 sends a notify data message with profile $\langle s1 \rangle$ to subscribe to S1 (message (15)). Since T1's profile matches S1's profile, the action field for S1's profile is executed and a notification message is sent to S1 (message (16)). S1 starts publishing data in the system (message (17)). Since T1's profile matches S1's profile, the action field for T1's profile is executed and a notification message is sent to T1 (message (18)). T1 starts consuming and processing data.

Finally, Figure 3(d) shows how the client receives the result of her/his query. Once T1 has processed the data, the result of the computation is published and stored with profile $\langle r1 \rangle$ (message (19)). Since T1's profile matches C1 profile's subscription (Figure 3(b) message (7)), the action field for C1's profile is executed, resulting in a notification message being sent to C1 (Figure 3(d) message (20)). C1 retrieves the data by placing a request (message (21)), consequently the RP sends the requested data to C1 (message (22)).

B. Location-aware Functionality

Figure 4 depicts how streaming computations associated with a particular client can be transparently moved across sites following the client, and ensuring that data is always processed in the best resource.

In Figure 4(a), client C1 is a user with visual impairment navigating to its final destination. Client C1 used rendezvous point RP1 to perform her/his initial streaming computations. While client C1 is moving away from RP1 towards her/his destination, RP1 is periodically checking the current location of the client C1 to identify if there is a most suitable location to process C1's data. Eventually the client C1 gets far enough from the RP1 that the system decides to relocate her/his streaming computation to another RP. Then, RP1 sends a message with action update to client C1 that indicates the RP she/he needs to transition to (message (1)). Client C1 sends a message to stop the topology associated to its profile (message (2)). The RP1 notifies topology T1 that it needs to stop (message (3)). Topology T1 unsubscribes its results as it is no longer generating them (message (4)) and it deactivates itself. Then RP1 notifies client C1 that her/his interest has been deleted and it can proceed to migrate to the new RP point.

Figure 4(b) shows how the RP1 transfer client C1's status to RP2. The rest of the scenario follows as in Figures 3(b)-(d), where the client request to start a topology (i.e. T2) and subscribe to results from said topology.

VI. Experimental Evaluation

We have deployed our edge streaming engine framework on a cluster of 32 Intel(r) Xeon(r) e5620 2.40 GHz computers with 25 GB of memory, and a 1 Gbps Ethernet interconnection. In our experiments, each rendezvous point has a multi-queue based on Kafka [17] version 2.10-0.9.0.0, and a streaming engine based on Storm [25] version 0.9.3. Our implementation of the overlay network and ARMS uses JXSA [4] version 2.6.

We have performed two set of experiments to evaluate the proposed framework. First, we evaluated the scalability of our framework and the overheads involved in the basic orchestration mechanisms. Second, we emulated a real scenario to compare our locality-aware approach versus a single cloud approach.

A. Scalability Experiments

To evaluate the scalability of the framework, we distributed RPs among the physical machines of our cluster to emulate an environment with multiple peers. These experiments were aimed at understanding the overheads involved in the basic orchestration mechanisms of our framework. We performed several experiments in which the system had a different number of RP nodes and amounts of available data.

1) Querying Operation: The first set of experiments measured the overhead involved in performing a location-based query to identify relevant resources around the location of a client node. In our experiments, the location-based query used the GPS coordinates of each RP node and the client to identify the RP around the client. As we described in Figure 3a, a location-based query involves sending *statistics*

[4]https://jxse.kenai.com/

action messages to all RP with a matching location, which ask RPs to send their information to the client. In these experiments, we only measured the overhead incurred in querying the database and constructing the "notify_interest" message, i.e., the notification delivery was not measured. For these experiments, we had a total of 1000 RP nodes and we varied the position of the client, resulting in a different number of RP nodes matching the location requirement. Figure 5 collects the results of these experiments.

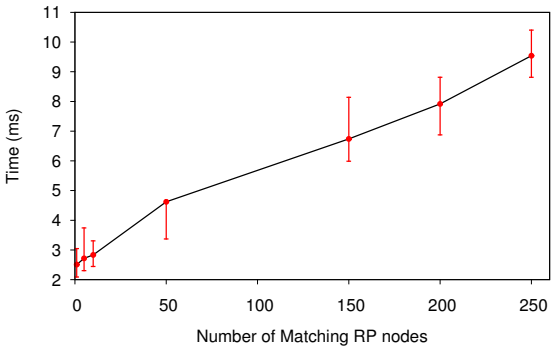


Fig. 5: Location-based query overhead for different number of matches per query.

Figure 5 shows that the overhead of performing a location-based query increases linearly with the number of matching RP nodes. Results show a maximum overhead of 10 milliseconds, which occurred when the query found 250 RP nodes being within the specified location.

The second set of experiments measured the overhead involved in the profile matching operation at an RP node. We used a *notify_data* action message for these experiments. We considered different scenarios in which the system had different number of data profiles (i.e. subscriptions) and the request returned a different number of profile matches. As we described in Figure 3b, a *notify_data* request asks an RP to find matching profiles and to send a *notify* message to the profile owners – e.g., requesting to start streaming data. In this experiment, we measured the overhead of performing the matching operation and constructing the "notify" messages. The experiment was conducted using profiles similar to the one described in Figure 2, which contains sets of complex keyword tuples containing wildcards and/or ranges. Figure 6 collects the results of these experiments.

We can observe in Figure 6 that, for a moderate database size of up to 10^4 profiles, the profile-matching operation incurs in an overhead between five and 10 milliseconds. Nonetheless, when we increase to 10^5 profiles, the overhead increases significantly, as could have been expected given the memory and data access times required to identify and retrieve such a large number of profiles. As a consequence, we need to ensure that the number of profiles that each RP manages is within a manageable range to minimize the overheads. This can be achieved by implementing load-balancing techniques and limiting the knowledge of each RP to its closest neighbors.

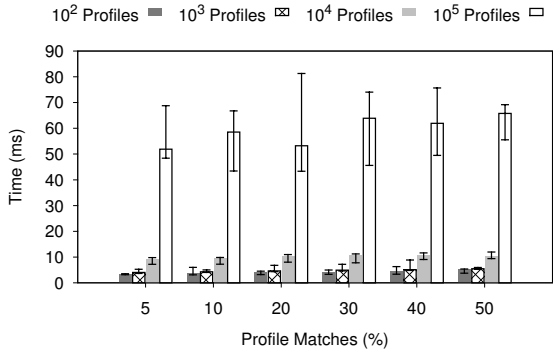


Fig. 6: Profile matching overhead. Results are grouped based on the matching ratio expressed as a percentage of the total number of stored profiles.

2) Propagating New Information through the System: First we performed a set of experiments to understand the behavior of the system when multiple RP nodes request to join an existing deployment of our system. Specifically, we measured the time from when the RP nodes sent their joint message until all RP nodes existing in the system were updated. In our experiments, we varied both the number of RP nodes wanting to join and the number of existing RP nodes in the system. Figure 7 collects the results.

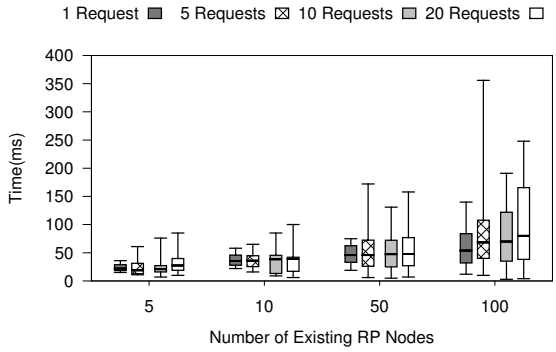


Fig. 7: Time neccessary to update information of all RP nodes in the system after multiple new RP nodes join.

Figure 7 shows that the time required to have a consistent view of all the new RP nodes after the requests are placed is relatively low, and that it scales properly when increasing the number of existing RP nodes. In the case of a small system with only five existing RP nodes the overhead is around 20 milliseconds on average. The overhead increases to around 70 milliseconds in the case of having a large system with 100 existing RP nodes. Additionally, we can observe that varying the number of requests does not significantly affect the time required to propagate the information in a system with up to 50 RP nodes. In the case of having 100 RP nodes, we observe that increasing the number of requests, increases the overhead as well as the dispersion around the average. Although it is important to maintain an updated view of the system to

ensure that we can efficiently process clients' requests, it is not critical for the regular operations of the system. Our system has mechanisms to adapt to changes in the availability of RP nodes to improve data processing efficiency. Therefore, we consider that having an eventually consistent system is sufficient.

Next, we measured the time required to propagate and store the information of a new topology across all RP nodes in the system. In these experiments, we considered several scenarios involving different number of RP nodes as well as different number of concurrent requests to register new topologies. Figure 8 collects the results.

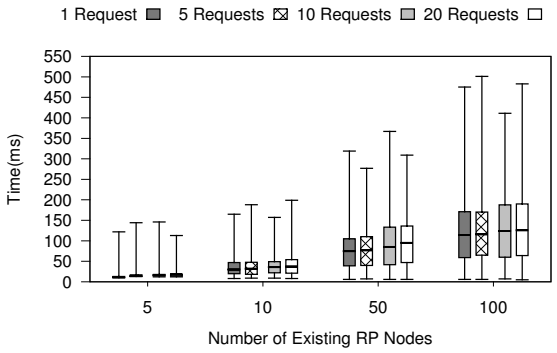


Fig. 8: Time neccessary to update information of all RP nodes in the system after multiple topologies are registered.

We can observe that in Figure 8 that the time required to propagate and store the information of new topologies across all the RP nodes in the system is relatively low. In particular, it can vary from a few milliseconds to around 100 milliseconds on average when having a large system with 100 RP nodes. The main factor affecting the time required to propagate the information is the number of RP nodes as each individual message is relatively small (around 5KB). As in the case of adding RP nodes to the system, we also consider that updating the information about available topologies is eventually consistent. Moreover, in a real scenario, not every RP node will have the information of every topology, as some of them might not be relevant to all RP nodes.

B. Performance experiments

In this section we have performed a set of experiments to compare the proposed edge stream processing approach with a traditional approach in which the stream processing is located in a fixed location at the core of the infrastructure. To perform these experiments we deployed two streaming frameworks (based on Apache Kafka plus Apache Storm), one in our local cluster and another one in Amazon AWS. In both cases, Apache Kafka and Storm were deployed in a single machine to avoid any additional latency due to communication across machines of a Kafka or Storm cluster. In Amazon AWS, we used an instance of type t2.2xlarge, which has similar characteristics to the machines in our local cluster (8 CPUs and 32 GB of memory). Note that we do not assume that the edge will be equipped with similar characteristics, we are only using this type of instance for computational comparison only. When using our approach, our framework was deployed across both infrastructures – i.e. we had two RP nodes exposing the capabilities of each stream engine. We emulated a scenario inspired by the one described in Section II in which a client performs a request to obtain navigation indications. The data-processing workflow (i.e. topology) was composed by three sequential steps (i.e. bolts in Apache Storm terms) that performed some basic data transformations that did not require significant computational time. In our experiments we evaluated the performance of both approaches using three different anonymized workloads consisting of 100, 1000, and 10000 data elements. Each data element is around 1KB of size and is generated sequentially. The client and data source were close to our local cluster. Figures 9 and 10 collect the results of these experiments.

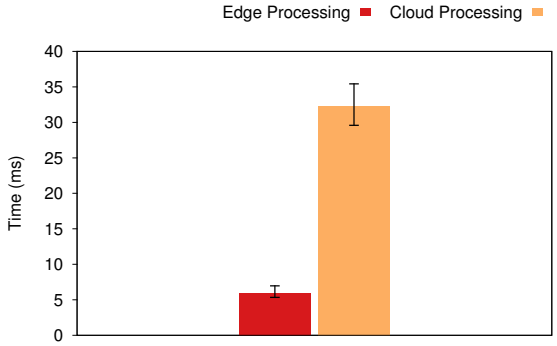


Fig. 9: Delay in processing a data element.

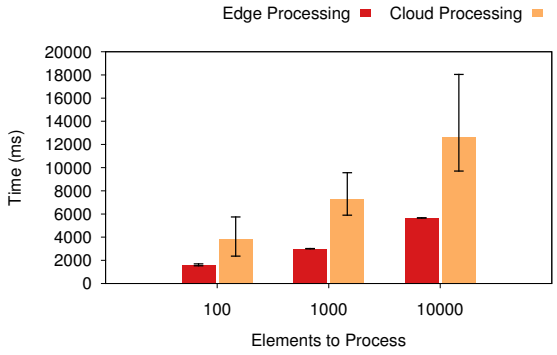


Fig. 10: Completion time for different workloads.

Figure 9 shows the delay in processing a data element. The delay is measured as the time spent between the data producer generates the data element and the destination queue receives it for processing. We observe that in our case, the latency is up to 78% less than a traditional approach consisting of sending data for processing to a cloud. The main reason is that our approach is able to dynamically identify different sources of computation and data using the location of the client. Therefore, it chooses to process the data in the place that is closest to the data source and the client. Our approach still incurs some small overheads

as we have to identify the best processing site and data has to travel from source to destination. Nonetheless, these overheads are negligible compared with those incurred when transferring data between the edge and the core of the infrastructure.

Figure 10 collects the computational time required to process a stream of data elements. We can observe that our approach is up to 56% more efficient than a traditional cloud-based approach. In this particular case, this is directly related to the delay mentioned before that the system incurs when processing each data element. Since the delay of each data element is larger than its computational time, the system is often idle waiting for small periods of time until a new data element arrives.

VII. Conclusion and Future Work

In this paper we presented a data-driven programming framework that allows users to decide at runtime the transformations that are required to process a given data stream by looking into the data itself. The proposed framework has location-aware mechanisms capable of taking information related to the location of a client and data sources and seamlessly and efficiently allocating streaming computations at the edges of the infrastructure. We implemented a prototype of our data-driven stream processing framework and experimentally evaluated its performance. We observed that our framework is able to handle hundreds of nodes and perform data-matching operations in thousands of data profiles in the order of milliseconds. Moreover, we emulated a real smart city usecase by deploying two geographically distributed stream-processing engines. In this scenario, our approach showed to be up to 56 % more efficient than a single cloud approach.

Currently, we are exploring various issues to extend our current approach. We would like to introduce access control layer to enhance privacy and security of user and prevent exposing information about other users. Additionally, we would like to include some proactive migration mechanisms to reduce overheads and delays in the application. Since we know the route that each user is taking, we can distribute computation along multiple RP that intersect with a user's path.

Acknowledgements: This work is supported in part by NSF via grants numbers ACI 1339036, ACI 1441376. The research at Rutgers was conducted as part of the Rutgers Discovery Informatics Institute (RDI2).

References

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, M. Cherniack, J. hyon Hwang, W. Lindner, A. S. Maskey, E. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the borealis stream processing engine. In *In CIDR*, pages 277–289, 2005.

[2] Y. Ahmad and U. Cetintemel. Network-aware query processing for stream-based applications, 2004.

[3] Y. Ahmad, U. Cetintemel, J. Jannotti, A. Zgolinski, and S. Zdonik. Network awareness in internet-scale stream processing. In *IEEE Data Engineering Bulletin*, 2005.

[4] M. Balazinska, H. Balakrishnan, and M. Stonebraker. Contract-based load management in federated distributed systems. In *1st Conf. on Symposium on Networked Systems Design and Implementation*, pages 15–15, Berkeley, CA, USA, 2004.

[5] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli. Fog computing and its role in the internet of things. In *MCC Workshop on Mobile Cloud Computing*, MCC '12, pages 13–16, 2012.

[6] T. Buddhika and S. Pallickara. NEPTUNE: real time stream processing for internet of things and sensing environments. In *2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, May 23-27, 2016*, pages 1143–1152, 2016.

[7] P. Carbone, S. Ewen, S. Haridi, A. Katsifodimos, V. Markl, and K. Tzoumas. Apache flink: Stream and batch processing in a single engine. *Data Engineering*, page 28, 2015.

[8] A. Carzaniga and A. L. Wolf. Content-based networking: A new communication infrastructure. In *Revised Papers from the NSF Workshop on Developing an Infrastructure for Mobile and Wireless Systems*, pages 59–68, 2002.

[9] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, , and S. Zdonik. Scalable distributed stream join processing. In *Conference on Innovative Data Systems Research*, 2003.

[10] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003.

[11] J. Gong, C. Feeley, H. Tang, G. Olmschenk, V. Nair, Z. Zhou, Y. Yu, K. Yamamoto, and Z. Zhu. Building smart transportation hubs with internet of things to improve services to people with special needs. Technical report, Department of Civil Engineering, Rutgers University, 2016.

[12] IBM. Gryphon: publish/subscribe over public networks. white paper. 2007.

[13] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *Proceedings of the 6th Annual International Conference on Mobile Computing and Networking*, MobiCom '00, pages 56–67, 2000.

[14] N. Jiang, A. Quiroz, C. Schmidt, and M. Parashar. Meteor: a middleware infrastructure for content-based decoupled interactions in pervasive grid environments. *Concurrency and Computation: Practice and Experience*, 20(12):1455–1484, 2008.

[15] N. Jiang, Schmidt, and Parashar. A decentralized content-based aggregation service for pervasive environments. In *ACS/IEEE International Conference on Pervasive Services*, pages 203–212, 2006.

[16] N. Jiang, C. Schmidt, V. Matossian, and M. Parashar. Enabling applications in sensor-based pervasive environments. In *In Proc. of BaseNets 2004, SanJose,CA*, pages 871–883, 2004.

[17] J. Kreps, L. Corp, N. Narkhede, J. Rao, and L. Corp. Kafka: a distributed messaging system for log processing. netdb'11, 2011.

[18] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter heron: Stream processing at scale. In *ACM SIGMOD Intl. Conf. on Management of Data*, pages 239–250, 2015.

[19] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *ACM SIGMOD Intl. Conf. on Management of Data*, pages 49–60, 2002.

[20] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tinydb: An acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1), Mar. 2005.

[21] D. McAuley, R. Mortier, and J. Goulding. The dataware manifesto. In *2011 Third International Conference on Communication Systems and Networks (COMSNETS 2011)*, pages 1–6, 2011.

[22] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-aware operator placement for stream-processing systems. In *Intl. Conf. on Data Engineering*, pages 49–, 2006.

[23] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The case for vm-based cloudlets in mobile computing. *IEEE Pervasive Computing*, 8(4):14–23, 2009.

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

[25] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy. Storm@twitter. In *ACM SIGMOD Intl. Conf. on Management of Data*, pages 147–156, 2014.

[26] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In *USENIX*, 2012.