Demo: The RAINBOW Analytics Stack for the Fog Continuum

Moysis Symeonides*, Demetris Trihinas†, Joanna Georgiou*, Michalis Kasioulis*, George Pallis*, Marios D. Dikaiakos*, Theodoros Toliopoulos‡, Anna-Valentini Michailidou‡, Anastasios Gounaris‡

* Department of Computer Science
University of Cyprus
Email: {msymeo03, jgeorg02, mkasio01, gpallis, mdd}@cs.ucy.ac.cy

† Department of Computer Science
University of Nicosia
Email: trihinas.d@unic.ac.cy

‡ Department of Informatics
Aristotle University of Thessaloniki
Email: {tatoliop, annavalen, gounaria}@csd.auth.gr

I. Introduction

With the proliferation of raw Internet of Things (IoTs) data, Fog Computing is emerging as a computing paradigm for delay-sensitive streaming analytics with operators deploying big data distributed engines on Fog resources [1]. Nevertheless, the current (Cloud-based) distributed analytics solutions are unaware of the unique characteristics of Fog realms. For instance, task placement algorithms consider homogeneous underlying resources without considering the Fog nodes' heterogeneity and the non-uniform network connections, resulting in sub-optimal processing performance. Moreover, data quality can play an important role, where corrupted data, and network uncertainty may lead to less useful results. In turn, energy consumption can critically impact the overall cost and liveness of the underlying processing infrastructure. Specifically, scheduling tasks on nodes with energy-hungry profiles or battery-powered devices may temporarily be beneficial for the performance, but it may increase the overall cost, or/and the battery-powered devices may not be available when needed. A Fog-enabled analytics stack must allow users to optimize Fog-specific indicators or trade-offs among them. For instance, users may sacrifice a portion of the execution performance to minimize energy consumption or vice versa. Except for the performance issues raised by Fog, the state-of-the-art distributed processing engines offer only low-level procedural programming interfaces with operators facing a steep learning curve to master them. So, query abstractions are crucial for minimizing the deployment time, errors, and debugging.

Towards that, we introduce the **RAINBOW Analytics Stack**, which offers a holistic approach for real-time data management and processing for Fog realms. Moreover, it provides a distributed solution encapsulating pluggable task schedulers that optimize user-defined trade-offs among performance indicators, like energy consumption, processing latency, data quality, etc. Finally, the RAINBOW declarative query model enables users to express streaming analytic insights, leaving the RAINBOW stack to compile, optimize, and execute them. Table I shows a comparison between features of RAINBOW Analytics Stack and other state-of-the-art streaming platforms.

II. The RAINBOW Analytics Stack

A. Assumptions & Preconditions

The RAINBOW Analytics Stack assumptions and operational preconditions are as follows:

Control Plane: The RAINBOW Analytic Stack utilizes a controller-worker architectural paradigm, in which the controller (or Control Plane) represents a single point of communication between users and distributed storage and processing. Control Plane is responsible for service discovery, health checking, task placement, and execution coordination. Due to the critical and resource demanding nature of these procedures, the Control Plane has to be deployed on a host node with increased processing capabilities, e.g., a cloud server or a commodity physical server.

Fog Nodes: represent the workers of the stack. The only precondition is that every processing node should be capable of hosting linux-based operating system. We should note that we deployed the RAINBOW stack on light-weight ARM-based nodes like raspberries v4 with no performance issues.

IoT Sensors: are either connected on Fog nodes or can be requested from them (e.g., via APIs). RAINBOW Monitoring Agents (Sec.II-B) capture data from IoT sensors, along with utilization metrics from the Fog Node and the running containarized services. For this demo, we enrich the monitoring agents to reproduce IoT data streams from datasets files.

Networking: All nodes of the infrastructure need to be interconnected. For that reason, we consider that a mesh network exists in which all nodes can communicate to each other.

Stack Deployment & Configurations: We minimize the user's installation efforts and improve the system's interoperability by containerizing all of our services. So another precondition for both controller and Fog nodes is to have installed the Docker runtime environment. Moreover, we provide docker-compose YAML files for all services that users can execute them without any update. Our services are open-sourced, utilize open-source libraries, and their configurations are described via YAML files or as environmental parameters. More details about the installation and configuration of the project can be found in its documentation site[1].

[1] http://bit.ly/3sXTXjU

978-1-6654-9792-3/22/$31.00 ©2022 IEEE

Name	URL	Pluggable Schedulers	Declarative Queries	IoT Monitoring	Fog-oriented Optimizations	Performance Metrics Trade-offs
Apache Kafka Streams	kafka.apache.org		✓			
Apache Storm/Flink/Spark	{storm,flink,spark}.apache.org	✓(*Storm*)				
NebulaStream [1]	nebula.stream			✓	✓	
StreamSight [2]	bit.ly/3lWFpwR		✓		✓	
RAINBOW Analytics Stack	bit.ly/3sXTXjU	✓	✓	✓	✓	✓

TABLE I: Comparison among Streaming Analytics Engines

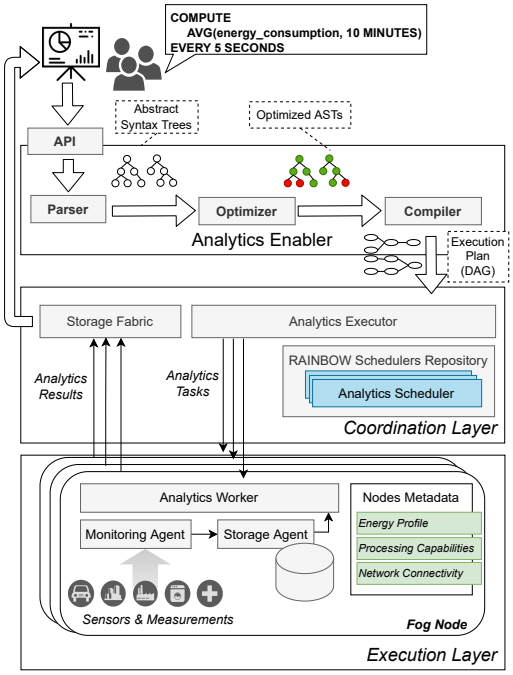

Fig. 1: RAINBOW Analytics Stack Overview

B. System Overview

Figure 1 depicts a high-level overview of the RAINBOW Analytics Stack. Starting from the Fog Nodes, the *Monitoring Agents* retrieve the monitoring data, including performance utilization metrics and IoT measurements, and disseminate them to the co-located *Storage Agents*, and, consequently, to the *RAINBOW Storage Fabric*. Specifically, Storage Fabric represents a logical component that abstracts and unifies the inter-connected Storage Agents by providing a decentralized API for access to monitoring data. To realize it, we utilize Apache Ignite[2], which is an open-source in-memory database. We created a lightweight wrapper on top of it and materialize state-of-the-art data sharing and replication algorithms [3]. It is important to mention that in a real deployment, each Storage Agent is capable of providing data locality information. Hence, monitoring data are immediately available without data needing to be moved to a central location. The last RAINBOW service that exists on a Fog Node is the *Analytic Worker*.

Furthermore, *Analytic Executor* creates and orchestrates a distributed processing layer that is composed by the Analytics Workers. When an artifact of streaming queries reaches the Analytic Executor, it coordinates the job deployment and facilitates the provisioning of the execution environment on the Analytic Workers. Since the Analytics Workers co-exist with Storage Agents, they are able to retrieve metrics directly from the local Storage Agent. RAINBOW's distributed processing module is built upon the Apache Storm[3], aiming not to implement yet another distributed data processing engine but rather to design and deploy novel scheduling algorithms that are decoupled from the underlying engine and acknowledge the unique settings found in the majority of Fog environments. With a provisioned execution environment in hand, the Analytic Executor invokes the respective *Analytic Scheduler* from the *RAINBOW Schedulers Repository*, which will provide near-to-optimal efficiency for analytic queries based on the user-desired optimization polices [4]. Currently, RAINBOW provides (i) a baseline scheduler that utilizes a round-robin task placement algorithm; (ii) a scheduler that takes into account the energy consumption of the underlying nodes and the processing performance; and (iii) a data quality-aware scheduler, which provides increased processing performance by sacrificing a portion of data quality [4].

Through the *RAINBOW Dashboard*, users are able to retrieve raw metrics from Storage Fabric or submit a set of IoT streaming declarative queries. For the latter, we utilize and extend the *query language* of the StreamSight Framework [2], which provides high-level declarative abstractions. The query model eases the definition of streaming analytic queries, whilst automatically minimizing the re-computations of the analytic tasks and unnecessary data transfer. Specifically, once a query is given by the user to the RAINBOW Analytics Enabler, the Parser will parse the query and translate it into an Abstract Syntax Tree (AST) representation. An AST expresses the language's grammar rules, and the final level of the tree contains the tokens and symbols of the query model. Through the validation of the ASTs, the Parser guarantees the correctness of the submitted queries. Then, the Optimizer will attempt to optimize the AST so a "better" logical execution plan can be derived. "Better" means that the query is optimized for Fog environments by extracting correlations among queries of the same job, pruning unnecessary computations and minimizing data shuffling to reduce communication latency. In turn, AST can be annotated with different scheduling policies. Then, the Compiler takes as input the optimized plans and generates the final executable artifact by recursively traversing them and "translating" each operator to the underlying engine code.

Then, the executable artifact is shipped to the Analytic Executor. With information about resource availability and storage metadata, and the RAINBOW-enabled Analytic Scheduler, the Analytic Executor executes the job at runtime, updates the job scheduling by following the invoked scheduler, and stores the results back to Storage Fabric. Finally, the dashboard retrieves the results from Storage Fabric and displays them.

[2] http://ignite.apache.org

[3] http://storm.apache.org

insight_name = **COMPUTE** *composite_expression*
[**WHEN** *filter_expression*]
[**EVERY** *time_interval*]
[**WITH** *optimizations*]

Fig. 2: RAINBOW Query Modeling

C. Query Modeling

RAINBOW query model allows users to create insights, denoted as high-level queries composed from raw monitoring metric streams. In a nutshell, an insight is a new data stream that results from one (or more) processed stream(s). Query model operators introduce aggregations, compositions, and transformations on top of multiple metric streams. Figure 2 depicts the basic structure of an insight. The simplest insight may include only the *insight_name* followed by a `COMPUTE` statement. The `COMPUTE` statement requires a composite expression (e.g., an aggregation function on a stream). Furthermore, the model provides three optional primitives, namely, (i) `WHEN` primitive, that filters a stream by applying specific offers; (ii) `EVERY` primitive, that alternates a purely streaming execution to a (micro-)batch query evaluation; and (iii) `WITH` statement, in which users define optimizations provided by the RAINBOW stack, like sampling, prioritizing of the results, etc.

```
COMPUTE
  AVG(vehicle_delay,10 MINUTES) BY city_segment
EVERY 5 SECONDS
WITH SCHEDULER=<scheduler_name>
```

Insight 1: Representative RAINBOW Query

For instance, Insight 1 illustrates a representative example of a query, which computes the average vehicle delay per city segment for a 10-minute sliding window with new datapoints considered every 5 seconds, whilst being optimized by a specific scheduler. The selection of tailored optimization based on the scenario could be extremely beneficial for the user in Fog Computing analytics. Specifically, one may need different optimization strategies even between queries that are submitted together. As we described earlier, the RAINBOW processing layer offers various scheduling algorithms. To this end, for the selected algorithm definition, RAINBOW's query model offers the `SCHEDULER` primitive. The `SCHEDULER` could be selected deliberately for every insight to dictate at low-level the operator's placement on Fog nodes.

III. Demonstration

We will demonstrate the usability of RAINBOW Analytics Stack by showing insights from a real-world IoT application.
Application: Let us suppose that a public transportation operator develops an IoT application that analyzes on-time streaming data from its fleet. Towards that, the operator would like to deploy diverse queries, evaluate the generated results, and configure the system's performance. Vehicles continuously report their locations and other attributes (i.e., environment conditions) to perform location-based or overall summarized analytic tasks (i.e., vehicle delay reporting).
Deployment & Results: We will use workload generators to reproduce the dataset and we will submit a set of representative and interesting queries. During the demonstration, the audience will have the opportunity to be familiarized with the RAINBOW configurations and query language, investigate the query editor and system's dashboard, and see the performance and energy consumption differences among RAINBOW's placement strategies. For instance, Figure 3 illustrates the performance and energy consumption difference of the RAINBOW Analytics Stack deployment on a heterogeneous micro cluster (four RPi v4 with 4cores@1.4Ghz/4Gb and a commodity service with 12cores@2.4Ghz/12Gb). The baseline is the default round-robin scheduling approach of Apache Storm engine, while the resource and energy strategies introduce the resource-aware and energy-aware RAINBOW schedulers, respectively. Briefly, we can identify that with resource-aware scheduler we achieved 24ms reduction during the execution but with 12% more energy consumption, while the energy-aware scheduler had 18% less energy consumption by sacrificing about 20ms of latency.

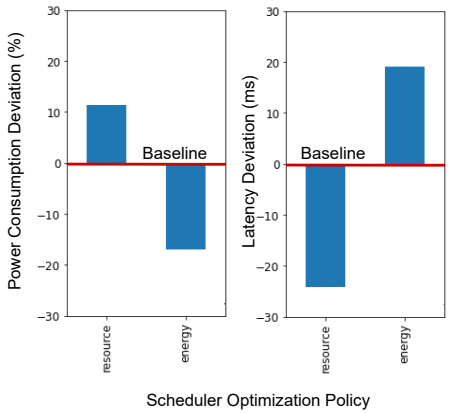

Fig. 3: Energy and Latency of two RAINBOW Schedulers in Comparison to the Baseline Apache Storm Scheduler

Testbed & Reproducibility: To create our testbed, we utilize a fog computing emulator [5]. It gives us the ability to rapidly deploy our stack on a laptop or on a cloud cluster, capture metrics, and apply energy consumption models. For the sake of the presentation, we will execute the emulation on a laptop to avoid the need of extra hardware. Nonetheless, attendees are encouraged to try the demo offline on both emulated and real devices (i.e., Raspberry Pi's). All configurations from both emulator and system will be publicly available[1].

Acknowledgement. This work is partially supported by the EU Commission through RAINBOW 871403 (ICT-15-2019-2020) project, and from RAIS (Real-time analytics for the Internet of Sports), Marie Skłodowska-Curie ITN, under grant agreement No 813162.

References

[1] S. Zeuch, E. T. Zacharatou, S. Zhang, X. Chatziliadis, A. Chaudhary, B. Del Monte, D. Giouroukis, P. M. Grulich, A. Ziehn, and V. Mark, "Nebulastream: Complex analytics beyond the cloud," *VLIoT*, 2020.
[2] Z. Georgiou, M. Symeonides, D. Trihinas, G. Pallis, and M. Dikaiakos, "StreamSight: A Query-Driven Framework for Streaming Analytics in Edge Computing," in *Proceedings of the 11th IEEE/ACM UCC*, 2018.
[3] T. Toliopoulos, A.-V. Michailidou, and A. Gounaris, "Data placement in dynamic fog ecosystems," in *38th IEEE ICDE Workshops*, 2022.
[4] A.-V. Michailidou, A. Gounaris, M. Symeonides, and D. Trihinas, "Equality: Quality-aware intensive analytics on the edge," *Inf. Syst.*, 2022.
[5] M. Symeonides, Z. Georgiou, D. Trihinas, G. Pallis, and M. D. Dikaiakos, "Fogify: A fog computing emulation framework," in *IEEE/ACM SEC2020*, 2020.