



Project Title AN OPEN, TRUSTED FOG COMPUTING PLATFORM FACILITATING THE DEPLOYMENT, ORCHESTRATION AND MANAGEMENT OF SCALABLE, HETEROGENEOUS AND SECURE IOT SERVICES AND CROSS-CLOUD APPS

Project Acronym RAINBOW

Grant Agreement No 871403

Instrument Research and Innovation action

Call / Topic H2020-ICT-2019-2020 / Cloud Computing

Start Date of Project 01/01/2020

Duration of Project 36 months

D4.1 – Data Management Services – Early Release

Work Package	WP4 – RAINBOW Data Management Services
Lead Author (Org)	Demetris Trihinas (UCY)
Contributing Author(s) (Org)	M. Symeonides, G. Pallis, M. D. Dikaiakos (UCY); A-V. Michailidou, T. Toliopoulos, G. Vlahavas, A. Gounaris (AUTH); S. Kousiouris, S. Venios (SUITE5)
Reviewers	T. Toliopoulos (AUTH), J. Kaldis (UNISYSTEMS), Thomas Pusztai (TUW)
Due Date	31.03.2021
Actual Submission	31.03.2021
Version	1.0

Dissemination Level

<input checked="" type="checkbox"/>	PU: Public (*on-line platform)
<input type="checkbox"/>	PP: Restricted to other programme participants (including the Commission)
<input type="checkbox"/>	RE: Restricted to a group specified by the consortium (including the Commission)
<input type="checkbox"/>	CO: Confidential, only for members of the consortium (including the Commission)



The work described in this document has been conducted within the project RAINBOW. This project has received funding from the European Union's Horizon 2020 (H2020) research and innovation programme under the Grant Agreement no 871403. This document does not represent the opinion of the European Union, and the European Union is not responsible for any use that might be made of such content.



Versioning and contribution history

Version	Date	Author	Notes
0.0	17.02.2021	Demetris Trihinas (UCY)	Deliverable structure and content placeholders
0.1	24.02.2021	Demetris Trihinas (UCY)	Introduction
0.2	04.03.2021	Demetris Trihinas (UCY), Moysis Symeonides (UCY)	SOTA for geo-distributed data processing and fog analytics
0.3	08.03.2021	Theodoros Toliopoulos (AUTH)	SOTA, user roles, requirements and interactions for distributed data storage
0.4	12.03.2021	Moysis Symeonides (UCY), Sotiris Kousiouris (SUITE5), Stefanos Venios (SUITE5)	User roles, requirements and interactions for distributed data processing and fog analytics
0.5	15.03.2021	Anna-Valentini Michailidou (AUTH), George Vlahavas (AUTH), Theodoros Toliopoulos (AUTH), Anastasios Gounaris (AUTH)	Distributed data storage architecture and implementation
0.6	18.03.2021	Demetris Trihinas (UCY), Moysis Symeonides (UCY), Stefanos Venios (SUITE5)	Distributed data processing and fog analytics service architecture and implementation
0.7	20.03.2021	Demetris Trihinas (UCY), George Pallis (UCY), M. D. Dikaiakos (UCY)	Conclusions, reference and release of initial document draft
0.8	23.03.2021	Demetris Trihinas (UCY), Moysis Symeonides (UCY), Theodoros Toliopoulos (AUTH), Stefanos Venios (SUITE5)	Document finalized for internal review
1.0	31.03.2021	Demetris Trihinas (UCY)	Document finalized and ready for submission

Disclaimer

This document contains material and information that is proprietary and confidential to the RAINBOW Consortium and may not be copied, reproduced or modified in whole or in part for any purpose without the prior written consent of the RAINBOW Consortium

Despite the material and information contained in this document is considered to be precise and accurate, neither the Project Coordinator, nor any partner of the RAINBOW Consortium nor any individual acting on behalf of any of the partners of the RAINBOW Consortium make any warranty or representation whatsoever, express or implied, with respect to the use of the material, information, method or process disclosed in this document, including merchantability and fitness for a particular purpose or that such use does not infringe or interfere with privately owned rights.

In addition, neither the Project Coordinator, nor any partner of the RAINBOW Consortium nor any individual acting on behalf of any of the partners of the RAINBOW Consortium shall be liable for any direct, indirect or consequential loss, damage, claim or expense arising out of or in connection with any information, material, advice, inaccuracy or omission contained in this document.



Table of Contents

Executive Summary	6
1 Introduction	8
1.1 Document Purpose and Scope	10
1.2 Document Relationship with other Work Packages	11
1.3 Document Structure	12
2 State of the Art and Key Technology Axes Challenges	13
2.1 Geo-Distributed Data Storage and Sharing	13
2.2 Geo-Distributed Data Processing	15
2.3 Fog Service Analytics and Query Models	16
3 Distributed Data Storage and Sharing Service	18
3.1 Requirements and Exposed Functionality	18
3.1.1 Functional Requirements	18
3.1.2 Non-Functional Requirements	21
3.2 Reference Architecture and Implementation	22
3.2.1 Apache Ignite	24
3.2.2 DBMS Comparison	25
3.2.3 DBMS Benchmarks	26
3.2.4 Server and Client Instances	30
3.2.5 Components	31
3.3 Interaction with other RAINBOW Services and Components	32
3.4 API and Documentation	33
4 Distributed Data Processing Service	36
4.1 Requirements and Exposed Functionality	36
4.1.1 Functional Requirements	37
4.1.2 Non-Functional Requirements	40
4.2 Reference Architecture and Implementation	41
4.2.1 High-Level Logical Overview of Analytics Workflow	41
4.2.2 Apache Storm	43
4.2.3 Apache Storm in the RAINBOW Analytics Ecosystem	44
4.2.4 Analytics Job Scheduling in Fog Realms	47
4.3 Interaction with other RAINBOW Services and Components	50
4.4 API and Documentation	51
5 Fog Analytics Service	53
5.1 Requirements and Exposed Functionality	53
5.1.1 Functional Requirements	54
5.1.2 Non-Functional Requirements	57
5.2 Reference Architecture and Implementation	57
5.2.1 Query Model Expressivity	60



5.2.2	RAINBOW-Enabled Optimizations.....	62
5.2.3	Analytics Job Compilation Process	65
5.3	Interaction with other RAINBOW Services and Components	69
5.4	API and Documentation	69
6	<i>Conclusion</i>	70
7	<i>References</i>	72
Appendix	75
	EBNF Descriptive Query Model	75



List of tables

Table 1: Scientific Papers Published within WP4 Scope.....	11
Table 2: Distributed Data Storage and Sharing and interacting user groups.....	18
Table 3: System-wide RAINBOW function requirements relevant to Distributed Data Storage and Sharing.....	19
Table 4: Distributed in-memory database qualitative comparison.....	25
Table 5: RAINBOW Distributed Data Processing service and interacting user groups....	36
Table 6: System-wide RAINBOW function requirements relevant to Distributed Data Processing service.....	37
Table 7: Current status of RAINBOW-enabled analytics job scheduling algorithms.....	50
Table 8: Distributed Data Processing Service REST API.....	51
Table 9: Fog Analytics service and interacting user groups.....	53
Table 10: System-wide RAINBOW function requirements relevant to Fog Analytics.....	54
Table 11: Window-based model operators.....	61
Table 12: Accumulative-based model operators.....	61

List of figures

Figure 1: The RAINBOW Architecture with Data Management Services Highlighted.....	9
Figure 2: High-level overview of instances and components of the Distributed Data Storage and Sharing service.....	23
Figure 3: Ignite and Redis comparison for workload A of YCSB.....	28
Figure 4: Ignite and Redis comparison for workload B of YCSB.....	29
Figure 5: Local data caches schemas.....	30
Figure 6: Logical Overview of the Distributed Data Processing service in the RAINBOW ecosystem.....	42
Figure 7: Storm in the RAINBOW Ecosystem.....	45
Figure 8: Storm topology.....	46
Figure 9: High-Level Overview of the Fog Analytics Cycle.....	58
Figure 10: Insight Abstract Syntax.....	60
Figure 11: Exemplary Abstract Syntax Tree.....	66
Figure 12: Exemplary query adopting the RAINBOW query model vs the native Storm programming model.....	67
Figure 13: Queries reusing intermediate results to reduce unnecessary data computations.....	69



Executive Summary

The aim this Deliverable is to provide a comprehensive overview and documentation report for the early release of the RAINBOW Data Management Services which are designed and developed within the scope of Work Package 4 (WP4). The purpose of WP4 is to provide the RAINBOW ecosystem with intelligent data management services, such as data storage and sharing mechanisms (T4.1), which can be deployed alongside the fog continuum so that analytic insights are extracted from fog services via geo-distributed data processing (T4.2) with the use of high-level analytic query abstractions (T4.3).

This Deliverable begins by presenting the challenges introduced in reference to data management in geo-distributed deployments, such as in the case of fog computing realms. Next, it continues with the identification of the requirements and exposed functionality for the components comprising the RAINBOW Data Management Services. From the identified set of requirements, reference architecture and public APIs, the readers obtain a comprehensive overview of the early implementation of the Data Management Services, which constitute key aspects for both the RAINBOW Mesh and Orchestration stack, and are key components comprising the first prototype release of the RAINBOW Platform.

Finally, the Deliverable concludes and outlines the work to be conducted towards introducing D4.2 that will assess the accomplishment of the requirements, features and toolsets introduced in this deliverable and will provide the final documentation report of the RAINBOW Data Management Services.



Table of Abbreviations

WP	Work Package
IoT	Internet of Things
EBNF	Extended Backus–Naur Form
DB	Database
DBMS	DataBase Management Service
SSL	Secure Sockets Layer
TLS	Transport Layer Security
VPN	Virtual Private Network
ACID	Atomicity, Consistency, Isolation and Durability (refers to DB transactions)
AST	Abstract Syntax Tree
JSON	JavaScript Object Notation
SQL	Structured Query Language
KPI	Key Performance Indicator
OSN	Online Social Network
DAG	Directed Acyclic Graph



1 Introduction

The broad vision of the RAINBOW project is to empower IoT service operators to solely focus on the design and development of their services business logic, leaving to RAINBOW the burden of *how* and *where* services must be placed (in the fog continuum), establishing secure collaboration among entities and dealing with low-level aspects in data analysis including heterogeneous resource management, mobility and data movement.

To this end, Deliverable D4.1, henceforth simply referred to as D4.1, provides a comprehensive overview and documentation report for the early release of the RAINBOW Data Management Services. In particular, these services -developed within the scope of Work Package 4 (WP4)- contribute to the enablement of interoperable and location-aware data processing across the fog continuum. This is achieved by pushing “intelligence” to the network “edge” with -in place- data management and fog service analytics through decentralized edge APIs capable of “talking” to each other without the need of offline, manual or human intervention.

To this end, the RAINBOW Data Management Services include three vital software components:

- **The Distributed Data Storage and Sharing Service:** The role of this service is to provide persistent and in-memory data storage capabilities to nodes scattered across the fog continuum to ensure in-time access to recently collected monitoring data by collaborating entities (i.e., analytics, orchestration, routing). To achieve this, a high-performance indexing scheme is maintained across the fog continuum so that locally stored recent and historic monitoring data, are accessed with low-latency (stable algorithmic complexity). With monitoring data, we refer to both fog node utilization and any application-level metrics that the user wishes to store across the fog topologies for subsequent use (e.g., compute analytic insights). In turn, the data are replicated and partitioned across nodes to ensure user-desired data quality constraints. In turn, both a push and pull-based API is provided for monitoring data delivery.
- **The Distributed Data Processing Service:** The role of this service is to enable (geo-) distributed data processing, with the use of open and popular big data engines, to extract analytic insights from deployed fog services. To achieve this, novel algorithms for scheduling analytic jobs over the secure overlay mesh network are utilized. These algorithms are specifically tailored to support fog-aware critical end-user requirements, including performance indicators, (e.g., latency and throughput), data quality, energy consumption and cost constraints,

along with optimization policies for coping with network uncertainties that are highly evident in the fog continuum.

- **The Fog Analytics Service:** The role of this service is to ease the description and programmability of complex analytic jobs, by providing a high-level and declarative query model that supports the abstraction of analytics from real-time monitoring data, along with the declaration of end-user fog-aware requirements (e.g., optimize the analytics job for performance while within a certain cost budget). The query model is completely decoupled from the underlying distributed processing engine to promote the reuse of analytics jobs. In turn, the compilation of analytic jobs provides some initial optimizations that attempt to reduce the unnecessary computation (and distribution over the network) of intermediate query results that are a significant overhead in geo-distributed environments.

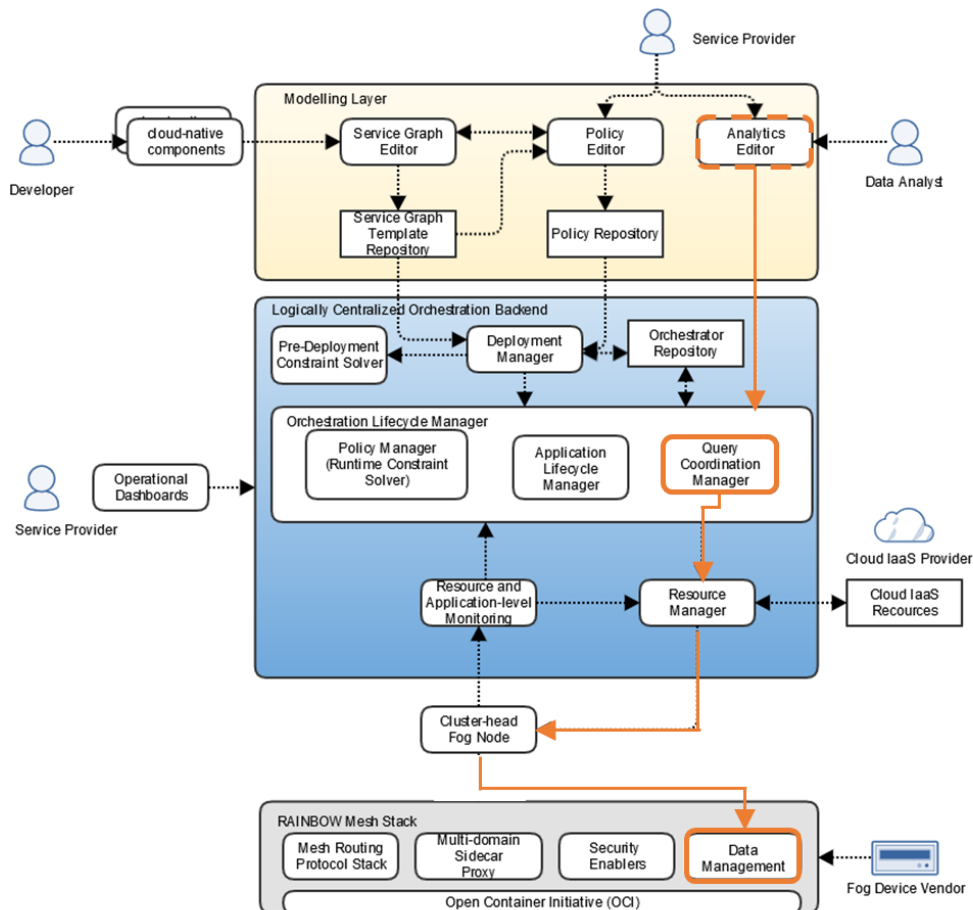


Figure 1: The RAINBOW Architecture with Data Management Services Highlighted

Figure 1 depicts a high-level overview of the RAINBOW Platform, highlighting the key software components of the Data Management Services. The *Query Coordination*



*Manager*¹, as the component name reveals, is part of the Orchestration Layer and is in charge of the runtime coordination of the Data Management Services. In particular, upon application deployment, this component handles both the initial configuration and runtime coordination of the Data Management Services, which are part of the RAINBOW Mesh Stack. These include the configuration of the two RAINBOW services deployed on the fog nodes, namely, the *Storage Agent* and the *Analytics Worker*. Initial configuration refers to the parameterization of the services, including general service parameters such as network binding(s) and domain-specific parameters such as the data eviction period for the cache managed by the *Storage Agent* and the maximum capacities of the resources reserved for the *Analytics Workers* (e.g., threads, memory, etc). In turn, runtime coordination includes the deployment of continuous analytic jobs over the fabric inter-connecting the *Analytics Workers* for *Distributed Data Processing* and deployed across the application's allocated fog nodes. Requests for new analytic jobs are received from the *Analytics Editor*, which is found in the Analytics Perspective of the RAINBOW Dashboard and part of the RAINBOW Modelling Layer², while updated insights and plotted data can be graphically viewed via the *Analytics Runtime Facet* of the Analytics Perspective.

Finally, it should be mentioned that in a deployment, each *Storage Agent* while completely capable of providing monitoring data at the local fog node level, Agents among collaborating fog nodes are also seamlessly inter-connected establishing a *Storage Fabric*. This, represents a logical sub-component that abstracts and unifies the functionality offered by inter-connected *Storage Agents*, providing a decentralized API for access to monitoring data. Hence, monitoring data are immediately made available (e.g., for distributed data processing) through the RAINBOW secure overlay mesh network without data needed to be moved to a central (cloud) location that will provide data access but with both a performance penalty and costs incurred for data movement.

1.1 Document Purpose and Scope

The purpose of this deliverable is to provide a comprehensive overview and documentation report of the early release of the RAINBOW Data Management Services which contribute to providing interoperable analytic capabilities to fog-enabled deployments via intelligent data storage and processing mechanisms capable of operating in the fog continuum and on top of trusted overlay mesh networks. In respect to this, D4.1 aims to derive a clear overview of the early design and development of the three components comprising the RAINBOW Data Management Services and are developed under the umbrella of WP4, namely: (i) the Distributed Data Storage and

¹ Also referred to as the Analytics Enabler.

² This component, although tightly coupled with fog analytics, is highlighted with a dashed lining as it is part of the RAINBOW Dashboard developed within the scope of WP5.



Sharing Service; (ii) the Distributed Data Processing Service; and (iii) the Fog Analytics Service. To this end, D4.1 documents for each component of the RAINBOW Data Management layer, the requirements that must be satisfied to overcome the challenges introduced when deploying data storage and analytics services in the fog continuum, their functionalities, how they operate and the first version of their exposed API which is used to interact with other RAINBOW components, users and/or third-party services.

Finally, we note that parts of D4.1 are based on a number of scientific papers [1]–[5], which introduce core concepts of the components part of the RAINBOW Data Management Services and WP4. These papers, all developed within the first reporting period, are highlighted below:

Table 1: Scientific Papers Published within WP4 Scope

WP4 Scientific Papers	RAINBOW Partners
PROUD: PaRallel OUTlier Detection for Streams. T. Toliopoulos, C. Bellas, A. Gounaris, and A. Papadopoulos. In Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20). Association for Computing Machinery, New York, NY, USA, 2717–2720.	AUTH
Fogify: A Fog Computing Emulation Framework. M. Symeonides, Z. Georgiou, D. Trihinas, G. Pallis and M. D. Dikaiakos, 2020 IEEE/ACM Symposium on Edge Computing (SEC) , San Jose, CA, USA, 2020, pp. 42-54.	UCY
[Best Demo Award] Emulating Geo-Distributed Fog Services. M. Symeonides, Z. Georgiou, D. Trihinas, G. Pallis and M. D. Dikaiakos, 2020 IEEE/ACM Symposium on Edge Computing (SEC) , San Jose, CA, USA, 2020, pp. 187-189.	UCY
A Self-stabilizing Control Plane for Fog Ecosystems. Z. Georgiou, C. Georgiou, G. Pallis, E. M. Schiller, and D. Trihinas, In 2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC) , pp. 13-22, December 2020.	UCY

1.2 Document Relationship with other Work Packages

This deliverable is built on the foundation of D1.1 and D1.2, which provide a concrete documentation of the RAINBOW ecosystem requirements along with the current version of the reference architecture, including the key technologies and features supported by RAINBOW. To this end, D4.1 extends the RAINBOW documentation by providing a comprehensive report for the RAINBOW Data Management Services. What is more, D4.1



serves as a guide for D4.2, the RAINBOW Data Management Services - Final Release, which will assess the accomplishment of the requirements, features and toolsets introduced in this deliverable and will provide the final documentation of the RAINBOW Data Management Services.

1.3 Document Structure

The rest of this deliverable is structured as follows: Section 2, provides an updated guide of the comprehensive report introduced in D1.2 and referring to the State-of-the-Art landscape in data management for containerized applications deployed across geo-distributed realms. Section 3, 4 and 5, present a comprehensive documentation report introducing the reference architecture, exposed functionality and implementation details referring to the software components of the RAINBOW Data Management Services. Section 6 concludes this Deliverable and outlines the work to be conducted towards introducing D4.2. In the Appendix, a comprehensive overview of the fog analytics descriptive query model is provided in a normative format (EBNF).



2 State of the Art and Key Technology Axes Challenges

In this Section, we will update the State-of-the-Art presented in D1.2. Particularly, we present the challenges introduced in reference to storing and processing data to derive high-level analytic insights in fog-enabled geo-distributed execution environments.

2.1 Geo-Distributed Data Storage and Sharing

Edge computing frameworks, like RAINBOW, are highly distributed as tens or even hundreds of devices are placed at multiple locations. These devices need to access persistently stored data, modify and save them to a database. Moreover, real-time analysis on-the-fly is crucial when dealing with Edge Computing applications and streaming data, like for example the analysis of sensor data. When choosing a Database Management System for such a scenario new challenges arise, compared to a centralized solution.

Scalability, as previously mentioned, is crucial in Edge Computing as we deal with tens or hundreds of devices, as said previously. High scalability also increases the elasticity of the DBMS and the ability to handle workload changes. Scalability in such scenarios is horizontal by means of adding more devices when demand for resources arises [6], [7]. Couchbase [8] provides Multi-Dimensional scaling that scales queries, indexes and data, supporting more than one hardware profile, resulting in isolation of services.

The reliability and fault-tolerance of a distributed DBMS is often achieved through data replication, i.e., copies of data that are stored in multiple devices [6], [9]. The master-slave model is used in multiple DBMSes, whereas multiple replication types like transactional, snapshot or merge and schemas like full or partial exist. Multi-master replication is preferable when dealing with multiple devices. Amazon Aurora [10], ArangoDB [11], CouchDB [12], PostgreSQL [13] and Redis [14] provide this feature. The Multi-master paradigm also increases the availability and response time of the DBMS. Also, distributed DBMSes have to handle more aspects regarding concurrency and recovery, when compared to a centralized DBMS. More specifically, data consistency is trickier due to the multiple copies of data and distributed commits. Riak [15] tackles this by allowing conflicting copies of data to exist at the same time while guaranteeing eventual consistency.

Moreover, DBMSes should also take into account the failure of links and devices. Apache Cassandra [16] features no single point of failure. A *Write Ahead Log* [17], [18] is also used to keep logs of transactions in case of device failure.



In order to optimize the use of multiple edge devices and reduce bottlenecks, i.e., overloading of certain devices, load balancing algorithms must be a part of the DBMS. This can be achieved by dynamically altering the placement of data to devices in order to off-load them. Slicer [19] is a service that partitions data using keys while monitoring the load of each key and making rebalancing moves. Accordion [7] keeps load balanced through scaling (adding or removing devices) and predicts bottlenecks based on transaction affinity. Other works [20], [21] achieve fine-grained partitioning of data by detecting and carefully placing the “hot” tuples.

The nature of data RAINBOW deals with is streaming and multiple data are produced per second. The fast analysis of them and real-time responses are crucial in Edge Computing scenarios. Saving and accessing data through the disk would cause a large non-acceptable overhead, thus the use of in-memory databases for analysis and distribution is the way forward. VoltDB [22] is a main memory DBMS based on H-Store providing elastic scalability, rapid failover and consistent low latency but distributed transactions are performed by a single thread. Redis [14] is an open-source, fast main-memory data structure store. Redis can eliminate delays in data retrieval achieving very fast response times with read and write operations taking less than a millisecond, while it also provides high availability, scalability, fast fault recovery, built-in replication and on-disk persistence. Hazelcast IMDG [23] is an open-source in-memory data grid. The main advantage of using data grids is speed, especially when dealing with vast streaming data. Data are evenly distributed to the cluster nodes providing horizontal scaling. Apache Ignite [18] is an open-source, distributed store designed to work with big data and clusters of nodes. The form in which data is being stored is in key-value pairs which can be replicated or partitioned across the nodes of the cluster, achieving scalability and fault-tolerance. Ignite also supports co-located processing enabling the analysis of data on nodes and achieving lower data transferring across the network making it suitable for data-intensive or compute-intensive analytics like RAINBOW use cases.

Finally, a distributed DBMS needs to take additional security measures due to the extensive number of users and devices that access the data. The security should be considered both in the communication, that is at the exchanging of data as well as in data by means of authentication and encryption. The first part can be achieved through SSL and TLS protocols and the use of VPN while the second through digital certificates. What DBMSes can provide to security is the encryption of data upon storing them using keys [24] as well as authentication and access control mechanisms [25], [26].



2.2 Geo-Distributed Data Processing

For more than the better parts of the last two decades, we are witnessing the exponential growth in both the volume of available data and the velocity at which data is generated. For many of today's diverse applications, the tools to process data at scale are frameworks adopting variants of the MapReduce/DataFlow programming paradigms that exploit various degrees of parallelism in the data flow [27]–[30]. In particular, frameworks such as Hadoop, Spark, Flink and Storm, decompose analytics jobs into stages, where each stage can then be further divided into independent tasks that are scheduled in parallel and executed on the worker nodes of a distributed computing cluster [31]. Therefore, any per-task performance gain can significantly improve the overall performance of the analytics job.

As finding the (near-) optimal placement for analytic tasks over distributed computing nodes is not a hard-enough problem itself, the geo-distribution of data analytics due to the prevalence of the Internet of Things (IoT) is exhausting the limits of the schedulers available in today's big data engines. Delay-sensitive IoT applications are now embracing fog/edge computing to process data in-proximity of the data origin to reduce any potential overheads of disseminating data back-and-forth to the cloud [32], [33]. However, data processing in fog ecosystems has its challenges [34]. In fog realms, resource heterogeneity is the norm, which contradicts with the operating requirements of data analytics frameworks that are optimized for homogeneous machine clusters found in the cloud [35]. In turn, the network capabilities of each worker can also significantly differ, as well as, the network distance from the other workers [36]. The latter has the potential to significantly impact the performance of an analytics job as slow running tasks on “distant” worker nodes creates bottlenecks on the overall job performance. Hence, data processing using big-data engines is dominated by the communication between map-reduce phases, where several replicas of identical data are transferred to the reduce phase for obtaining final outputs.

Capitalizing on the effect of network heterogeneity in geo-distributed environments, Pu et al. introduce Iridium [37]. Iridium is an analytics scheduler for Apache Spark that attempts to achieve low query response times by optimizing both task and data placement. The system uses a linear solver that considers both site bandwidths and query characteristics to solve the placement problem, while redistributing datasets among the sites before queries' arrivals. In turn, Tetrium [38] extends the Iridium scheduler, by utilizing multi-resource allocation in geo-distributed clusters and jointly considers both compute and network resources for task placement and job scheduling. Focusing on the placement of tasks, Flutter [39] deploys the operators closer to the data sources and optimizes individually every stage of a Map-Reduce graph. Similarly, WANalytics [40] takes as input arbitrary DAGs of computations and optimizes each node of the graph



individually. The processing takes place at edge DCs while a heuristic mechanism is applied for the data transfer reduction between them. In turn, Gaia [41] is a framework that maintains the performance of geo-distributed execution by applying an intelligent communication mechanism over bandwidth-constrained networks in an attempt to guarantee both accuracy and correctness during the execution.

Most of the aforementioned frameworks consider deployments on geo-distributed data centers without exploring inherent needs of IoT applications, such as streaming execution, and without the consideration that the analytics workers may reside on very constrained nodes. Towards this, T-storm [42] supports the application of query operators over streaming settings by considering the inter-node and inter-process traffic to assign workload to the nodes, rather than the baseline approach adopted by the Apache Storm engine. In turn, R-Storm [43] is a resource-aware scheduler for Apache Storm that efficiently allocates analytic tasks to the underlying resources, while the T3-Scheduler [44] attempts to place communicating tasks closer to each other. Even if the aforementioned frameworks efficiently schedule query pipelines with streaming operators, they do not consider specific Fog and Edge computing characteristics. For instance, Edge computing frameworks, like EdgeWise, consider more processing restrictions to improve latency and throughput [45]. EdgeWise is built on top of Apache Storm and assigns streaming operators to underlying workers. Yet, the system is limited because it does not consider any network implications.

2.3 Fog Service Analytics and Query Models

IoT services spanning across the Fog continuum, generate vast amounts of streaming data ranging from IoT device performance to IoT service behavior and up to user-relevant data. The aforementioned data streams fall within the class of big data since they are characterized by their variety, velocity, and volume (3Vs). Consequently, big data distributed processing engines are the mainstream approach for stream processing. These processing engines, as the name denotes, feature strict programming models for defining, even large pipelines of operations on the ingested data (e.g., transformations, aggregations and grouping) by adopting the map-reduce paradigm (e.g., Hadoop), dataflow programming models (e.g., Spark, Flink), and/or user-defined Directed Acyclic Graphs (DAGs) (e.g., Storm). The latter is considered a steep learning curve for users, e.g., Service Operators³, that are not programming experts and are not familiar with the specific programming concepts.

³ Adopting RAINBOW user role terminology.



To alleviate the implication of advanced knowledge of a programming model, operator abstractions have already been introduced for the most popular big data frameworks. For instance, Trident [46] is a framework for Apache Storm, that introduces pipeline operators applicable to ingested data streams in order to minimize the Storm's DAGs programming effort. Similarly, the Apache Spark ecosystem includes two packages with high-level query abstractions on top of the Spark engine, namely SparkSQL [47] and Structured Streaming [48]. The former provides a set of SQL-like operators on top of the Spark programming model, while the latter enriches SparkSQL with streaming capabilities. Even if these approaches are in the right direction, they are focused only on big data analytics, without providing edge- and fog-oriented operators and optimizations.

Domain-Specific Languages (DSLs) offer pre-defined abstractions to represent concepts from an application domain and DSL compilers are rather optimized for this specific domain. The unique characteristics of IoT processing and Edge Computing demand new operators to express different constraints and optimizations such as sampling, upper error-bounds, bounded resources, placement awareness, among others [34].

Recently, a handful of frameworks have been proposed to derive analytic insights for edge computing and network telemetry. For example, Edgent [49] is a framework providing micro-kernel run-times with small footprints that are particularly tailored to deriving streaming analytics on IoT gateways, network routers, and edge devices. Tailored to edge network telemetry analytics, Sonata [36] is a framework that offers scalable streaming processing. The framework provides a declarative pipeline-interface that allows network operators to express their analytic queries. Under the hood, Sonata uses the programmable data-plane of network switches for query preprocessing and Spark for query execution. However, Sonata is developed solely for packet-level network telemetry analytics without any acknowledgment for other types of data. In contrast to the aforementioned, StreamSight is a framework (developed by UCY) [50] for edge-enabled IoT services which provides rich and declarative query abstractions for expressing complex analytics over data streams and compiling these queries into stream processing jobs for distributed processing engines. StreamSight offers several query operators to derive high-level analytic insights, along with execution optimizations and constraints tailored for edge computing to achieve latency, robustness, and approximations in query execution.



3 Distributed Data Storage and Sharing Service

In this Section, we present a comprehensive documentation report introducing the reference architecture, exposed functionality and implementation details referring to the Distributed Data Storage and Sharing Service.

3.1 Requirements and Exposed Functionality

Based on the user groups documented in D1.1, the identified users interacting with the RAINBOW Data Storage and Sharing are presented in Table 2 and are the following:

Table 2: Distributed Data Storage and Sharing and interacting user groups

User Group	Interaction with Distributed Data Storage and Sharing Service
Service Operator/Owner	Interacts with the RAINBOW Data Storage and Sharing service by defining the time range for which historical data will be persistently stored (data extinction period). This is done by using the graphical tools for service description enrichment made available through the Service Graph Editor of the RAINBOW Dashboard.
Service Developer	Interacts with the RAINBOW Data Storage and Sharing service by defining the schema and storage properties for custom application data that will be stored in the service's database.
RAINBOW Developer	Interacts with the RAINBOW Data Storage and Sharing Service by developing custom schemas and data access endpoints to increase the usage and the reachability of the service.
Fog Infrastructure Provider	Interacts with the RAINBOW Data Storage and Sharing Service by accessing monitoring and routing data regarding fog offerings utilization.

3.1.1 Functional Requirements

The RAINBOW system requirements, documented in D1.1 referring to the Distributed Data Storage and Sharing service are the following:



Table 3: System-wide RAINBOW function requirements relevant to Distributed Data Storage and Sharing

Req. No.	Requirement
FR.25	Efficient data storage and placement based on restrictions
FR.26	Specify monitoring data for storage based on analytic queries and orchestration SLOs

To satisfy the system requirements documented in D1.1 while also adhering to the key technology axes of the Data Management layer presented in D1.2, the following functionality must be exposed by the Distributed Data Storage and Sharing Service.

ID	FR.DSS.1
Title	Storage for real-time and historical monitoring data
Description	The Distributed Data Storage and Sharing service must provide the means to store both real-time and historical monitoring data using efficient data structures for fast read/write query operations.
Exposed Functionality	The Distributed Data Storage and Sharing service is able to efficiently store data using both key-value and sql-like tables. Real-time data are stored in-memory for faster query execution by using a key-value cache. The real-time data are also persistently stored in sql-like tables along with the rest of the historical data using specified indices making execution of queries faster. To this end, a prototype adopting (and extending) the open-source and popular Apache Ignite has been developed.

ID	FR.DSS.2
Title	Historical data eviction based on user-desired eviction policies
Description	The Data Storage and Sharing service must provide the means to control the time range of the historical data (data eviction period). This in turn limits the volume of the persistent data and the memory resources needed by the service so that even fog nodes with limited storage capabilities can be supported.
Exposed Functionality	The service is able to evict older historical data using their creation timestamp. The time range is a user specified parameter which is provided through the Service Graph Editor. Afterwards any data row whose lifetime exceeds the time range variable is automatically considered expired from the storage. Finally, the expired entries are removed. To this end, an initial management API has been implemented to control the storage nodes accordingly.



ID	FR.DSS.3
Title	Efficient partitioning and replication algorithms
Description	The Data Storage and Sharing service must provide the means to efficiently utilize the underlying fog resources by partitioning and/or replicating stored data when needed.
Exposed Functionality	The service is able to efficiently partition or replicate stored data based on the resource congestion and frequency of read requests. This is accomplished by taking into account many system, database and network metrics in order to decide the correct data placement that will lead to less data movement and more efficient resource allocation. In addition, the algorithms will also monitor fragile nodes and fully replicate the stored data in order to prevent loss of information. At the time of writing this deliverable, the deployment allows for custom data placement and partitioning, while query-aware analysis methods are currently under design.

ID	FR.DSS.4
Title	Secure data access
Description	The Data Storage and Sharing service must provide the means to access the storage. The access can either be used for writing or querying data. Additionally, the data should be available to the requesting service or component from any instance of the distributed data storage.
Exposed Functionality	The Distributed Data Storage and Sharing service is able to allow access to requesting services and components through an API layer on top of every distributed data storage instance (the Storage Fabric). The layer can be used for both writing data and requesting either the latest (real-time) or the historical stored data. This avoids the need of having to query every single instance for data when an aggregation must be performed.

ID	FR.DSS.5
Title	In-memory cache for routing tables
Description	The Data Storage and Sharing service must provide the means to temporarily cache the routing tables for the secure CJDNS overlay network protocol.
Exposed Functionality	The Distributed Data Storage and Sharing service is able to temporarily store in-memory the routing tables of the overlay network. Each node stores its own routing table in a key-value cache which can be queried by other services and components using the <i>Storage Fabric</i> API.



ID	FR.DSS.6
Title	Custom schema support for app-specific data storage
Description	The Data Storage and Sharing service must provide the option to temporarily store application-specific data from the application instances that run on the control plane.
Exposed Functionality	The Distributed Data Storage and Sharing service is able to allow applications running on the nodes to provide custom schemas (data models) and store data that are compliant with the specified schema. The data are stored in a distributed in-memory cache. The application-specific data are available for queries through an API layer running on top of the instances. The use of this API is optional.

ID	FR.DSS.7
Title	Deployment in geo-distributed realms
Description	The Data Storage and Sharing service must be able to function properly in a geo-distributed environment with heterogeneous networking and physical machines.
Exposed Functionality	The service is able to adapt on the movement of fog nodes either when a new one enters the network or when one leaves it. Also, the service takes into account the heterogeneity of the network links between the nodes in order to prevent loss of data. On both cases the functionality of the Data Storage and Sharing service is not compromised.

3.1.2 Non-Functional Requirements

ID	NFR.DSS.1
Title	ACID compliance
Description	The Data Storage service must comply with the ACID properties for database transactions, even if the data are stored in-memory for fast data access. Towards this, the service must guarantee for every transaction the data validity, through novel data replication and sharding mechanisms, despite possible errors due to the dynamic nature of the fog environment. Thus, the Data Storage service should not lose data over power failures, node movement and resource congestion.

ID	NFR.DM.2
Title	Data Volume
Description	The Data Storage service must be able to store an unbounded volume of data without errors while adhering to both the user-desired data eviction parameterization and the fog node's physical storage capabilities/limitations. Hence, the volume of the monitoring and the user-specified metrics can change at any time without impacting the functionality of the storage.



ID	NFR.DSS.3
Title	Non-Intrusiveness
Description	The Data Storage service must not interfere with the system or the functionality of any application(s). The stored data and the transactions should not consume excessive resources from either the underlying fog offerings or the containerized execution environment.

ID	NFR.DSS.4
Title	Authorized Data Access
Description	The Data Storage service must be secure when a service or component tries to access data. Each RAINBOW or third-party service should only be able to access the respective data through authentication. Thus, the Data Storage service should be able to handle requests from authenticated users/services.

3.2 Reference Architecture and Implementation

The Distributed Data Storage and Sharing service is responsible for the RAINBOW ecosystem needs for resilient data storage in a dynamic fog environment. To this end, the service provides a distributed solution comprised of instances communicating with each other in a peer-to-peer fashion departing from the leader-worker paradigm; still, the system behaves -in the eyes of its users- as a single coherent cluster. The various components that are provided make the data exchange with external components and users easy to execute with full ACID compliance during database operations.

The service builds upon a complete distributed database management system and more specifically Apache Ignite⁴ with the implementation of different instances and components fully supporting the RAINBOW requirements.

For the Apache Ignite DBMS, there are two distinct instance types that are available and applicable to the RAINBOW ecosystem settings:

- **Server instances:** this type of instance is responsible for data ingestion and storage. The instance is part of the RAINBOW mesh stack and is located on every fog node where the stack is deployed. Its main job is to store the node's monitoring data either locally or remotely based on the data placement algorithms. It also takes part in the ingestion and extraction tasks of user-application data as well as the caching of the network routing tables in a distributed data store.

⁴ <https://ignite.apache.org/>

- **Client instance(s):** This type of instance is responsible for data exchange and placement. The instance is deployed either on resource-rich fog nodes or cloud nodes and its main job is to re-distribute the stored data on the Server instances based on resource congestion, networking issues and query execution variables. Finally, the client instance is also responsible for mass data extraction using minor preprocessing functions.

Figure 2 presents an abstract overview of the instances and components interaction for the Data Storage and Sharing service.

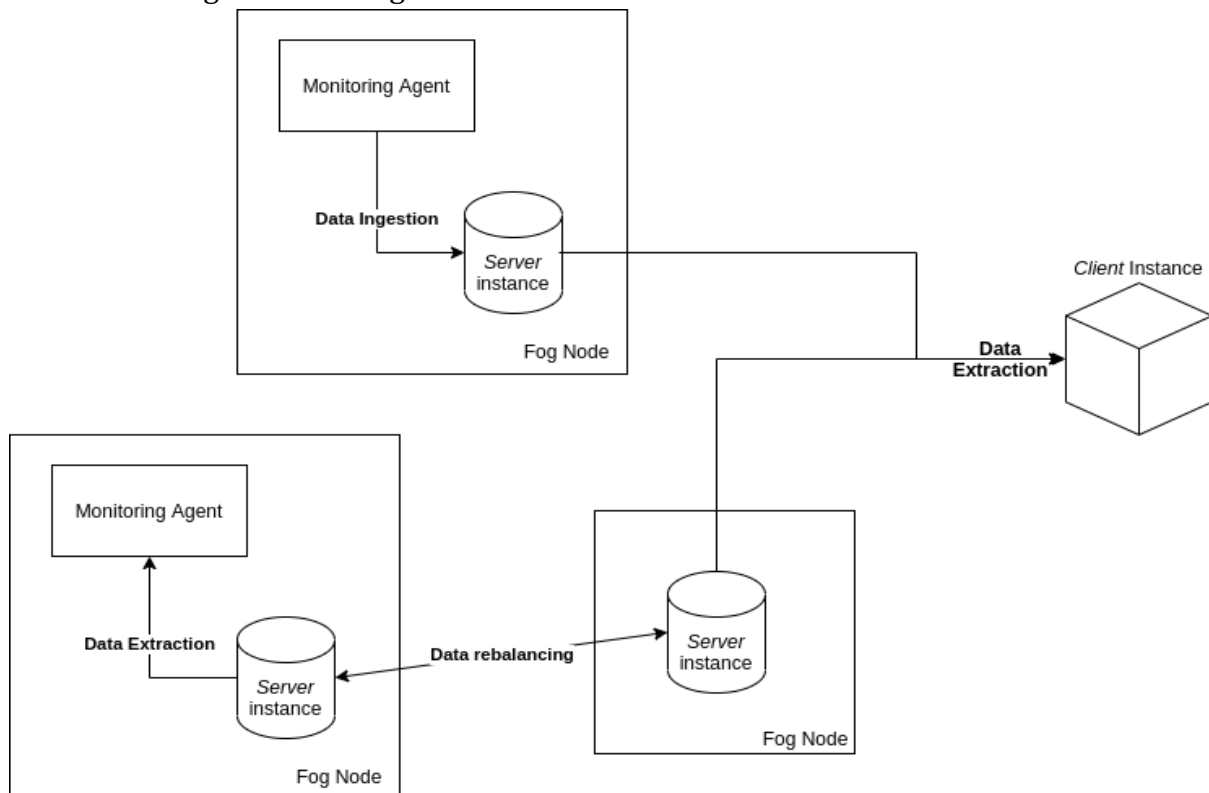


Figure 2: High-level overview of instances and components of the Distributed Data Storage and Sharing service

The main components (and functionality) of the RAINBOW Distributed Data Storage and Sharing service, are the following:

- **Data ingestion:** This component is part of the *Server* instances and is responsible for the storage of incoming (monitoring) data. Data are stored both in-memory and persistently up to a user-specified time period which constitute the latest and historical data respectively. The data are stored locally unless the placement has changed due to the *Client* instance re-distribution algorithms. Also, the ingestion component is responsible for storing any user-application data in the partitioned in-memory store.



- **Data extraction:** This component is available for both instances with different extraction mechanisms. Its main job is to make the stored data available to authorized services and users. On the *Server* instances, the component extracts the locally stored data from both the in-memory cache and the persistent storage. On the *Client* instances, the component can extract data from every available *Server* instance depending on the incoming query. Finally, the *Client* extraction can also provide the requesting services with results stemming from quick processing of the stored data, e.g. the average CPU load for every fog node. The Distributed Data Processing service (introduced in Chapter 4) uses this component to obtain access to real-time monitoring data.
- **Data rebalancing:** This component is responsible for the placement and re-distribution of the stored monitoring data. The geo-aware algorithms continuously re-assess the status of the fog nodes and decide whether there is a need to partition and/or replicate the local data of a fog node. The algorithms take into consideration both the node's resource and network utilization and the analytics agent placement to decide the source and the target nodes for the data movement.

3.2.1 Apache Ignite

The basis of the Distributed Data Storage and Sharing service is the database management system itself. To this end, Apache Ignite was chosen as the distributed database that can be used as a main-memory database and supports different persistent storage options. Ignite provides different ways to store and access data.

Ignite, as previously mentioned, has 3 different types of instances, *Server*, *Client* and *Thin Client*. Server instances are mainly responsible for storing data. Client instances can access stored data and either perform computations or query the data. Thin clients are smaller versions of the Client type that attach to a specific node for minor computations and querying.

Clustering in Ignite uses a decentralized approach without the support of the master/slave paradigm. A node can enter or leave a cluster without the permission of another node. Ignite provides different methods to help each node recognize the rest of the nodes in the cluster and notify everyone when a new one enters the cluster. The number of nodes can go up to thousands preserving linear performance.

Ignite provides two ways to store data with caches. The first one is the Key-Value paradigm whilst the second one can use specified custom schemas on the caches for SQL-like operations. Also, each cache has 3 different modes. The first one is the LOCAL mode in which the data are stored locally and can only be accessed by the specific Server



instance that stored them or by a Thin Client attached to that Server. The second mode is the PARTITIONED cache which partitions the data to the different server nodes based on some criteria. The criteria can be custom-made. The final mode is the REPLICATED cache which replicates every data point to a pre-specified number of nodes.

Each instance can run custom-made services depending on the needs of the user. The deployment can be achieved either by using custom instance filters or by providing deployed instances for each service. An instance can access the service even when deployed on a different (fog) node.

3.2.2 DBMS Comparison

To further elaborate on the choice of Apache Ignite an extended investigation of the different distributed in-memory DBMSes was completed. The results of the investigation are briefly presented in Table 4: Distributed in-memory database qualitative comparison where the top three (to date) DBMS solutions are compared based on the properties that involve the RAINBOW Distributed Data Storage and Sharing service's requirements.

Table 4: Distributed in-memory database qualitative comparison

	Redis	Hazelcast IMDG	Apache Ignite
ACID compliance	Not always guaranteed	Not always guaranteed	Full support for distributed ACID transactions
Cluster model	Leader-worker	All nodes are equal	All nodes are equal
Persistence	Yes	Yes	Yes (optional)
Consistency	Not guaranteed	Eventually consistent	Yes (if persistence is enabled)
Data sharing	Only replicated and partitioned data	Only replicated and partitioned data	Supports replicated, partitioned and local data

As the table presents, Apache Ignite is the database that presents the most properties in alignment with the service's requirements. It is the only system that is fully compliant with the ACID properties whilst the other two do not guarantee this type of transactions. This means that Ignite is the only database that can guarantee data validity despite possible failures that may be caused by the dynamic fog environment.



Persistency is available on all three databases, but guaranteed consistency is only present in Ignite as part of the ACID properties. Also, since every instance of the service needs to be able to communicate with each other, the leader-worker paradigm that Redis offers imposes restrictions that make the communication difficult.

Finally, Ignite is the only DBMS that supports local storage. This means that some or every instance can have a local in-memory and/or persistence cache with data available only to the instance itself. This also is part of the service requirements since the monitoring data should be stored locally on the fog node they are involved with. The other two databases only support partitioned caches when working in cluster mode.

From the aforementioned comparison Apache Ignite supports all requirements for the Data Storage and Sharing service and is the only choice for the database system.

3.2.3 DBMS Benchmarks

To provide additional information on the performance of Apache Ignite, we used the well-known YCSB benchmark suite [51], which is a standard framework to assist in the evaluation of different cloud-enable DBMS systems. The benchmarks compare Ignite against Redis.

YCSB includes a common set of workloads for evaluating the performance of different "key-value" and "cloud" serving stores. It supports benchmarking more than 40 database systems, including Cassandra, Couchbase, HBase, MongoDB, PostgreSQL, Redis, DynamoDB, Ignite and others. The YCSB project effectively comprises of two things:

- The YCSB Client, an extensible workload generator.
- The Core workloads, a set of workload scenarios to be executed by the generator.

The workloads in the core package are a variation of the same basic application type. Each operation against the data store is one of the following:

- Insert: Inserts a new record.
- Update: Updates a record by replacing the value of one field.
- Read: Reads a record, either one randomly chosen field or all fields.
- Scan: Scans records in order, starting at a randomly chosen record key. The number of records to scan is randomly chosen.

The following 6 standard workloads are defined in YCSB:



- **Workload A** (Update heavy workload): this workload resembles a session store recording recent actions. The read/update ratio is 50/50 and the default data size used is 1 KB records (10 fields, 100 bytes each, plus key).
- **Workload B** (Read mostly workload): this workload resembles a photo tagging application. Adding a tag is an update, but most operations are to read tags. The read/update ratio is 95/5, while the default data size is 1 KB records (10 fields, 100 bytes each, plus key).
- **Workload C** (Read only): this workload resembles a user profile cache, where profiles are constructed elsewhere (e.g., Hadoop). As such, the read/update ratio is 100/0. The default data size is 1 KB records (10 fields, 100 bytes each, plus key).
- **Workload D** (Read latest workload): this workload resembles user status updates in a database; people want to read the latest information that is stored. The read/update/insert ratio is 95/0/5, while the default data size is 1 KB records (10 fields, 100 bytes each, plus key). The insert order for this is hashed, not ordered. The "latest" items may be scattered around the keyspace if they are keyed by userid.timestamp. A workload which orders items purely by time, and demands the latest, is very different than the workload here (which is more typical of how people build systems).
- **Workload E** (Short ranges): this workload resembles threaded conversations, where each scan is for the posts in a given thread (assumed to be clustered by thread id). The scan/insert ratio is 95/5. The default data size is 1 KB records (10 fields, 100 bytes each, plus key). The insert order is hashed, not ordered. Although the scans are ordered, it does not necessarily follow that the data is inserted in order. For example, posts for thread 342 may not be inserted contiguously, but instead interspersed with posts from lots of other threads. The way the YCSB client works is that it will pick a start key, and then request a number of records; this works fine even for hashed insertion.
- **Workload F** (Read-modify-write workload): this workload resembles a user database, where user records are read and modified by the user or to record user activity. The read/read-modify-write ratio is 50/50. The default data size is 1 KB records (10 fields, 100 bytes each, plus key).

Each workload is comprised of a Load phase, where data are inserted into the database and a Run phase, where data are read, updated, deleted and modified. Several metrics are calculated during each phase, e.g. throughput, latency.

Since our purpose was to assess both the performance and scalability of the Ignite and Redis distributed databases, we executed all workloads on network configurations of 6, 10, 14 and 20 nodes. We used both the Ignite and Ignite-SQL variants for Ignite. At the same time, we ran Redis with two different configurations: the first, where only data

sharding is performed across nodes, with no worker nodes, and the second, with one worker node assigned to each leader node, for data duplication, as well as sharding of data across leader nodes. All workloads were executed for performing 1000, 5000, 10000 and 20000 operations, both during the Load phase as well as the Run phase. Both Ignite and Redis were run with in-memory configurations only, for all tests with partitioned caches and full replication.

As our testbed, we used the Fogify fog computing emulator⁵ (developed by UCY) to rapidly model and configure the experiment scenarios, where an emulated geo-distributed environment was deployed. Nodes were setup as Docker image containers with 1GB RAM and single core processor clocked at 1GHz, and we restricted the (internal) network interface for each container to a bandwidth of 10000Mbps and a network latency of 3ms. We ran all tests in a system equipped with an Intel Xeon CPU E5-2620 v2 @ 2.10GHz and 64 GB of RAM. Neither the CPU nor the RAM of the host system were saturated at any point during the execution of workloads for any number of nodes.

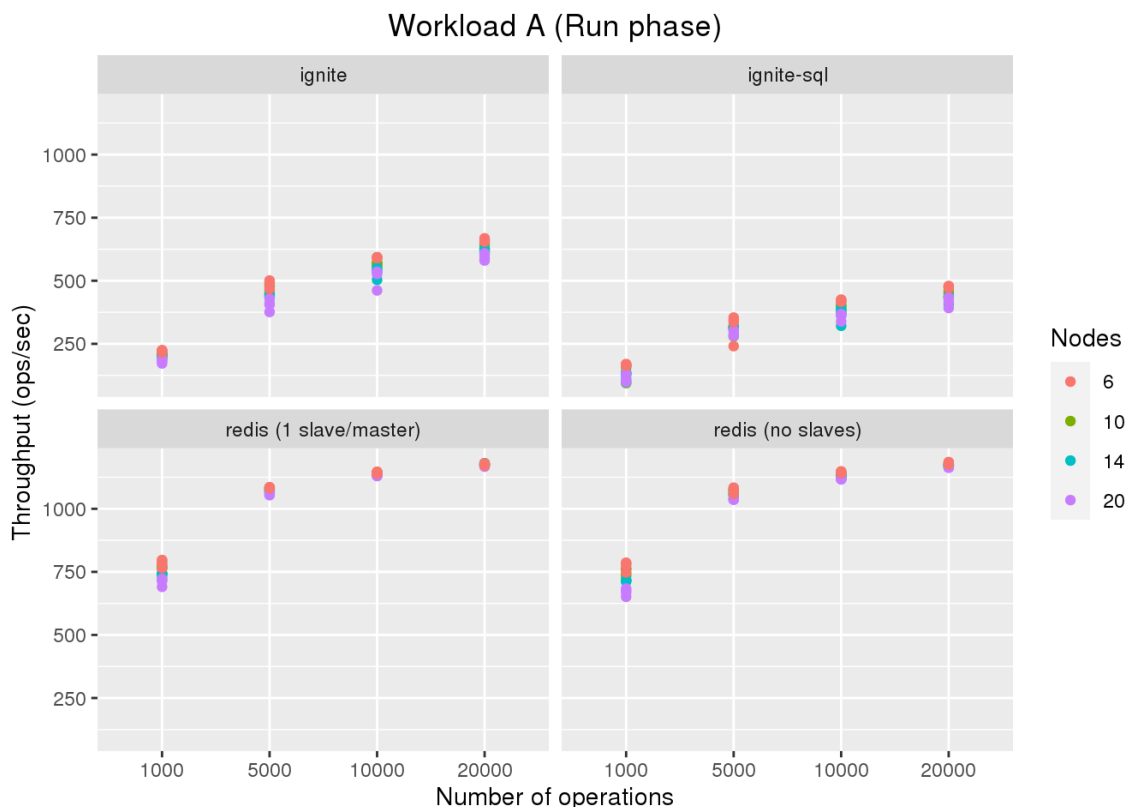


Figure 3: Ignite and Redis comparison for workload A of YCSB

Indicative results from the executions of a subset of the workloads are presented in Figure 3 and Figure 4. The behavior is similar for the rest of the workloads. Every

⁵ <https://ucy-linc-lab.github.io/fogify/>



experiment was run four times for completeness and the results of each run are presented. The chosen workloads are A and B which involve heavy updates and read operations, since this will be the case and the service's job on the RAINBOW platform.

In most cases, Redis seems to outperform Ignite by at least a small margin. It also seems to be the case that the results for Redis show less deviation than Ignite as the respective throughput values are more closely packed. This is becoming more evident as the number of nodes increases. The Ignite-SQL behavior seems to be slightly less efficient than simple Ignite. In addition, the two different configurations of Redis (no worker nodes and 1 worker node per leader) show very similar results.

However, as the number of operations increases, it seems that Ignite scales better than Redis. Even if, in most cases, Ignite does not reach the performance of Redis, the performance gap decreases as the number of operations increases. The performance of Redis seems to reach a plateau, while it appears that is possible for Ignite to further improve performance with an even larger increase for the number of operations. This improvement provides evidence that Ignite behaves better in dynamic environments, like the RAINBOW ecosystem, where many operations are needed.

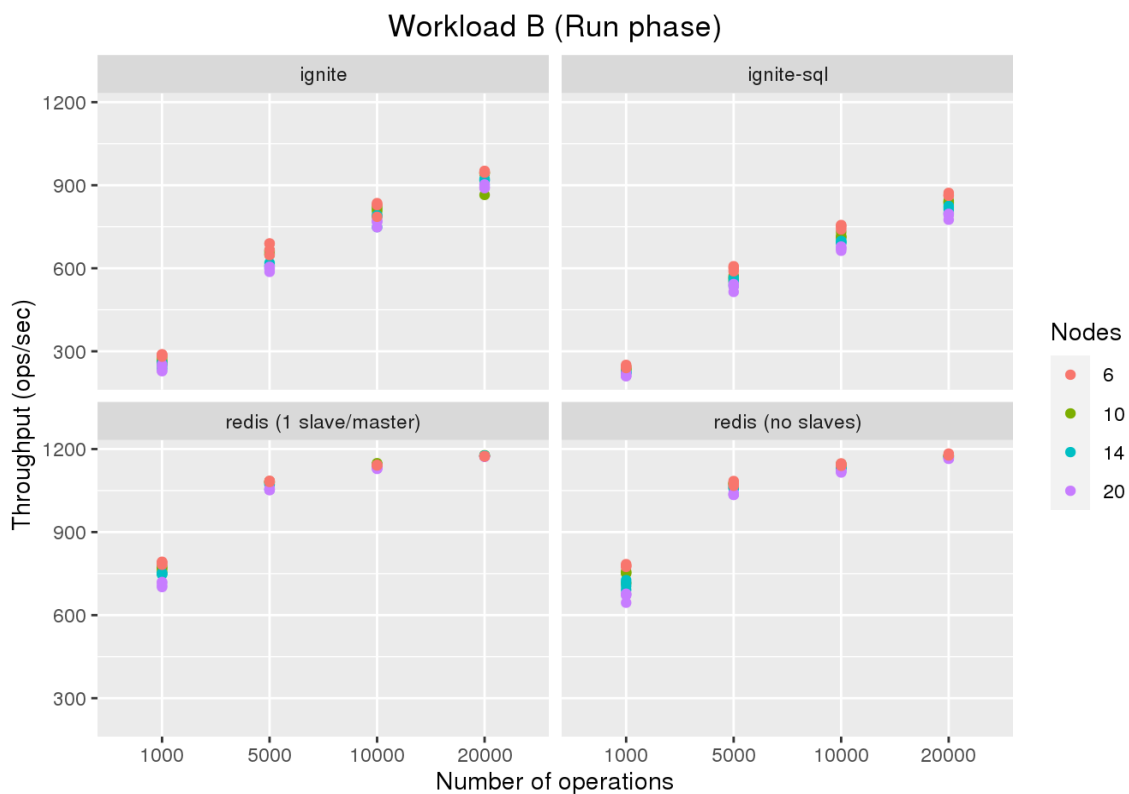


Figure 4: Ignite and Redis comparison for workload B of YCSB



3.2.4 Server and Client Instances

The RAINBOW Data Storage and Sharing service implements both *Server* and *Client* instances and deploys them on the different nodes based on their types. Every Fog node with the RAINBOW mesh stack has one *Server* instance. On the other hand, Cloud nodes and optionally resource-rich Fog nodes have the *Client* instances deployed.

Each *Server* instance is responsible for storing the local (monitoring) data that are ingested through the fog node's *Monitoring Agent*. The instance contains 3 different local caches. The first one is a key-value cache that is purely in-memory and stores only the latest values for the monitoring metrics. This cache is used for quick I/O of the metrics for the different RAINBOW services that require them, i.e. the analytics agent. The second cache is a key-value one with SQL-like properties like indices on different fields. This cache is used to persistently store the historical values of the monitoring metrics up to a user-specified time range. The time range is used to limit the volume of data and the memory resources needed for the storage. A third cache with SQL-like properties is available with persistency enabled to store metadata the mapping of a metric to an entity. Figure 5 presents the schemas for each cache with the keys being above the line and the values below.

Latest cache	Historical cache	Metadata cache
metric ID	metric ID - index	entity ID - index
value	timestamp - index w/ desc order	metric ID - index
timestamp	value	entity type
		description
		units
		name
		group name
		min value
		max value
		higher is better

Figure 5: Local data caches schemas

Finally, a partitioned in-memory key-value cache is available for storing and extracting user-application data and caching the networking routing tables. All *Server* instances have access to this cache with the data being distributed between them.



The second type of instance is called *Client* and is deployed mainly to cloud nodes. This type of instance does not store any data and its main responsibility is the control of the data placement. The instance through the use of appropriate algorithms, decides the partitioning, replication and general placement of data from a source *Server* instance to a target one. At the time of writing the deliverable, the data movement mechanisms are implemented whilst the techniques that take into consideration the analysis queries and resource and network utilization are partially implemented. Whenever a decision for data movement is taken, the *Client* communicates through an internal service with the source *Server* which in turn moves the necessary data to the target.

A second job of the *Client* is to act as an intermediate on the data extraction. The extraction is available locally by querying the respective *Server* instance or globally by querying the *Client*. The *Client* can accept authorized queries that involve more than one instances and through the respective internal services communicates with the instances at interest to gather the required data. The result can either be the whole set of data or a processed result, i.e. the average CPU load of all nodes. Also, the *Client* instances have direct access to all partitioned caches for the routing tables and user-application data.

3.2.5 Components

The first component is responsible for the *data ingestion*. This is available only on the *Server* instances which are storing the data. This component is available through a REST API and every service that needs to store data and is authorized to do so can use it by passing the data in a json format. The ingestion afterwards stores the data in the necessary caches. If the data come from the Monitoring Agent, they are stored both on the latest cache overwriting the values and on the historical one for persistent storage. On any other case the data are written to the specified key-value in-memory cache based on their usage.

The second component is responsible for *data extraction* based on some queries and is available through a REST API. The data extraction component is available from both the *Server* and the *Client* instances with similar functionality.

On the *Server* instances the component takes a query and outputs the resulting local data. The queries can have different filters such as the entity or metric ids and/or a time period. On the *Client* instances the queries have an additional variable which is the node in question. The user can either get data from specified nodes or from every available one. Also, the output from the *Client's* data extraction can be an aggregated value over the monitoring data.



Finally, the data extraction component works in a similar way with the user-application data cache and the networking routing table cache. The difference in this case is that there are no filters on the queries since only a custom key-value pair is stored and is agnostic to the Data Storage service.

The third important component is the *data rebalancing* one and is deployed on both types of instances. This component comprises different internal services based on the type of instance. In the *Server* instances, the component is responsible for the data movement from the local *Server* to a remote one. The implemented choices for movement are either the partitioning of the data based on a filter or a full replication of them to the target.

On the *Client* side the component is responsible for deciding whether a movement is necessary as well as the source and target destination. This decision is taken by considering factors such as a fog node's resource utilization, e.g. CPU load and memory consumption, the network connection and limits between the fog nodes as well as the analytics queries and the placement of the Analytics Workers. These mechanisms are partly implemented.

3.3 Interaction with other RAINBOW Services and Components

The Data Storage and Sharing service interacts with five RAINBOW components of the platform as well as the user application(s) running on the control plane through an exposed REST API. The access to data must be authorized before processing the queries. The authorization is part of the attestation/security service and is documented in the deliverables of WP2.

In particular, the Data Storage and Sharing service interacts with the following RAINBOW components:

- **Monitoring Agent:** This component utilizes the data storage to store the locally extracted monitoring data based on the monitoring data model and schema. Both a monitoring and a data storage instance are deployed together on each fog node and whenever the metrics are extracted the Agent "pushes" them through the API to the local data storage instance. The latest monitoring data are stored as real-time data in the in-memory cache for fast querying whilst also persistently saved along with the rest of the historical data. Through the Monitoring API, the Storage provides users with secure access to both real-time and historic monitoring data by authorized entities via both a push and pull data delivery interface.
- **Distributed Data Processing Service.** This component interconnects the collaborating Analytics Workers deployed over the mesh network of an IoT application's fog continuum and utilizes the data storage to continuously query



both the monitoring and application-specific data so that high-level analytic insights expressed through user-expressed continuous analytic queries can be derived. The data are either queried from the instances themselves or the client instance for the specified stored data.

- **Attestation/Security Service.** The Data Storage and Sharing service cooperates with the attestation and security service for authentication. Every data access request must first be authorized in order to be processed. Both the requesting components and user-application(s) should be part of the control plane and therefore authenticated by the security service.
- **Side-Car Proxy.** The Data Storage and Sharing service utilizes the fog node's side-car proxy in order to extract the routing tables used for encrypted networking between collaborating entities comprising the RAINBOW overlay mesh network. Each data storage instance extracts and stores the local node's routing table for further querying. In addition, the tables are updated regularly from the network components to be fully aligned.
- **Orchestrator Service:** The SLO enforcement component or the RAINBOW Orchestration service, utilizes real-time and historic monitoring data to derive if deployed applications and the underlying virtual and containerized infrastructure must expand or contract in order to meet current demand, achieve targeted performance and efficiently utilize provisioned resources.

3.4 API and Documentation

Internally, the RAINBOW Distributed Data Storage and Sharing service interacts using the internal Ignite components deployed as independent (micro-) services, each on a Dockerized container. Each instance can call a service of another one even if it is not implemented in it, e.g. a *Client* instance can call the internal data extraction micro-service of a *Server* instance to quickly query the local data.

All internal micro-services are based on Java interfaces with pre-defined public functions that are visible to every Ignite instance. All three implemented components for data ingestion, extraction and rebalancing are built on top of these micro-services and can be called from members of the distributed database cluster.

Finally, the communication between the nodes is executed using specific identifiers that the cluster imposes on each instance whenever it joins in. Every instance is discoverable by other instances in the cluster and can be identified either by the hostname and address or by the unique identifier. This process makes the use of micro-services for specific nodes easier to handle.



On the other hand, for external connections a REST API has been implemented. The external connections include only the components of data ingestion and extraction and not the data rebalancing one which works internally in the database cluster. Every interested RAINBOW service has access to their respective data caches to write and/or read data based on the permissions.

The REST API allows POST requests through a port on every node it is deployed at. For the *Server* instances there are two available request routes, the first one is for data ingestion whilst the second one is for data extraction. Both of them write and read data from the local caches. The input for each request is a json format with the necessary fields that act as filters on the queries.

On the ingestion requests a list of the input metrics is necessary along with the essential fields. This list needs to be under the “monitoring” field to distinguish the monitoring data from other possible user-application data. An example json input is available in Code Snippet 1.

For the extraction part the requests have optional fields that are used as the query filters. These fields include the metric id, the entity id and a Boolean variable that represents whether the latest or historical data are included. When historical data are needed two more fields that represent the time period of extraction are required. Two example json inputs for data extraction are available in Code Snippet 2 and Code Snippet 3.



```
{ "monitoring": [
  {
    "entityID": "entity1",
    "entityType": "type1",
    "metricID": "metric1",
    "name": "name1",
    "units": "metric units",
    "desc": "first metric",
    "group": "group1",
    "minVal": 5,
    "maxVal": 10,
    "higherIsBetter": true,
    "val": 6,
    "timestamp": 2611318068000
  },
  {
    "entityID": "entity2",
    "entityType": "type2",
    "metricID": "metric2",
    "name": "name2",
    "units": "metric2 units",
    "desc": "second metric",
    "group": "group2",
    "minVal": null,
    "maxVal": 10,
    "higherIsBetter": true,
    "val": 6,
    "timestamp": 2611318068001
  }
]
}
```

Code Snippet 1

```
{
  "metricID": ["metric1"],
  "from": 2611318068000,
  "to": 4611318068000,
  "latest": false
}
```

Code Snippet 2

```
{
  "entityID": ["entity1", "entity2"],
  "latest": true
}
```

Code Snippet 3

The complete source code of the Distributed Data Storage and Sharing service can be found in the RAINBOW source code repository:

<https://gitlab.com/rainbow-project1/rainbow-storage>



4 Distributed Data Processing Service

In this Section, we present a comprehensive documentation report introducing the reference architecture, exposed functionality and implementation details referring to the Distributed Data Processing Service.

4.1 Requirements and Exposed Functionality

Based on the user groups documented in D1.1, the identified users interacting with the RAINBOW Distributed Data Processing Service are presented in Table 5 and are the following:

Table 5: RAINBOW Distributed Data Processing service and interacting user groups

User Group	Interaction with RAINBOW Distributed Data Processing Service
Service Operator/Owner	Interacts with the RAINBOW Distributed Data Processing Service by selecting certain optimization strategies for the to-be executed analytics jobs based on the business aspects of the deployed IoT application. This is performed through the Analytics Perspective of the RAINBOW Dashboard.
Service Developer	Interacts with the RAINBOW Distributed Data Processing Service by developing the actual analytics job(s) that the Service Operator/Owner has selected strategies for job optimization. This job can either be developed using the underlying big data engine's programming model or via the RAINBOW-supported high-level declarative analytics query model (T4.3). In both cases, analytic jobs are submitted through the Analytics Perspective of the RAINBOW Dashboard.
RAINBOW Developer	Interacts with the RAINBOW Distributed Data Processing Service by developing ready-to-use schedulers, that are capable of optimizing analytic jobs based on user-desired optimization strategies and trade-offs, including, but not limited to, performance, latency, energy-consumption and data quality.
Fog Infrastructure Provider	Interacts with the RAINBOW Distributed Data Processing Service as the entity providing the fog offerings hosting the RAINBOW-enabled IoT applications in order to assess meta-



	analytics relevant to the execution of analytic jobs to optimize the quality of service of the provisioned fog resources.
--	---

4.1.1 Functional Requirements

The RAINBOW system requirements, documented in D1.1, referring to the Distributed Data Processing Service are the following:

Table 6: System-wide RAINBOW function requirements relevant to Distributed Data Processing service

Req. No.	Requirement
FR.23	Analytics execution mode

To satisfy the aforementioned system requirements, while also adhering to the key technology axes of the RAINBOW Data Management Layer, as presented in D1.2, the following functionality must be exposed by the RAINBOW Distributed Data Processing Service.

ID	FR.DPS.1
Title	Execute analytic tasks over heterogeneous fog nodes
Description	The RAINBOW Distributed Data Processing Service must provide the means for its <i>Analytic Workers</i> to execute analytic tasks in place and over heterogeneous fog nodes. This means that fog nodes may be configured with a wide range of resource capabilities, while the configuration of the Analytics Workers must be done with full transparency and no additional effort from RAINBOW users (zero-conf).
Exposed Functionality	The Distributed Data Processing Service's <i>Analytic Workers</i> are designed as configurable long-running daemons that upon instantiation are able to acknowledge how they should be parameterized so as to provision a dynamic set of lightweight threads (executors), even at runtime, dedicated to one (or more) fine-grained analytic task(s). This means that it is perfectly natural for the <i>Analytic Workers</i> of a fog deployment to feature different memory, compute and number of processing cores reserved from the underlying host (fog node).



ID	FR.DPS.2
Title	Deploy analytic jobs in geo-distributed fog topologies
Description	The Distributed Data Processing Service must be able to deploy and execute analytic jobs over geo-distributed environments and even function under heterogeneous networking settings between the <i>Analytics Workers</i> and the <i>Analytics Executor</i> .
Exposed Functionality	The configuration of the topology that will execute continuous analytics jobs is handled by the <i>Analytics Executor</i> and all network mappings between <i>Analytics Workers</i> are performed via the RAINBOW overlay mesh network so that requests for resources (e.g., fog nodes) spanning across different networks can be performed via re-active data routing. In turn, depending on the optimization strategy requested by the user, the <i>Analytics Scheduler</i> will attempt to reduce the movement of data across networks to ensure low query latency by avoiding straggling tasks due to query operators requiring data from nodes across (network) “distant” fog nodes (see FR.DPS.3).

ID	FR.DPS.3
Title	Analytics task placement based on different job optimization strategies for geo-distributed fog realms
Description	The RAINBOW Distributed Data Processing Service should provide users with the flexibility of selecting the policy/policies under which the task placement and execution of their analytics jobs will be optimized by the <i>Analytics Scheduler</i> . Specifically, RAINBOW must support a wide range of schedulers, each of which are embedded with an algorithmic process capable of optimizing the task placement of IoT applications deployed even in geo-distributed fog environments.
Exposed Functionality	The RAINBOW Distributed Data Processing Service exposes an extensible scheduler interface that eases the design and implementation of analytic job schedulers that can optimize the placement and execution of analytic tasks by the underlying <i>Analytics Workers</i> that are deployed, even, on heterogeneous fog nodes. Hence, upon the configuration of the Distributed Data Processing Service, the <i>Analytics Scheduler</i> is realized, with the respected algorithmic process for runtime task placement and can support even streaming analytics jobs. A realized placement algorithm takes as input the service graph, nodes’ ids, analytic job tasks, etc., while the system expects as output a well-defined placement strategy. Despite the design and implementation of 5



	novel scheduling algorithms, since the optimization can take various forms and be dependent on different KPIs of an application's business aspects, the RAINBOW Distributed Data Processing Service provides users with the ability to “plug-in” their own implementations with custom tailored optimizations that adhere to the RAINBOW scheduler interface.
--	---

ID	FR.DPS.4
Title	Operation under dynamic topology adaptation
Description	The Distributed Data Processing Service must be able to acknowledge dynamic alterations of the underlying infrastructure. These alterations may take various forms and include the change of provisioned resources, including the alteration of the current fog node(s) resources and/or the addition/removal of fog nodes.
Exposed Functionality	The Distributed Data Processing Service's <i>Analytics Executor</i> is designed to acknowledge dynamic changes to the underlying topology by continuously assessing the status of the infrastructure for the resources allocated to the analytics job via the RAINBOW <i>Resource Manager</i> . If a change has occurred to the underlying deployment, then the <i>Analytics Scheduler</i> is subsequently informed so that the continuous assessment of the task placement configuration is updated based on the algorithmic process acknowledging the difference in both fog node resources (vertical scaling) and the (de-)provisioning of fog nodes (horizontal scaling).

ID	FR.DPS.5
Title	Operation under unexpected events and extreme network uncertainties
Description	The Distributed Data Processing Service must be able to both acknowledge that the current deployment is undergoing unexpected events, at the same time, continue seamlessly and uninterrupted the execution of analytic jobs. These unexpected “events” may take various forms and include sudden increases in link/network latencies, the appearance of link failures, temporal link disconnections, node processing saturation and complete device fail-stops.
Exposed Functionality	Since the increase of processing delay may not be directly related to resource saturation (see FR.DPS.4), the <i>Analytics Executor</i>



internally speculates the execution of running jobs and tasks based on historical statistics. When an incident or an abnormal situation happens, the *Analytic Executor* decides if the alteration of the underlying infrastructure is transient or permanent. In the first case, the system temporarily redirects the load from problematic workers to other, under-utilized, *Analytics Workers* so that it can self-stabilize under transient faults. If the problem persists after a considerable time interval (e.g., a node fail-stop), then the *Analytics Executor* will consider the affected resources to be permanently unavailable and the *Analytics Scheduler* will be notified so that the algorithmic process does not assign tasks to nodes based on the aforementioned faulty resources.

4.1.2 Non-Functional Requirements

ID	NFR.DPS.1
Title	Robustness
Description	The Distributed Data Processing service must cope with any potential errors from unexpected inputs and faults during the execution, while also continue to work as usual after an interruption from unexpected crashes by restoring its last valid state.

ID	NFR.DPS.2
Title	Near Real-time Adaptability
Description	The Distributed Data Processing service must be able to adapt itself to the demands of the workload and requirements of the various optimization algorithms without degrading its performance, by adding or removing the necessary components to remain functional and produce in real-time or at least near real-time analytic task placement and execution decisions.

ID	NFR.DPS.3
Title	Extensibility
Description	The Distributed Data Processing service must support developers with the means to develop and “plug-in” their own customized analytic job scheduling algorithms so as to optimize the execution of analytic jobs that meet the certain optimization policies that the user deems as



important for a specific job executed in a geo-distributed fog environment.

4.2 Reference Architecture and Implementation

The Distributed Data Processing service is responsible for the RAINBOW ecosystem's needs for data processing so that real-time analytic insights can be extracted from the vast amounts of monitoring data collected from both the underlying fog resources and performance indicators from deployed IoT applications. To this end, the service provides a completely distributed solution with the data processing performed -in place- right where the data is generated so that analytic insights are extracted with low-latency and with the collected data never leaving the overlay mesh network interconnecting the collaborating fog nodes.

To this end, the Distributed Data Processing service builds upon Apache Storm⁶ with our aim being to not implement yet another distributed data processing engine but rather to design novel scheduling algorithms that are decoupled from the underlying engine and acknowledge the unique settings found in the majority of geo-distributed environments that IoT applications are deployed in.

4.2.1 High-Level Logical Overview of Analytics Workflow

Figure 6 presents a comprehensive overview of the logical interplay between the Distributed Data Processing service and interacting RAINBOW components. A typical flow starts with a *Service Operator*⁷ constructing a set of analytic queries by following a high-level declarative language (1). The *Analytics Editor* provides a graphical user interface to support RAINBOW users (e.g., Service Operators) in the composition and definition of analytic queries. Furthermore, users are able to introduce a set of diverse optimization and constraint policies along with their queries. The constructs and query operators of this language are presented in Chapter 5. When the user finishes the editing, the system transfers the queries to the *Analytics Enabler* module (2). What is more, the user has the opportunity to add, amend or stop the execution of analytic queries, even at runtime, by following the same procedure.

With the analytic queries in hand, the *Analytics Enabler* is responsible to create an execution plan for the efficient calculation of the set of queries on the underlying fog infrastructure. The internal process of the *Analytic Enabler* begins with parsing the

⁶ <https://storm.apache.org/>

⁷ For more information on the identified User Roles of the RAINBOW ecosystem, please see D1.1.

queries and forming an Abstract Syntax Tree (AST) for each query. An AST is an intermediate tree-based representation in which each node represents a grammatical rule of the language and every leaf corresponds to a language's symbol. A query is syntactically correct if the *Analytic Enabler* translates it without any error. If all submitted queries are correct, the *Analytic Enabler* will then perform an initial query optimization process. In this step, the *Analytic Enabler* finds correlations between the ASTs of different queries and combines them in order to eliminate the re-computations and minimize data transfer in the execution phase. Then, the optimized execution plan is forwarded (3) to the *Analytic Executor*.

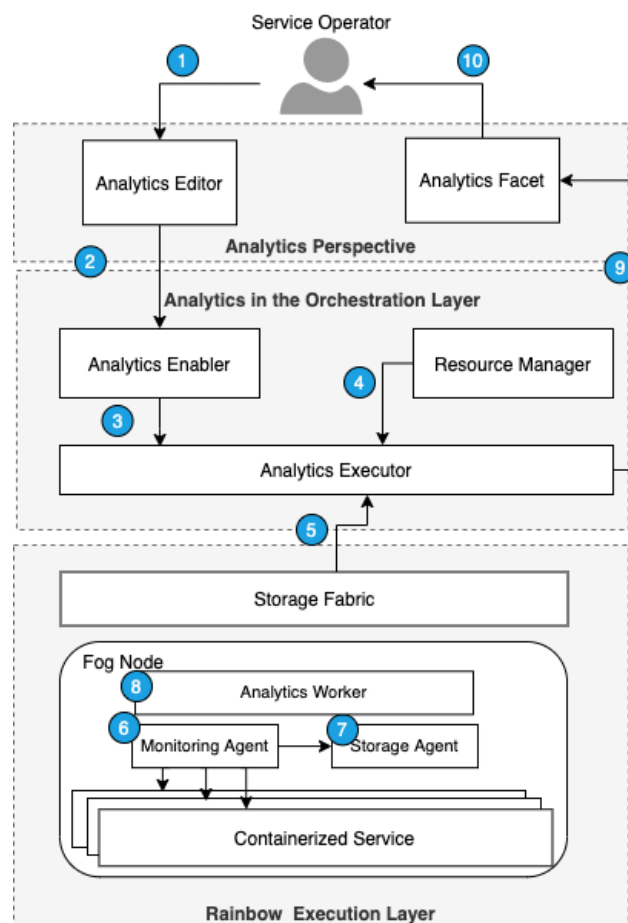


Figure 6: Logical Overview of the Distributed Data Processing service in the RAINBOW ecosystem

Specifically, the *Analytic Executor* is responsible for coordinating the execution of the queries, placing the underlying tasks to the *Analytic Workers*, and observing the non-blocking processing. To perform the latter tasks, the *Analytics Executor* requests information about the underlying infrastructure resources, like available CPUs, memory, network bandwidth, etc. from the *Resource Manager* (4), part of the RAINBOW Orchestration Services, and data placement metadata from the *Storage Fabric* (5). It should be mentioned that, in a real deployment, each *Storage Agent* is capable of



providing data locality information. So, in Figure 6, the *Storage Fabric* represents a logical sub-component that abstracts and unifies the functionality offered by inter-connected local *Storage Agents* by providing a decentralized API for access to monitoring data. Hence, monitoring data are immediately made available through the RAINBOW secure overlay mesh network without data needing to be moved to a central (cloud) location that will provide data access but with both a performance penalty and costs incurred for data movement. With information about resource availability and storage metadata, the *Analytic Executor* invokes the RAINBOW-enabled *Analytics Scheduler*, which performs an analytics task placement algorithm to provide near-to-optimal efficiency for analytic queries based on the user-desired optimization policies.

In the RAINBOW execution layer, we have three components that take part in the analytic processing, namely, the *Monitoring Agents* (6), *Storage Agents* (7), and *Analytics Workers* (8). Initially, a *Monitoring Agent* generates streams of monitoring metrics, continuously “pushing” them to the local *Storage Agent*. The generation of the monitoring streams does not fall within the scope of this deliverable, as there are many more details about the monitoring agents in the D3.1. The *Storage Agents* store the data in a distributed high-performance in-memory data structure with minimum retrieval time for the latest data. Finally, the *Analytic Workers* retrieve the stored data from *Storage Agents*, via direct access to the Storage Fabrics, process them, and forward the results back to the *Analytic Executor*. The executor disseminates results to the *Analytics Facet* of the *Analytics Perspective* of the RAINBOW Dashboard (9). This perspective is a component that retrieves the computed query results and visualizes them through different and customizable graphical representations of the data (e.g., time-series graphs, charts, etc).

4.2.2 Apache Storm

We have opted for Apache Storm as our distributed stream processing framework for RAINBOW. Apache Storm is a free and open-source system that achieves fast analysis of data in real time. Recent studies show that Storm can process millions of data tuples per second per participating cluster node [52]. This along with its scalability and fault-tolerance mechanisms fulfil the use cases’ and Edge Computing paradigm’s requirements. Moreover, Apache Storm can be used with multiple programming languages and databases, thus not limiting the pool of choices for the rest of the RAINBOW platform. Another important factor that lead to the selection of this specific framework is the ease it provides when there is a need to customize the assignment of analytic tasks to worker nodes. More specifically one can actuate a custom scheduler by implementing the `IScheduler` interface, and without the need to resort to source code refactoring. With this functionality the scheduling decisions of the optimization algorithms are enforced into the use-cases. Finally, it is very easy to generate service graphs, create dependencies



as links, set the stream grouping policies between services and configure the parallelism of the graph by means of number of workers, threads and tasks per service, allowing the RAINBOW framework to further capitalize on optimization opportunities. More information on the scheduling algorithms and user-driven optimization policies are presented in Section 4.2.4.

In relevance to other candidate frameworks, Hadoop⁸ is one of the oldest and popular open-source frameworks for distributed data processing, but it is more tailored to batch processing huge volumes of data that are persistently stored across a distributed file system (HDFS). In turn, the programming model of Hadoop adopts a pure MapReduce approach and therefore it is particularly limited in terms of query operators, while non-perfectly parallelizable MapReduce jobs face significant performance penalties when the jobs are iterative and intermediate data must move back-and-forth from the distributed filesystem.

More suitable candidate frameworks are Flink⁹ and Spark, and specifically, one of the newest extensions of Spark, denoted as Spark Streaming¹⁰. Spark Streaming adopts and extends the DataFlow programming model of Spark and therefore includes numerous query abstractions for stream data processing, including data transformations, grouping, aggregations and even filters. However, while Spark Streaming may seem like it supports a true stream processing paradigm (due to its naming), “under the hood” it is simply a wrapper framework over Spark batch processing implementing “micro-batching”. This means that ingested data is “windowed” and the processing logic is enforced on the window. In turn, introducing scheduling algorithms to Spark, and similarly to Flink, is particularly difficult as the scheduling process is tightly coupled with the implementation, meaning that the scheduling must be changed in the Spark codebase, recompiled and then deployed again in order for a change to be supported. Still, despite its scheduling limitations, Spark Streaming as a programming model can be supported by RAINBOW and its query model, and in Chapter 5 we make reference to this. However, it must be noted that users opting for this will not be able to make use of any of the RAINBOW-enabled Schedulers capable of optimizing IoT analytic jobs over fog realms.

4.2.3 Apache Storm in the RAINBOW Analytics Ecosystem

A Storm cluster architecture-wise is comprised of two basic components: a *Master* node, denoted with the name *Nimbus*, and *Worker* nodes, which are denoted as *Supervisors*. *Nimbus*, quite similar to the *JobTracker* in a MapReduce cluster (e.g., Hadoop), is the entity

⁸ <https://hadoop.apache.org/>

⁹ <https://flink.apache.org/>

¹⁰ <https://spark.apache.org/streaming/>

responsible for the analytics job coordination that includes the scheduling of analytic tasks to *Supervisors* and the overall overview of the cluster lifecycle management (e.g., handling failures). In turn, *Supervisors* are the nodes that accept analytic tasks from *Nimbus* and coordinate their execution on the local environment they have access to. For RAINBOW this environment is the fog node where the *Supervisor* is deployed on, as shown in Figure 7.

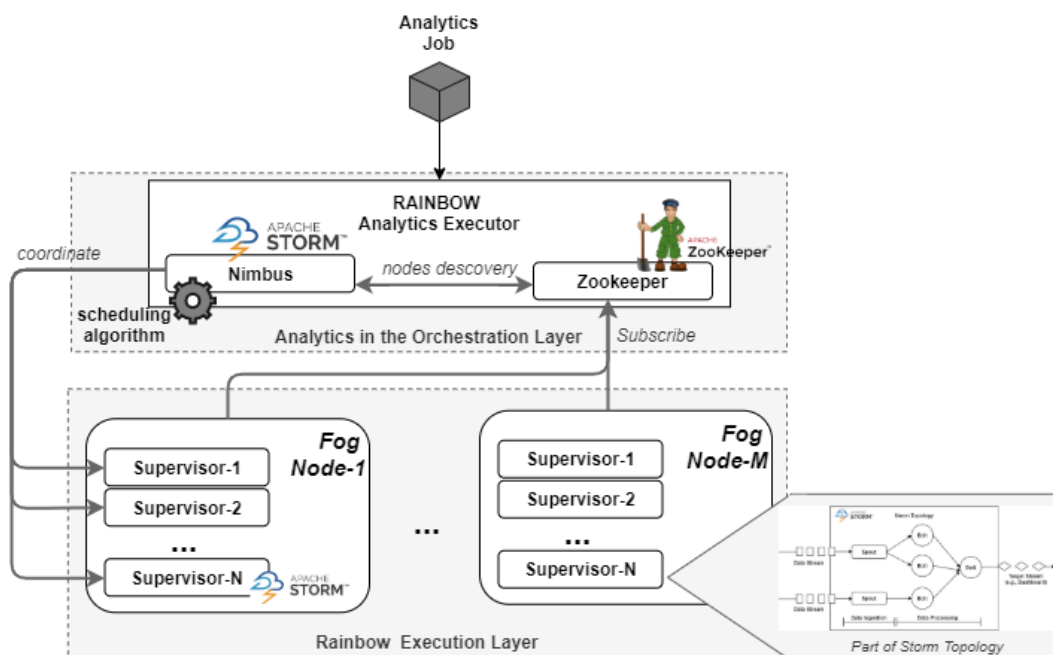


Figure 7: Storm in the RAINBOW Ecosystem

Hence, the *Supervisors* are the actual implementation of the *Analytics Workers* that the RAINBOW Mesh Stack features. In turn, *Nimbus* is one of the main software components comprising the *Analytics Executor*. It is worth mentioning that a third component is also required for the successful deployment of a Storm cluster, although not considered part of Storm per se. This third component is *ZooKeeper*¹¹, which handles the cluster communication overlay between *Nimbus* and the *Supervisor* nodes along with some additional functionality including worker health monitoring.

In Storm, an analytics query is described as a *Topology*, namely the input data structure received by the Storm cluster for continuous execution and analytics insight extraction. Note that an analytics job may contain multiple queries and therefore, multiple Topologies. In its most simplistic form, a *Topology* is a Directed Acyclic Graph (DAG) comprised of multiple nodes that can take one of two forms. Specifically, nodes can be *Spouts* or *Bolts*, as shown in Figure 8. A *Spout* node is linked to a data source and is in charge of handling data ingestion by receiving data as a stream of tuples and delegating

¹¹ <https://zookeeper.apache.org/>

these tuples to respective *Bolts*, based on the configured *Topology*. Examples of data sources are various DBMSes, OSNs, distributed file-systems and even high-performance queueing services. In turn, *Bolts* are the nodes performing the actual data processing and can be implemented to perform data aggregations, groupings, filtering and even data transformations. It is worth noting that a *Bolt* may consume data from multiple streams and, in turn, generate multiple streams that are, in turn, further connected to other downstream *Bolts*.

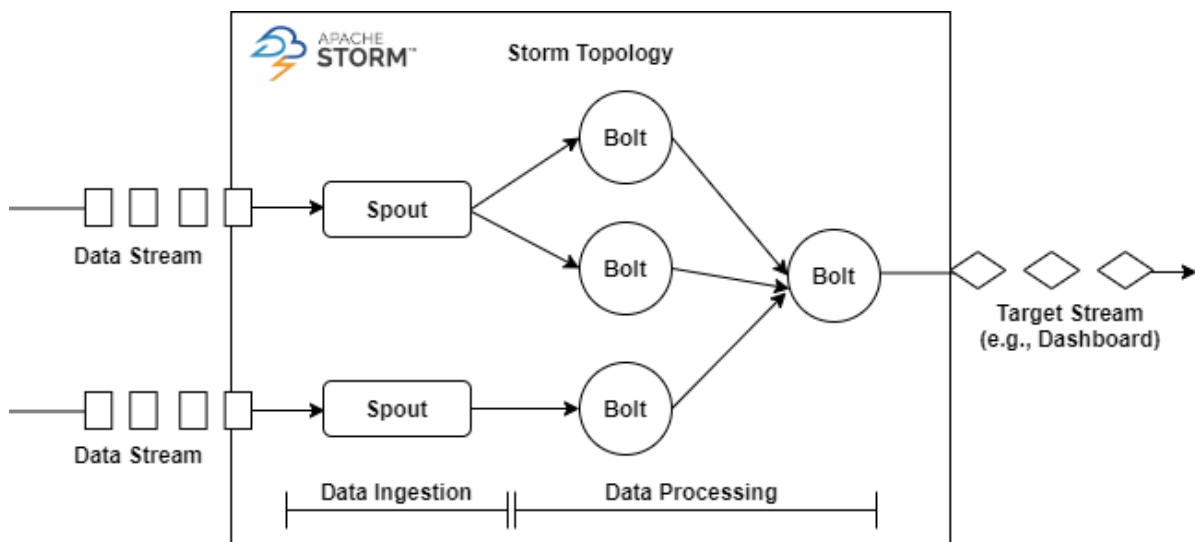


Figure 8: Storm topology

In relevance to RAINBOW-enabled IoT applications, we have designed and developed a *Spout* node capable of ingesting streaming data from the *Storage Fabric* interconnecting the *Storage* instances deployed over an application's fog nodes. Through this *Spout*, access to the Storage Fabric is provided so that monitoring data relevant to the deployed analytic queries can be accessed. Specifically, through the *Spout*, monitoring data can be extracted in two modes: either through a per *metric* request, where a specific metric is requested via its metric id or through a per *entity* request, where all the metrics relevant to a monitored entity are requested. In the latter case, an entity can be a fog node or a containerized execution environment.

In turn, several *Bolt* nodes have been designed and implemented to support popular query operators for descriptive statistics, while a number of *Bolts* are currently being designed to support machine learning operators for classification and clustering. These *Spout* and *Bolt* nodes are open-sourced and made publicly available via the RAINBOW source code repository. While users are free to use them as desired in implementing their own custom analytics Topologies, RAINBOW goes one step beyond and can automatically map analytic queries, expressed in a high-level declarative language (described in



Chapter 5), to a Topology which is then executed by a Storm cluster configured on top of an application's overlay mesh network that interconnects collaborating fog nodes.

4.2.4 Analytics Job Scheduling in Fog Realms

The heart of the RAINBOW project contribution to fog-aware distributed data processing is in scheduling analytic jobs in the fog continuum (right where data is actually generated) and to do so while adhering to user-desired policies for job optimization.

As previously mentioned, in Storm, *Nimbus* uses an `IScheduler` implementation to assign tasks to the supervisors. These tasks are the *Spout* and *Bolt* nodes of a Storm *Topology*. The default Storm Scheduler, dubbed as fog-agnostic, attempts to allocate computing resources evenly to topologies. It works well in terms of *fairness* among topologies, but it is impossible for users to predict the placement of topology components in the Storm cluster, regarding which component of a topology needs to be assigned to which supervisor node. This is a particular downside for IoT applications deployed in a fog environment where heterogeneity is the “norm”, meaning that both fog node computing resources, as well as, network links can (significantly) differ.

Nonetheless, *Nimbus* enables users to design and deploy custom schedulers by, first, adopting and implementing the `IScheduler` interface, and secondly, by assigning upon cluster configuration the relevant scheduler to *Nimbus*. The `IScheduler` interface contains two methods for implementation. Specifically, the `prepare(Map config)` method is called upon once and provides any initial configuration relevant to the custom scheduler implementation. In turn, the `schedule(Topologies topologies, Cluster cluster)` method is the method that actually performs the scheduling processing and therefore, allocating analytic tasks (segments of the Storm Topology) to *Supervisors*. The input to the scheduling process is the topologies/queries that must be decomposed into segments and assigned to supervisors, along with the `cluster` object capturing in a list all the available Supervisors that can process tasks on the cluster. Note that in a static configuration of the underlying infrastructure the scheduling process only needs to run once as no changes are foreseen to the deployment. However, this is far from the case in a highly dynamic fog continuum with heterogeneous fog nodes that are dynamically (de-) provisioned.

To support RAINBOW-enabled scheduling with the algorithmic process adopting a more data-oriented approach so that the decision-making process is more informative, the `schedule()` method is called periodically and the input given under the `cluster` object is (currently) extended to include: (i) the IoT application's service graph, (ii) the fog nodes in the current `cluster` along with their current capabilities; and (iii) up-to-date monitoring data for the fog nodes along with network statistics include link quality



and latency. Note that, as shown, in Table 7, algorithms 4 and 5 will take decisions relevant to energy consumption and costs. Hence, relevant fog node capabilities along with monitoring data must also be provided under the `cluster` object and will be made available in the next version of the Distributed Data Processing service.

The goal of the RAINBOW project is to provide analytics jobs with optimizations relevant to fog computing settings based on the user needs collected in D1.1 and also expressed in relevant scientific outputs (e.g., research papers) [2]. Towards this, RAINBOW will enable Service Operators to select one of the following optimization strategies upon deploying an analytics job over the fog realm of an IoT application:

Fog-agnostic optimization

This policy can be considered the *baseline* when evaluating various scheduling algorithms, and essentially, it provides the default Storm Scheduler, which adopts a fair task allocation approach. Going beyond the default, this Scheduler accepts the RAINBOW-enhanced `cluster` object enriched with monitoring data from the underlying cluster so that the scheduling process can be performed periodically to acknowledge dynamic changes in the cluster due to the (de-) provisioning of resources from both vertical and horizontal scaling.

Performance-based optimization

This policy attempts to optimize the placement of tasks to worker nodes by considering as the most important QoS metric the average latency of the Storm topology. Fundamentally, the average latency can be defined as the latency of the slowest path in the topology DAG with regards to a single input data batch and consists of the average communication latency between the operators in the path. This path is denoted as the *critical path*. Hence, as our focus is on geo-distributed realms, the dominating factor contributing to latency is the communication cost with the execution latency of each operator is considered negligible. Towards this, the Scheduler's algorithmic process attempts to solve a constraint satisfaction algorithm which will optimize the placement of tasks to worker nodes so that the latency of the critical path is minimized while still ensuring that the compute capabilities of an operator can be fulfilled by the candidate fog nodes in terms of CPU speed and RAM.

Optimization employing a trade-off between performance and data quality

This policy attempts to optimize the placement of tasks to workers by considering data quality as a first-class citizen and optimizes for both performance and quality. The quality of the data is an important aspect in IoT scenarios. Low quality can lead to less useful results. The quality of data can be categorized into multiple dimensions. Some examples of these are *completeness*, *timeliness* and *accuracy*. Some of the most common factors that



lead to decreased data quality include the heterogeneity of data sources, missing and dirty data due to network malfunctions or security constraints. Therefore, this policy enables the algorithmic process of the Scheduler to extend the performance-based optimization so that we can solve a problem that trades latency for an increased fraction of incoming data, for which data quality measurements are performed across heterogeneous geo-distributed devices. This problem is considered NP-hard due to the fact that the more the quality checks, the less a fog device can be assigned tasks of upstream operators, thus inducing higher communication cost, which contradicts to the optimization of the latency. Hence, in our current research efforts, a Linear Programming optimization process is designed, where for each operator we consider, first, the placement of its parent nodes and then further optimize it heuristically. As this approach may fall into local optima that are deemed far from the optimal solution, a spring relaxation algorithm introduces a solution with low or no intra-operator parallelism. These two techniques are not seen as competing to each other. After running both techniques, we choose the best one.

Optimizations employing a trade-off between performance and energy consumption and introducing capped costs

These two optimization policies will be the focus of our work for the next version of the Distributed Data Processing service. In particular, we will design an optimization process that does not only consider the communication latency but also energy consumption especially when a set of fog nodes are battery-powered. The latter is particularly important as certain nodes may be “selectable” in terms of compute capabilities but feature different power levels for analytic computations (e.g., Raspberry Pi 4-8W, Nvidia Jetson Nano 32-56W). Hence, power-hungry modules embedded in a battery-powered cluster, can severely impact an analytic job in the near future as certain nodes may be deemed unavailable very early on due to battery exhaustion. Moreover, in geo-distributed realms, various costs should be taken into consideration, spanning from the compute cost, of using third-party fog nodes to network costs incurred due to large data transfers. Hence, the user should be able to configure a cap in terms of cost to the job as a trade-off with performance (job latency). As jobs are continuous (streaming) jobs, this cost cap must not be an absolute number but rather a quantity deemed relevant in a certain (configurable) timeframe (e.g., minutes, hours, day, etc).



Table 7: Current status of RAINBOW-enabled analytics job scheduling algorithms

Scheduling Algorithmic Process	Algorithm Designed	Algorithm Implemented	Tested in Realistic Settings ¹²	Integrated within RAINBOW
Fog-agnostic (fair task allocation)	X	X	X	X
Performance-based (latency) optimization	X	X	X	X
Employing a trade-off between performance and data quality	X	X	X	
Employing a trade-off between performance and energy consumption				
Performance-based optimization with capped costs				

The current status of the scheduling algorithms for IoT analytics in fog computing environments, is presented in Table 7. We note that algorithms 4 and 5 will be part of the second release of the Distributed Data Processing service, while the integration of algorithm 3 with RAINBOW will be performed before the first release of the RAINBOW platform (M18).

4.3 Interaction with other RAINBOW Services and Components

The Distributed Data Processing Service as a key component of the RAINBOW Data Management Layer, interacts with various RAINBOW components of the platform. In particular, the Distributed Data Processing Service interacts with:

- **The Distributed Data Storage and Sharing Service:** as mentioned in Chapter 3, this component interacts with the Distributed Data Processing service by serving streaming (monitoring) data for IoT analytics computation.
- **The RAINBOW Dashboard:** through the Analytics Perspective, this component interacts with the Distributed Data Processing Service by: (i) enabling users to submit continuous analytic jobs for execution; (ii) enabling users to define optimization policies for improving certain KPIs of the execution of analytic jobs (e.g., optimize performance, energy consumption); and (iii) providing intuitive plotting and visuals to users for exploring the real-time insights extracted from the analytic jobs.

¹² using the fogify emulator (<https://ucy-linc-lab.github.io/fogify/>) to deploy various IoT applications under different scenarios.



- **The RAINBOW Orchestrator:** through the SLO Engine, this service utilizes the Distributed Data Processing Service to submit analytic jobs that retrieve, process and extract high-level analytic insights from performance metrics that are extracted from RAINBOW Monitoring after monitoring the underlying resource utilization of the fog nodes an IoT application is deployed on. With this data, the SLO Engine can take intelligent decisions about the deployment's runtime behavior and if something is deemed "troubling", then corrective actions can be enforced (e.g., auto-scaling).

4.4 API and Documentation

Despite the fact that RAINBOW users can design and submit analytic queries and entire jobs for execution via the Analytics Editor part of the RAINBOW Dashboard, the Distributed Data Processing Service provides a REST API, depicted in Table 8, for developers to create applications that can interact with the service without the need of the intermediate UI.

Table 8: Distributed Data Processing Service REST API

Path	Method	Parameters	Description
/api/insights	POST	Desired queries	With this method, entities submit queries adopting the declarative fog analytics query model. The system validates their syntax and returns a <i>deployment-id</i> for the submitted analytics job.
/api/insights/<deployment-id>	PUT	Amended queries	With this method, entities update submitted queries with the <i>id</i> referring to the <i>deployment-id</i> .
/api/insights/<deployment-id>	DELETE	-	This method deletes, or better un-deploys, a submitted analytics job by referring to the job with its <i>deployment-id</i> .
/api/insights/<deployment-id>	GET	-	This method returns the current status of an analytics job along with the submitted queries as a reference based on the given <i>deployment-id</i> .



Project No 871403 (RAINBOW)

D4.1 – Data Management Services – Early Release

Date: 31.03.2021

Dissemination Level: PU

The complete source code of the Distributed Data Processing service can be found in the RAINBOW source code repository:

<https://gitlab.com/rainbow-project1/rainbow-analytics>



5 Fog Analytics Service

In this Section, we present a comprehensive documentation report introducing the reference architecture, exposed functionality and implementation details referring to the Fog Analytics Service.

5.1 Requirements and Exposed Functionality

Based on the user groups documented in D1.1, the identified users interacting with the RAINBOW Fog Analytics Service are presented in Table 9 and are the following:

Table 9: Fog Analytics service and interacting user groups

User Group	Interaction with RAINBOW Fog Analytics Service
Service Operator/Owner	Interacts with the RAINBOW Fog Analytics Service by submitting analytic jobs to extract insights via the RAINBOW high-level descriptive query model, and afterwards, define certain optimization strategies for the to-be executed analytics jobs based on the business aspects of the deployed IoT application. In turn, the Service Operator/Owner can observe in the extracted insights through intuitive plots and visuals. All of these functionalities can be performed through the Analytics Perspective of the RAINBOW Dashboard.
Service Developer	Interacts with the RAINBOW Fog Analytics Service by submitting analytic jobs to extract insights from deployed IoT applications via the RAINBOW high-level descriptive query model. This can be performed either through the Analytics Perspective of the RAINBOW Dashboard or by submitting an analytic job(s) directly through the API of the Distributed Data Processing Service.
RAINBOW Developer	Interacts with the RAINBOW Fog Analytics Service by developing (i) ready-to-use query operators for the high-level descriptive query model, (ii) new compiler(s) that map the RAINBOW query model to the programming paradigm supported by underlying distributed data processing engines (DPE); and (iii) designing novel algorithms optimizing the mapping of the RAINBOW query model to the underlying DPE model so that the data processing pipeline is optimized for



	certain conditions (e.g., intermediate data reduction, approximate answers, etc).
--	---

5.1.1 Functional Requirements

The RAINBOW system requirements, documented in D1.1 referring to the Fog Analytics Service are the following:

Table 10: System-wide RAINBOW function requirements relevant to Fog Analytics

Req. No.	Requirement
FR.22	Compile and Execute of analytic insights through a high-level and descriptive query model

To satisfy the system requirements documented in D1.1 while also adhering to the key technology axes of the Analytics Engine presented in D1.2, the following functionality must be exposed by the Fog Analytics Service.

ID	FR.AS.1
Title	High-Level declarative query model for fog analytics
Description	The Fog Analytics Service must provide RAINBOW users with the ability to design analytics jobs composed of queries extracting analytic insights from monitoring data harvested by the deployment of their IoT applications over a fog environment. To achieve this a descriptive query model must be provided with the syntax, for query composition, understandable even from non-expert users and should not imply knowledge of a particular programming model or assume a specific distributed processing engine.
Exposed Functionality	Many user roles will interact with the Fog infrastructure through the RAINBOW Query Model. For instance, developers who would like to evaluate the performance of deployed services, or/and service owners, who would like to observe the well-functionality of the whole system. Given that users may not be familiar with distributed computations, we offer them an abstract query language tailored for Fog and Edge monitoring analysis. Specifically, users declare their desired queries with operations like Window Operators, Accumulated Operators, Compositions, and Filters Predicates. Especially, Window Functions are



	<p>aggregations, like sum, average, max, min, etc. that apply to values that fall in a time window. On the other hand, Accumulated Functions, like running average, running max, running min, EWMA, etc., generate the cumulative results from the time that insight is deployed up to the "latest" incoming data point. To produce more sophisticated insights, users can apply Compositions, which gives users the opportunity to combine multiple windows or/and accumulated operators in a single insight. Finally, Filters can be applied on any stream, and, obviously, discard the values of the stream that do not match the filter predicate criteria.</p>
--	---

ID	FR.AS.2
Title	Streaming analytics (continuous queries)
Description	The Fog Analytics Service must support streaming analytic queries that will be evaluated in real-time. This requirement comes from both users, which would like to observe their applications and be aware of inefficiencies, and the RAINBOW scheduler that needs to analyze the monitored data as soon as possible.
Exposed Functionality	There are two approaches in streaming data processing, namely, continuous execution, where the processing is performed when a new datapoint comes into the system, and the (micro-)batch streaming processing, where the system splits the incoming datapoints into buckets and processes them in predefined intervals (e.g., every 5 seconds). The RAINBOW query model supports both methods with the default to be the pure-streaming approach and the (micro-) batching approach to be declared with the EVERY keyword. We decided to provide both methods because each one has different benefits in Fog Infrastructure monitoring. On the one hand, the continuous execution offers real-time responses but requires more compute resources. On the other hand, the micro-batch processing increases the response's latency, by adding the interval on it, but occupies fewer underlying resources. To this end, users are able to select if they need pure real-time results, by sacrificing more resources, or are tolerant in latency for better resource utilization.

ID	FR.AS.3
Title	Query model decoupled from the underlying processing engine



Description	Fog Computing is a constantly evolving environment, so RAINBOW query model should not be coupled with a specific underlying engine. Specifically, the model should be easily translatable into different distributed engines with minimum effort.
Exposed Functionality	The RAINBOW Query Model is not designed as a direct extension of any distributed processing engine. Rather, it adopts abstract syntax trees as intermediate representations of the queries, so that the query model can be optimized and compiled irrespective of the underlying distributed processing engine. Towards this, users can write queries and analytic jobs once, and use the same set of queries anywhere without needing to scratch the entire job and start again when changing the execution environment. This avoids in a sense the “analytics governance lock-in” and enables interoperable query descriptions. For RAINBOW three compilers will be made available for the aforementioned query model. These will support spark-streaming, storm and the rainbow-enabled distributed data processing service utilizing storm but with fog-aware scheduling.

ID	FR.AS.4
Title	Optimization of generated analytics job execution plan
Description	The compilation of a set of queries should generate a highly optimized executable that minimizes the data transfer and computations in the execution time.
Exposed Functionality	As we mentioned before, the RAINBOW stack generates a set of ASTs as intermediate representations of the queries. Taking advantage of the ASTs formalism, Query Optimizer eventually recognizes similar statistics between queries and eliminates the re-calculation of them. Thus, the generated optimized execution plan minimizes the re-computation between different tasks and, consequently, generates less network traffic and computational footprint to the underlying infrastructure. Finally, performing calculation pruning before the compilation phase, makes the optimization process beneficial independently from the underlying processing engine.



5.1.2 Non-Functional Requirements

ID	NFR.AS.1
Title	Query Model Ease of Use and Expressivity
Description	The use of the RAINBOW query model must be (significantly) simpler than using directly the underlying processing engine's programming model and the query model expressiveness must cover the majority of the operators supported by the underlying engine and are required when defining streaming analytic queries for IoT services.

ID	NFR.AS.2
Title	Model Extensibility
Description	The RAINBOW query model must be extensible to support the addition of new query operators and the addition of new operators must be supported without affecting prior and currently running queries.

ID	NFR.AS.3
Title	Query Operator Encapsulation
Description	The alteration of existing operators of the RAINBOW query model, either for improvement or to fix certain issues (e.g., bug), must not affect prior and currently running queries.

5.2 Reference Architecture and Implementation

RAINBOW eases the distributed data processing of streaming (monitoring) data by providing a complete streaming analytic stack, ranging from the query definition and optimization to analytic jobs' deployment and visualization of the results. Figure 9 illustrates a high-level overview of the system's architecture.

We design and implement the Fog Analytics Service in a way that abstracts and hides all unnecessary information from the end-users. Specifically, a *query model* is introduced with decoupled abstractions from the underlying processing engines that can express a wide range of analytic queries and optimizations related to the processing of data streams generated across fog and edge computing realms. The system translates the queries into Abstract Syntax Trees (ASTs) and optimizes the correlations between them. Then, the system is able to translate the queries into the programming model of the underlying processing engine. Finally, an executable with generated optimized DAGs will

be seamlessly deployed to the processing engine. We should note here that, a wide range of optimizations offered by the model can be performed by providing "hints" to the *Analytics Scheduler* of the RAINBOW Distributed Data Processing Service. Nonetheless, analytic jobs output by the Fog Analytics Service can still run on any Apache Storm and Spark cluster but the jobs will function without the user of any RAINBOW-enabled optimizations as the baseline schedulers of these systems do not optimize analytic jobs for fog-enabled IoT applications.

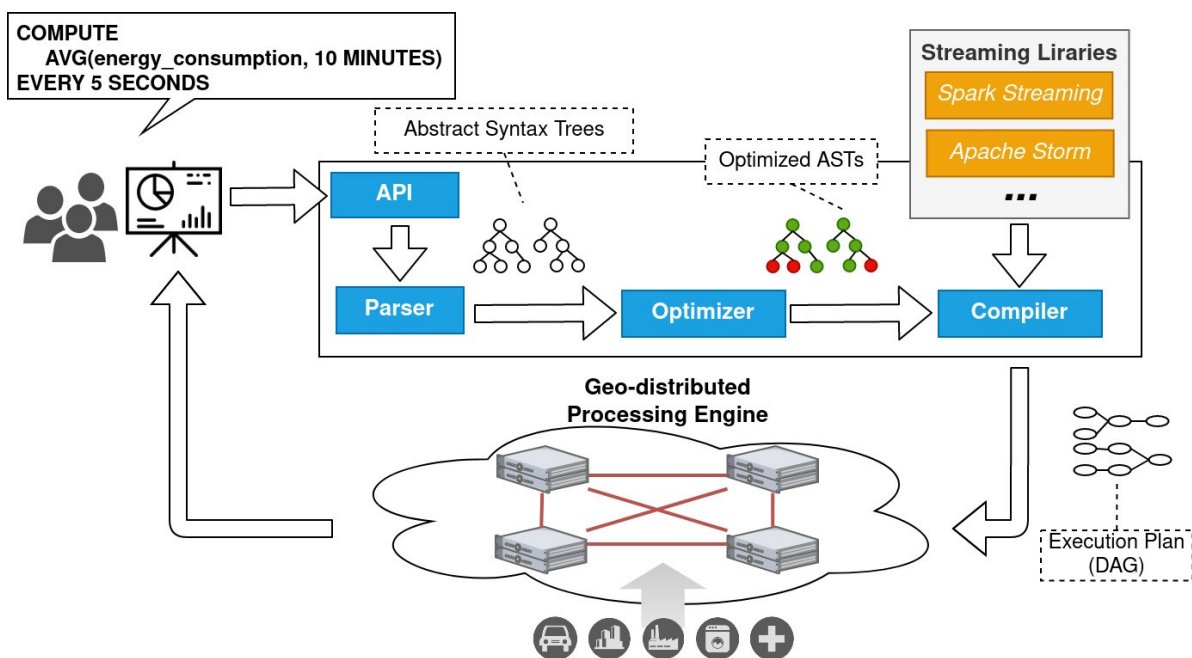


Figure 9: High-Level Overview of the Fog Analytics Cycle

The main components of the RAINBOW Fog Analytics Service are the following:

- Query Model:** Strictly speaking, the query model is not an actual system component. However, the query model is the abstract language, used by RAINBOW users to express analytic queries in a high-level declarative format. The alternative is for users to resort to expressing analytic jobs as coded query operator pipelines that adopt programming models specific to the underlying distributed data processing engine with a steep learning curve involved and significant cost for development, debugging and evolution. As our query model, RAINBOW adopts StreamSight¹³ (initially designed by UCY) as its query model. This is a declarative SQL-like query language supporting the extraction of analytic insights relevant to descriptive statistics from IoT applications. The StreamSight query model has been significantly extended and its expressivity along with

¹³ <https://github.com/UCY-LINC-LAB/StreamSight>



various optimizations for fog environments are presented in the following chapter subsections.

- **API:** the component responsible for managing and authorizing access to the Fog Analytics Service functionalities. Users submit, remove or update the analytic queries through it.
- **Parser:** the component that forms the queries into respective Abstract Syntax Trees (ASTs). Each AST expresses the language's grammar rules, and the final level of the tree are the tokens and symbols of the query model. If no valid AST can be constructed from a query, the process stops and returns the suitable error message. Through this process, Parser guarantees the correctness of the submitted queries.
- **Optimizer:** is a subcomponent of the system that takes as input a set of ASTs and provides an optimized abstract plan to the Compiler. Specifically, Optimizer extracts similarities between the queries and prunes the unnecessary computations and data exchanges between the Fog nodes. Furthermore, Optimizer enforces policies and restructures the order of the operators to filter out useless data early.
- **Compiler:** takes as input the optimized plans from the Optimizer and generates the final executable artifact. To achieve this, the Compiler recursively traverses the optimized plans and "translates" each operator to the underlying distributed engine code. The generated code should be specific for each underlying engine, so during the translation process, Optimizer invokes the selected streaming library for the respected underlying distributed data processing engine.
- **Streaming Libraries:** As previously mentioned, a streaming library supports the Optimizer by implementing a general interface that materializes a set of required operators for the Compiler to be functional. The initial version of StreamSight only supported as the underlying programming model the basic operators for descriptive statistics made available by Spark Streaming. However, RAINBOW needs much more comprehensive operators, like pure streaming execution, optimizations that need collaboration with the scheduler of the data processing layer, fine-grained sampling and anomaly detection techniques, etc. To this end, we introduce a new RAINBOW-enabled library implemented to support Apache Storm as well that is compatible with RAINBOW's data processing layer and the needed functional requirements. It must be highlighted though, that any given analytics job, with its queries expressed with the StreamSight model, can be deployed without any changes to Spark Streaming, Storm and RAINBOW-enabled Storm clusters.



5.2.1 Query Model Expressivity

The RAINBOW query model offers users the ability to create *insights*, denoted as high-level queries composed from raw low-level metric monitoring streams. In a nutshell, an insight is a new data stream that comes from one (or more) processed stream(s). Query model operators introduce aggregations, compositions, and transformations on top of multiple monitoring metrics exposed by the input stream.

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

Figure 10: Insight Abstract Syntax

Figure 10 depicts the basic structure of an insight. The simplest insight structure includes 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 offers three optional primitives, namely, (i) **WHEN** primitive that filters a stream by applying specific predicates; (ii) **EVERY** primitive that alternates a purely streaming execution to a (micro-)batch query evaluation; and (iii) the **WITH** statement in which users define optimizations provided by the RAINBOW platform, such as sampling, prioritizing of the results, constraints enforcement, etc.

A *composite expression* can be a simple aggregation, e.g., an average over a stream, or could be recursively constructed via the left and right-hand composite expressions. The following queries illustrate some representative examples. For instance, the first insight generates the average energy consumption of the system for the last 10 minutes, which falls into the simple aggregation category, while the second and third insights compute the energy cost for the last ten minutes (the overall energy consumption multiplied by the cost of the unit (e.g., \$)0.002), and the energy consumption per CPU utilization unit, respectively. We should note here that our model includes many arithmetic symbols (such as +, -, *, /) and a set of arithmetic functions (window-based or accumulative). The fourth insight generates the RAM utilization per fog node. Specifically, when an insight includes the “**BY**” token, the query’s output is a list of pairs (e.g., <fog node id, value>) where the second element of each pair is the result of aggregated values and the first element is the distinct values of the “**BY**” attribute. In Table 11 and Table 12 there are all window-based and accumulative functions that are already created and provided from the model. The window-based functions need a window that denotes the past period of interest for aggregating values, while the accumulated functions generate the results computed solely based on all previous values.



energy_10min = **COMPUTE AVG**(energy_consumption, 10 **MINUTES**)

energy_plus_100_10min = **COMPUTE** 0.002 * **SUM**(energy_consumption, 10 **MINUTES**)

energy_per_cpu_15min = **COMPUTE AVG**(energy_consumption, 15 **MINUTES**) / **AVG**(cpu_util, 10 **MINUTES**)

ram_util_5min = **COMPUTE AVG**(ram_util, 5 **MINUTES**) **BY** fog_node_id

Table 11: Window-based model operators

Operator	Description
SUM	The sum of all values within a window
COUNT	The number of values within a window
MAX	The maximum value of the window
MIN	The minimum value of the window
AVG	The mean of all values within a window
TOP-K	The top k values from a window

Table 12: Accumulative-based model operators

Operator	Description
RUNNING_MEAN	The running average of all values in the stream
RUNNING_MAX	The max value of all values
RUNNING_MIN	The min value of all values
EWMA	The exponential weighted moving average
PEWMA	The probabilistic exponential weighted moving average

The **WHEN** statement represents a filter that can be attached to a stream or an expression so that left-hand operations are only processed if the filter predicate evaluates to true. A **WHEN** statement could be either a simple numeric value or even a more sophisticated composition that follows RAINBOW's model. Users are also capable of applying multiple filters by concatenating them with an **AND** logical operator. The following queries depict 2 exemplary insights including **WHEN** statement. Specifically, insight *energy_10min_over_100* will produce results only if the average energy



consumption of the last ten minutes exceeds 100 kilowatts. The *abnormal_temperature* will generate results only if the average environmental temperature of ten minutes exceeds the running average by three standard deviations.

```
energy_10min_over_100 = COMPUTE AVG(energy_consumption, 10 MINUTES) WHEN > 100
abnormal_temperature = COMPUTE AVG(temperature, 10 MINUTES) WHEN >
                        RUNNING_MEAN(temperature) +
                        3*RUNNING_STD(temperature)
```

Since some engines, like Apache Spark, do not provide pure-streaming execution and because the processing of grouped data into micro-batches could be much more efficient, RAINBOW's query language includes the **EVERY** statement. The **EVERY** construct denotes the interval at which an insight will be evaluated. Contrary to that, a pure streaming insight will generate a value for every newly entered datapoint into the stream as shown in the following examples:

```
energy_10min = COMPUTE AVG(energy_consumption, 10 MINUTES) EVERY 5 SECONDS

energy_plus_100_10min = COMPUTE 0.002 * SUM(energy_consumption, 10 MINUTES)
                        EVERY 5 SECONDS

energy_per_cpu_15min = COMPUTE AVG(energy_consumption, 15 MINUTES) / AVG(cpu_util, 10 MINUTES)
                        EVERY 15 SECONDS

ram_util_5min = COMPUTE AVG(ram_util, 5 MINUTES) BY fog_node_id EVERY 10
                SECONDS
```

Finally, the optional **WITH** statement allows users to define certain optimization strategies and constraints to improve runtime performance in the query execution. Users can define multiple optimizations for the same query by connecting them with an **AND** statement. The optimizations highlighted with the **WITH** statement are highly coupled with the RAINBOW Analytics Scheduler and are presented in the following section.

5.2.2 RAINBOW-Enabled Optimizations

RAINBOW's model offers users the ability to define fine-grained optimizations that can speed-up the execution or/and select execution policies. In the initial version of the system, we design seven optimizations that users can leverage in their queries. These optimizations are *Query Prioritization*, *Sampling*, *Sampling with Error Margin & Confidence*, *Adaptive Sampling*, *Outlier Detection*, *Scheduling Algorithm Selection*, and *Execution Placement*.



5.2.2.1 Query Prioritization

Prioritization allows users to order their queries based on their significance. For instance, a query that generates the maximum environmental temperature to ensure the fire-protection of the infrastructure is more critical than a simple statistic. RAINBOW's model offers users the keyword **SALIENCE** to define the order of their queries (higher **SALIENCE** means higher priority). Specifically, the value of **SALIENCE** can prioritize the query processing over other queries so as when a high load influx exists, high priority insights are not delayed. If a query does not provide the **SALIENCE** primitive, the system considers it equals to zero. The following examples illustrate the aforementioned fire-protection example, so when the processing engine cannot process both of the queries, it will generate results only for *abnormal_temperature*.

```
abnormal_temperature = COMPUTE MAX(temperature, 10 MINUTES) WHEN >70 WITH SALIENCE 5
```

```
cpu_util_20min = COMPUTE AVG(cpu_util, 20 MINUTES) EVERY 5 SECONDS
```

5.2.2.2 Sampling

Through sampling, users execute their insights on top of a portion of the incoming data to get approximate but in-time results. The user defines the sample size or percentage of remaining data points as the parameter of the query. RAINBOW utilizes reservoir sampling as the default sampling method. The core functionality of reservoir sampling, namely the probabilistic selection of a random fixed-size sample from an unknown size dataset, makes it one of the most popular streaming sampling techniques. In case of percentage definition (and not size of buffer), the system estimates the buffer's size during the execution and adjusts it properly.

```
ram_util_20_sample = COMPUTE AVG(ram_util, 10 MINUTES) WITH SAMPLE 0.2
```

5.2.2.3 Sampling with Error Margin & Confidence

Even if reservoir sampling is a popular approach in fog analytics, it has a certain limitation. Namely, in reservoir sampling, every data point is selected with equal probability. However, if various and heterogeneous sources produce the incoming stream, the latter may significantly change the sample's quality. To assure the statistical quality of the generated results, we introduce an operator that allows sampling with error confidence. With this operator, users are able to define the maximum acceptable error and the confidence of the results as shown in the following example. We implemented a combination of the reservoir and stratified sampling tailored to Edge and IoT streams, namely the Weighted Hierarchical Reservoir Sampling (WHRS) as introduced in [53]. Implementation-wise, each input operator acts independently and applies the reservoir sampling early on, right after measurements' injection. The



alteration of the traditional reservoir sampling is that each stream has a dynamically adjustable reservoir size depending on its runtime statistics. To generate these statistics, a weighting mechanism is applied, where the significance of each stratified reservoir is periodically updated. To this end, RAINBOW Analytics Service adjusts the significance of each stratified reservoir to comply with the user-defined boundaries, as described in the definition of the query.

```
network_io_bytes_sample = COMPUTE AVG(network_io_bytes, 10 MINUTES)  
                        WITH MAX_ERROR 0.05 AND CONFIDENCE 0.95
```

5.2.2.4 Adaptive Sampling

Adaptive sampling is a technique for dynamic adjustment of the sampling rate depending on the context of the streaming data. In this setting, the rate at which data is ingested by the data source (e.g., Spout in Storm) dynamically changes so that the collection periodicity is not statically defined (e.g., in contrast to the EVERY construct). For instance, during stable phases of a metric stream, the sampling rate is decreased to ease processing and data transfer. This optimization inherently adopts the adaptive sampling algorithmic process that will be introduced in the RAINBOW Monitoring. The following example introduces an example of utilizing adaptive sampling as an optimization to a CPU stream.

```
adaptive_sampling = COMPUTE MAX(cpu_pct, 1 MINUTES) WITH ADAM AND CONFIDENCE 0.95
```

5.2.2.5 Outlier Detection

There are many implementations of outliers' detection in streaming analytics. We selected to provide a set of distance-based outlier detection algorithms, as we introduced in [1]. A distance-based outlier detector considers as outliers all data points that have less than K -neighbours in a distance, denoted as R . In our first implementation, we consider only single dimension outlier detection. That dimension is the current stream value as described in the query definition from the user. Because the outlier detector in a streaming setting needs a window and a sliding interval, we created a new window operator named OUTLIER_DETECTOR that comes with an extra WITH statement in which the user selects a specific algorithm and its parameters. Next, we demonstrate an example of PMCOD algorithm [1] that has as parameters a window of ten minutes, five seconds sliding interval, takes into consideration fifty neighbours in range equals to 0.5.

```
disk_outliers = COMPUTE OUTLIER_DETECTION(disk_io_bytes, 10 MINUTES) EVERY 5 SECONDS  
              WITH ALGORITHM pmcod AND K=50 AND R=0.5
```



5.2.2.6 Scheduling Algorithm Selection

The selection of tailored optimization based on the scenario could be extremely beneficial for the end-user in Fog Computing analytics. Specifically, one may need different optimization strategies even between queries that are submitted together. As we described in Section 4.2.4, the RAINBOW data processing layer offers an extensible interface for implementation scheduling algorithms and a set of predefined schedulers. To this end, for the definition of the selected algorithm, RAINBOW's query model offers the SCHEDULER primitive. The SCHEDULER could be selected intendedly for every insight and at low-level dictates the operator's placement on Fog nodes. We should note here that the automated optimizations, like the reuse of intermediate results, will work over queries that will be handled by the same scheduler.

```
running_cpu_util = COMPUTE RUNNING_AVG(cpu_util) WITH SCHEDULER=<scheduler_name>
```

5.2.2.7 Execution Placement

Last but not least, through RAINBOW's query model one has the opportunity to specify a subset of cluster nodes on which his/her query will be executed. Especially, the PLACEMENT_ON statement specifies a set of parameters as hints to the RAINBOW's scheduler. The nodes metadata declares a set of metadata for the nodes capable of hosting the query. The latter metadata is known from the RAINBOW scheduler and is assigned to nodes from RAINBOW's platform. For instance, if the data, needed from a query, exists on a specific region (region-1), the user can define the "region_name=<region-1>" metadata property to place the execution of the query only on nodes existing in the same region to minimize the network traffic. Furthermore, there are two optional parameters, namely, the number of workers and the DEDICATED attribute. The first parameter specifies how many workers will host the query while if the DEDICATED exists, the orchestrator will dedicate a set of nodes that are allowed to run only that query.

```
node_network_io_10mins = COMPUTE SUM(network_io_bytes, 10 MINUTES)  
                        WHEN node_id == <fog_node_id>  
                        WITH PLACEMENT_ON region_name="region_1" 5 DEDICATED
```

5.2.3 Analytics Job Compilation Process

To generate our model, we use the EBNF standard which is a de-facto approach for defining new programming languages and query models. The initial EBNF description can be found in the deliverable appendix. Furthermore, we utilize the ANTLR parser¹⁴ in order to automatically produce the grammar and token classes. After that, we

¹⁴ <https://www.antlr.org/>

materialized and connected that classes with the **Parser** component to be compatible with the rest of the system.

During the submission of the queries, the **Parser** builds an Abstract Syntax Tree (AST) for each submitted query that follows the syntactic and grammatic rules of the RAINBOW query model (EBNF description). Figure 11 depicts the AST representation of the first exemplary query. In a nutshell, the leaves of the tree are the tokens and symbols of the query model while the tree structure obeys the grammar rules. Thus, an insight is syntactically valid if the system can generate the AST of the insight without any syntax error.

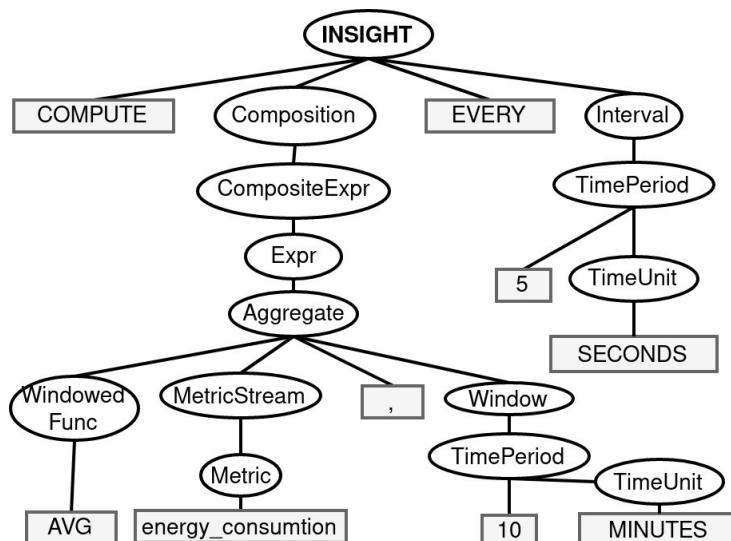


Figure 11: Exemplary Abstract Syntax Tree

Given an AST (or a set of ASTs), the Compiler is capable of translating them into low-level code of the underlying engine. Specifically, the Compiler recursively traverses the AST and automatically maps the grammar rules and tokens into streaming operators of the deployed engine. However, the programming models of different distributed engines may vary. For instance, Spark Streaming offers pipeline operators (e.g., map, reduce, filter) while Apache Storm uses purely user-defined graphs (DAGs). Furthermore, the engine itself may have specific limitations, e.g., Spark Streaming does not evaluate its operators continuously but splits the stream into micro-batches. Last but not least, there is no compatibility between different engines' source codes. In order to display the difficulty of distributed engine's code writing, Figure 12 depicts a query written in the RAINBOW query model and written by following Apache Storm programming model, respectively.



avg_metric_10mins = **COMPUTE AVG**(metric, 10 **MINUTES**)

```
public class SumBolt extends BaseWindowedBolt {
    OutputCollector collector;
    String field;
    private Map<String, Object> conf;
    //Current sum
    private double sum = 0;

    public SumBolt(String field){
        this.field = field;
    }

    @Override
    public void prepare(Map<String, Object> topoConf, TopologyContext
collector) {
        this.collector = collector;
        this.conf = topoConf;
    }

    @Override
    public void execute(TupleWindow inputWindow)
    /*
     * The inputWindow gives a view of
     * (a) all the events in the window
     * (b) events that expired since last activation of the window
     * (c) events that newly arrived since last activation of the window
     */
    List<Tuple> tuplesInWindow = inputWindow.get();
    List<Tuple> newTuples = inputWindow.getNew();
    List<Tuple> expiredTuples = inputWindow.getExpired();

    List<Tuple> tuples = inputWindow.get();
    if((boolean)this.conf.getDefault("ustreamsight.logs",false)){
        System.out.println("All:"+tuplesInWindow.size());
        System.out.println("New:"+newTuples.size());
        System.out.println("Exp:"+expiredTuples.size());
    }
    // Optimized SUM
    for (Tuple tuple : newTuples) {
        if(tuple.contains(this.field)) {
            sum += (double) tuple.getValueByField(this.field);
        }
    }
    for (Tuple tuple : expiredTuples) {
        if(tuple.contains(this.field)) {
            sum -= (double) tuple.getValueByField(this.field);
        }
    }
    collector.emit(tuples,new Values(sum));
}

@Override
public void declareOutputFields(OutputFieldsDeclarer declarer) {
    declarer.declare(new Fields(this.field));
}
```

Figure 12: Exemplary query adopting the RAINBOW query model vs the native Storm programming model

The same example needs over 15 lines of code in Apache Spark, completely different from Apache Storm. To alleviate this heterogeneity, and of course minimize the query's



definition effort, RAINBOW Analytic Service introduced an abstract interface that includes the definition of the minimum required methods for Compiler to be capable of model-to-code translation. The current version of the system includes two Streaming Libraries namely, an old proof-of-concept implementation of the Spark Streaming library that includes only the bare minimum functionality, and a newly created, more comprehensive and complete implementation on Apache Storm that includes all RAINBOW-enabled functionalities and is the default streaming library of the RAINBOW platform.

So, the compiler produces and joins data streams recursively until insight is built as an executable job. However, a naive translation of queries to underlying engine code could increase the processing delay, given that users submit multiple and related queries, and the processing engines, like Apache Storm or Spark, evaluate them separately as independent processes. For instance, the following examples illustrates two insights namely, the average energy consumption of the last ten minutes and the average energy consumption increased by 100.

`energy_10min = COMPUTE AVG(energy_consumption, 10 MINUTES) EVERY 5 SECONDS`

`energy_plus_100_10min = COMPUTE AVG(energy_consumption, 10 MINUTES) + 100 EVERY 5 SECONDS`

With the before-mentioned strategy, even if both streams feature the same 10min arithmetic mean from the same input stream, both pipelines must be recomputed. Reasonably, the re-computation of the same sub-query incurs a huge communication penalty, if the generated tasks are placed on different machines, but also computation overhead even if the tasks are collocated. The latter bottlenecks are significant for the Fog computing realms, where the processing power is restricted, or even occurs extra energy consumption, and the network bandwidth is valuable. To address this, our optimization process identifies identical queries or part of queries (same aggregation, input stream, interval, and optimization) between the submitted insights and combines them to generate an optimized execution plan. Figure 13 visualizes the ASTs of the aforementioned example. Understanding that the two insights have an identical aggregation, the optimizer will merge them to minimize the re-computation of the second's insights sub-tree (namely the red part of the tree). That optimized plan gives to the distributed engine the hidden semantic knowledge that is missing in low-level operations. With that optimized plan, the underlying engine will "cache" the results and speed up the overall execution.

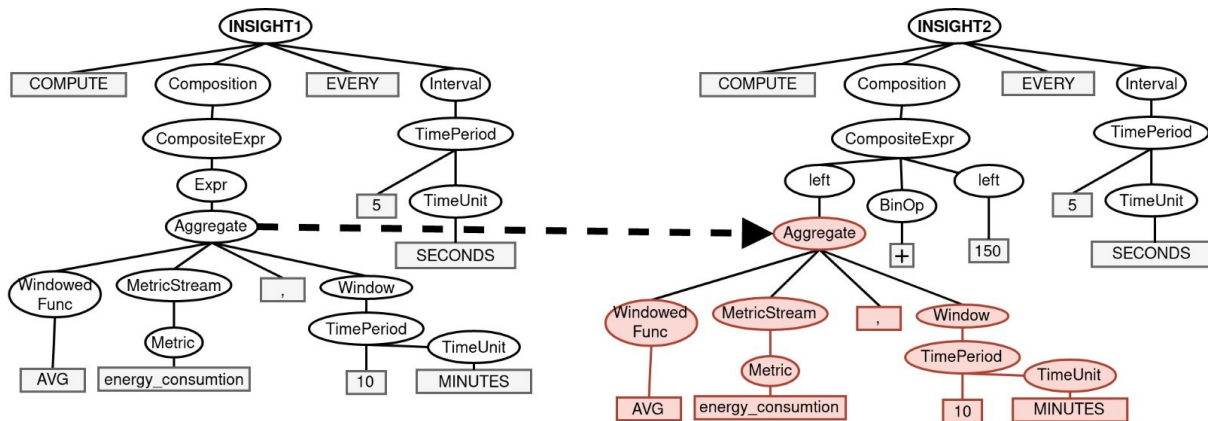


Figure 13: Queries reusing intermediate results to reduce unnecessary data computations

5.3 Interaction with other RAINBOW Services and Components

The RAINBOW Fog Analytics Service is “run” through the RAINBOW Dashboard and “sits” on top of the Distributed Data Processing Service with the goal of easing the definition and compilation of analytics jobs for users. To avoid repetition, we will omit referring to the interacting services, which are the RAINBOW Dashboard and the Distributed Data Processing Engine, both described in Section 4.3.

5.4 API and Documentation

As previously mentioned, the RAINBOW Fog Analytics Service “sits” on top of the Distributed Data Processing Service and therefore “shares” the API referenced in Table 8. In turn, the complete source code of the Fog Analytics Service can be found in the RAINBOW source code repository:

<https://gitlab.com/rainbow-project1/rainbow-analytics>



6 Conclusion

In this deliverable, we presented all necessary information regarding the implementation aspects of the RAINBOW ecosystem Data Management Services along with the major decisions about the realization technologies and tools. The Data Management Services contain three loosely coupled sub-components, namely, the Distributed Data Storage and Sharing Service, the Distributed Data Processing Service, and the Fog Analytics Service.

Initially we introduce a thorough report of the challenges that are related to data management when applied to Fog Computing infrastructure(s). Towards this, we extensively examined the current state-of-the-art technologies and the most recent advances in key technology axes. Having a clear overview of the state-of-the-art and the challenges of the topic, we proceed with the identification of the fine-grained functional and non-functional requirements for each component. The requirements led us to select specific tools, design the architecture, and introduce solutions tailored to the RAINBOW users' needs.

We chose Apache Ignite as the main realization technology for the Distributed Data Storage and Sharing Service since it can be seen as both persistent and in-memory distributed data storage. An extensive feature- and experimental-wise comparison among similar state-of-the-art systems indicated that Ignite behaves much more effectively in Fog Computing environments. Most importantly, Ignite allows us to implement data rebalancing algorithms without the need for re-engineering the whole system but only configuring thin clients on top of it. Finally, high-performance indexing schemas have been introduced to speed up the retrieval of particular information (latest data, historical data, and metadata) across the fog continuum.

The Distributed Data Processing service enables geo-distributed data processing by utilizing an open-source, scalable, and extensible distributed engine, namely Apache Storm, as the RAINBOW data processing engine. The main contribution of RAINBOW in the distributed processing is a handful of novel Fog-aware scheduling algorithms that will be implemented and be available by the end of the project. The early version of the system provides a baseline and a latency-optimized scheduling algorithm along with other three schedulers which are still in progress. Schedulers consider various aspects during the scheduling process such as data quality, energy consumption, and cost.

The Fog Analytics Service is the last component that is introduced by the current deliverable, and its role is to ease the definition of complex streaming analytic queries. To achieve that, Fog Analytic Service provides a high-level and declarative query model that abstracts the description of analytics from real-time monitoring data. Despite the



fact that the model is decoupled from the underlying engine, it promotes the declaration of fog-aware optimizations that RAINBOW schedulers can apply.

Finally, we provided extensive documentation on how every sub-component interacts with each other as well as with the other RAINBOW components.

7 References

- [1] T. Toliopoulos, C. Bellas, A. Gounaris, and A. Papadopoulos, “PROUD: PaRallel OUtlier Detection for Streams,” in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 2717–2720.
- [2] M. Symeonides, Z. Georgiou, D. Trihinas, G. Pallis, and M. D. Dikaiakos, “Fogify: A Fog Computing Emulation Framework,” in *2020 IEEE/ACM Symposium on Edge Computing (SEC)*, 2020, pp. 42–54.
- [3] M. Symeonides, Z. Georgiou, D. Trihinas, G. Pallis, and M. D. Dikaiakos, “Demo: Emulating Geo-Distributed Fog Services,” in *2020 IEEE/ACM Symposium on Edge Computing (SEC)*, 2020, pp. 187–189.
- [4] Z. Georgiou, C. Georgiou, G. Pallis, E. M. Schiller, and D. Trihinas, “A Self-stabilizing Control Plane for Fog Ecosystems,” in *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*, 2020, pp. 13–22.
- [5] A.-V. Michailidou, A. Gounaris, M. Symeonides, and D. Trihinas, “[Under-Submission] EQUALITY: Quality-Aware Intensive Analytics on the Edge,” *IEEE Trans. Big Data*, 2021.
- [6] R. Taft *et al.*, “CockroachDB: The Resilient Geo-Distributed {SQL} Database,” in *Proceedings of the 2020 International Conference on Management of Data, {SIGMOD} Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, 2020, pp. 1493–1509.
- [7] M. Serafini, E. Mansour, A. Abounaga, K. Salem, T. Rafiq, and U. F. Minhas, “Accordion: Elastic Scalability for Database Systems Supporting Distributed Transactions,” *Proc. {VLDB} Endow.*, vol. 7, no. 12, pp. 1035–1046, 2014.
- [8] Couchbase, “<https://www.couchbase.com/>.”
- [9] A. Lakshman and P. Malik, “Cassandra: A Decentralized Structured Storage System,” *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010.
- [10] “Amazon Aurora.” <https://aws.amazon.com/rds/aurora/>.
- [11] “ArangoDB.” <https://www.arangodb.com/>.
- [12] “Apache CouchDB.” <https://couchdb.apache.org/>.
- [13] “PostgreSQL.” <https://www.postgresql.org/>.
- [14] “Redis.” <https://redis.io/>.
- [15] “Riak.” <https://riak.com/>.
- [16] “Apache Cassandra.” <https://cassandra.apache.org/>.
- [17] “Apache HBase.” <https://hbase.apache.org/>.
- [18] “Apache Ignite.” <https://ignite.apache.org/>.
- [19] A. Adya *et al.*, “Slicer: Auto-Sharding for Datacenter Applications,” in *12th {USENIX} Symposium on Operating Systems Design and Implementation, {OSDI} 2016, Savannah, GA, USA, November 2-4, 2016*, 2016, pp. 739–753.
- [20] M. Serafini, R. Taft, A. J. Elmore, A. Pavlo, A. Abounaga, and M. Stonebraker, “Clay: Fine-Grained Adaptive Partitioning for General Database Schemas,” *Proc. {VLDB} Endow.*, vol. 10, no. 4, pp. 445–456, 2016.
- [21] R. Taft *et al.*, “E-Store: Fine-Grained Elastic Partitioning for Distributed Transaction Processing,” *Proc. {VLDB} Endow.*, vol. 8, no. 3, pp. 245–256, 2014.
- [22] “VoltDB.” <https://www.voltdb.com/>.



- [23] “Hazelcast IMDG.” <https://hazelcast.org/>.
- [24] “Amazon DynamoDB.” <https://aws.amazon.com/dynamodb/>.
- [25] “MongoDB.” <https://www.mongodb.com/>
- [26] “FaunaDB.”.
- [27] H. Herodotou, F. Dong, and S. Babu, “No one (cluster) size fits all,” *Proc. 2nd ACM Symp. Cloud Comput. - SOCC '11*, pp. 1–14, 2011.
- [28] Apache Spark, “[\url{http://spark.apache.org/}](http://spark.apache.org/).” 2017.
- [29] Flink, “Flink.” 2018.
- [30] “Apache Storm.”.
- [31] J. Dean and S. Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters,” *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.
- [32] K. Kloudas, M. Mamede, N. Pregoça, and R. Rodrigues, “Pixida: Optimizing Data Parallel Jobs in Wide-area Data Analytics,” *Proc. VLDB Endow.*, vol. 9, no. 2, pp. 72–83, Oct. 2015.
- [33] D. Trihinas, G. Pallis, and M. D. Dikaiakos, “Low-Cost Adaptive Monitoring Techniques for the Internet of Things,” *IEEE Trans. Serv. Comput.*, 2017.
- [34] M. D. D. Moysis Symeonides, Demetris Trihinas, Zacharias Georgiou, George Pallis, “Query-Driven Descriptive Analytics for IoT and Edge Computing,” *IEEE Int. Conf. Cloud Eng. (IEEE IC2E) 2019*, 2019.
- [35] J. Ren, H. Guo, C. Xu, and Y. Zhang, “Serving at the Edge: A Scalable IoT Architecture Based on Transparent Computing,” *IEEE Netw.*, vol. 31, no. 5, pp. 96–105, 2017.
- [36] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger, “Sonata: Query-Driven Streaming Network Telemetry,” in *Proceedings of SIGCOMM'18*, 2018.
- [37] Q. Pu *et al.*, “Low Latency Geo-distributed Data Analytics,” *SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, pp. 421–434, Aug. 2015.
- [38] C.-C. Hung, G. Ananthanarayanan, L. Golubchik, M. Yu, and M. Zhang, “Wide-Area Analytics with Multiple Resources,” in *Proceedings of the Thirteenth EuroSys Conference*, 2018.
- [39] Z. Hu, B. Li, and J. Luo, “Flutter: Scheduling tasks closer to data across geo-distributed datacenters,” in *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, 2016, pp. 1–9.
- [40] A. Vulimiri *et al.*, “WANalytics: Geo-Distributed Analytics for a Data Intensive World,” in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015, pp. 1087–1092.
- [41] K. Hsieh *et al.*, “Gaia: Geo-Distributed Machine Learning Approaching {LAN} Speeds,” in *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, 2017, pp. 629–647.
- [42] J. Xu, Z. Chen, J. Tang, and S. Su, “T-storm: Traffic-aware online scheduling in storm,” in *2014 IEEE 34th International Conference on Distributed Computing Systems*, 2014, pp. 535–544.
- [43] B. Peng, M. Hosseini, Z. Hong, R. Farivar, and R. Campbell, “R-Storm,” *Proc. 16th Annu. Middlew. Conf.*, Nov. 2015.
- [44] L. Eskandari, J. Mair, Z. Huang, and D. Eyers, “T3-Scheduler: A topology and Traffic aware two-level Scheduler for stream processing systems in a heterogeneous cluster,” *Futur. Gener. Comput. Syst.*, vol. 89, pp. 617–632, 2018.



- [45] X. Fu, T. Ghaffar, J. C. Davis, and D. Lee, “EdgeWise: A Better Stream Processing Engine for the Edge,” in *2019 {USENIX} Annual Technical Conference ({USENIX} {ATC} 19)*, 2019, pp. 929–946.
- [46] “Trident.”
- [47] M. Armbrust *et al.*, “Spark SQL: Relational Data Processing in Spark,” in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015, pp. 1383–1394.
- [48] M. Armbrust *et al.*, “Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark,” in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 601–613.
- [49] Apache, “Edgent.” 2019.
- [50] 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 International Conference on Utility and Cloud Computing (UCC 2018)*, 2018.
- [51] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, “Benchmarking cloud serving systems with YCSB,” in *Proceedings of the 1st ACM symposium on Cloud computing*, 2010, pp. 143–154.
- [52] P. Raj, “Chapter Seven - The Hadoop Ecosystem Technologies and Tools,” in *A Deep Dive into NoSQL Databases: The Use Cases and Applications*, vol. 109, P. Raj and G. C. Deka, Eds. Elsevier, 2018, pp. 279–320.
- [53] Z. Wen, D. Quoc, P. Bhatotia, R. Chen, and M. Lee, “ApproxIoT: Approximate Analytics for Edge Computing,” in *38th IEEE International Conference on Distributed Computing Systems (ICDCS 2018)*, 2018.



Appendix

EBNF Descriptive Query Model

```
// RAINBOW Model
statements      : statement (SEMI_COLON statement)* SEMI_COLON? EOF
                ;
statement       : streamDefinition #streamDefinitionExpr
                | insightDefinition #insightDefinitionExpr
                ;
streamDefinition : streamId COLON STREAM FROM streamProvider
                ;
streamId        : ID
                ;
streamProvider  : ID LPAR keyValueParams? RPAR
                ;
keyValueParams  : keyValuePair ( COMMA keyValuePair)*
                ;
keyValuePair    : ID EQUAL value
                ;
value           : STRING
                | INTEGER
                | FLOAT
                ;
insightDefinition : insightId EQUAL COMPUTE composition ;
insightId        : ID
                ;
composition      : expression #basecaseExpr
                ;
expression       : aggregate #aggregateExpr
                | metricStream #metricStreamExpr
                ;
metric           : STRING
                ;
membership       : LPAR member (COMMA member)* RPAR;
member           : ID;
aggregate        : windowedFunction LPAR metricStream COMMA window RPAR
                ;
metricStream     : metric (FROM membership)?
                ;
windowedFunction : ID;
window           : timeperiod
                ;
timeperiod       : INTEGER TIME_PERIOD;
```



```
// Tokens

COMPUTE: 'COMPUTE' | 'compute';
STREAM: 'STREAM' | 'stream';
FROM: 'FROM' | 'from';
TIME_PERIOD: 'MILLIS' | 'ms' | 'SECONDS' | 's' | 'MINUTES' | 'm' | 'HOURS' | 'h';
ID : [a-zA-Z_]+[a-zA-Z_0-9]*;
STREAM_PROVIDER: ID;

SEMI COLON: ';' ;
COLON: ':' ;
EQUAL: '=';
LPAR : '(';
RPAR : ')';
LBR: '[';
RBR: ']';

COMMA: ',';

STRING: '"' .*? '"';
INTEGER: '-'? [0-9]+ ;
FLOAT: ('+' | '-') ? [0-9] + ('.' [0-9] +)? ;
```