Knowledge Graph Stream Processing at the Edge

Joffrey de Oliveira
LIGM Univ Gustave Eiffel, CNRS, F-77454.
Engie Lab, CRIGEN
Marne la Vallée, France
joffrey.de-oliveira@univ-eiffel.fr

Christophe Callé
LIGM Univ Gustave Eiffel, CNRS, F-77454.
Engie Lab, CRIGEN
Marne la Vallée, France
christophe.calle@univ-eiffel.fr

Weiqin Xu
Engie Lab, CRIGEN
Stains, France
weiqin.xu@external.engie.com

Philippe Calvez
Engie Lab, CRIGEN
Stains, France
philippe.calvez1@engie.com

Olivier Curé
LIGM Univ Gustave Eiffel, CNRS, F-77454.
Marne la Vallée, France
olivier.cure@univ-eiffel.fr

ABSTRACT

We present a knowledge graph management system designed to run on Edge computing devices that handles high-frequency data streams. During the design phase, we took into account the inherent limitations of the devices, *i.e.*, limited computing power and storage space, as well as the expectations of applications, *e.g.*, low latency, high throughput, and intelligent data management. This results in a compact, decompression-free, in-memory, streaming-enabled RDF store that supports continuous querying and some forms of reasoning. The system addresses efficient query processing of data continuously arriving at a fast pace and is well-adapted to event-driven applications such as anomaly and risk detection. We empirically emphasize its accuracy, robustness, latency, and throughput properties on a real-world IoT setting originating from the energy management domain.

CCS CONCEPTS

• Information systems → Stream management; **• Applied computing → Event-driven architectures**; **• Computer systems organization → Distributed architectures**.

KEYWORDS

Stream processing, Knowledge graph, Edge Computing

ACM Reference Format:
Joffrey de Oliveira, Christophe Callé, Weiqin Xu, Philippe Calvez, and Olivier Curé. 2022. Knowledge Graph Stream Processing at the Edge. In *The 16th ACM International Conference on Distributed and Event-based Systems (DEBS '22), June 27–30, 2022, Copenhagen, Denmark.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3524860.3539644

1 INTRODUCTION

In recent years, there has been a strong interest for Edge computing[1]. This distributed computing paradigm complements Cloud computing by bringing the processing, management and storage of large data sets closer to where there are needed the most. The main benefits of this approach are to reduce response times and save network bandwidth by preventing data analysis from occurring in the cloud. These analyses typically support event-driven applications that include anomaly and risk detection in areas ranging from building, factory or vehicle monitoring to financial fraud.

Potentially, smart Edge computing, *i.e.*, involving reasoning services, can (i) facilitate the configuration of Internet Of Things (IoT) environments, (ii) deduce implicit consequences from the knowledge explicitly represented, (iii) provide background information and an explanation of inferences. To ensure such a behavior, a Knowledge Graph (KG) management system is required on Edge devices. Certain properties expected from such systems are to guarantee (i) a small memory footprint of the data management system as well as of the data and knowledge processed, and (ii) efficient query and inference processing.

Recently, we designed SuccinctEdge[17] which can be characterized as a federated event-based system (FEDS)[4] since it integrates and queries data pushed from multiples sensors into a SuccinctEdge client. Moreover, these clients push some data and metadata to a SuccinctEdge server when anomalies have been detected. The whole system can be defined as a compact, decompression-free, self-index, in-memory database management system for the Resource Description Framework (RDF) graph data model. We have met the aforementioned requirements through extensive use of succinct data structures (SDS) and specialized optimization algorithms. In this paper, we present extensions that allow the processing of unbounded data emitted from multiple sensors. By continuously querying an RDF data stream, our system is able to support event-driven applications requiring inferences.

To provide guarantees of low latency (the time between the start and end of an event) and high throughput (the total amount of work done in a given time), our extension relies on the open source Eclipse Mosquitto system. This lightweight message broker is suitable for Edge devices and offers a set of essential features for a stream processing platform. We can hence concentrate on tasks such as query processing and optimization, supporting several streaming models and window strategies and reasoning.

Concretely, our contributions consist of: (i) a solution to detect risks and anomalies from temporal data via the interrogation of unbounded RDF graphs, (ii) the support of different stream processing models, *i.e.*, true streaming and micro-batch, and window strategies, *i.e.*, tumbling and sliding, (iii) a rewrite of SuccinctEdge's query execution components: support for a continuous SPARQL extension, a new inference-enabled query optimization approach based on the distinction between static and dynamic portions of continuous queries, and (iv) a thorough evaluation of the accuracy, robustness, latency and throughput dimensions in a real-world scenario that is used at our energy partner.

Organization: in the next section, we present some background knowledge. In Section 3, we introduce a real-world motivating example. Section 4, provides an overview of our Streaming SuccinctEdge system. Section 5 presents some related work. We evaluate our system in Section 6, provide some lessons learned in Section 7 and conclude the paper in Section 8.

2 BACKGROUND KNOWLEDGE

2.1 RDF - SPARQL

RDF corresponds to a data model taking the form of a labelled, directed multi-graph. Assuming disjoint infinite sets I (IRI for Internationalized Resource Identifiers), B (blank nodes) and L (literals), a triple (S,P,O) ∈ (I ∪ B) x I x (I ∪ B ∪ L) is called an RDF triple where S, P and O respectively denote the subject, predicate and object of that triple.

SPARQL is a query language for RDF data. The SPARQL syntax follows the select-from-where approach of the SQL language. The `SELECT` clause specifies the (distinguished) variables appearing in the result set of the query. The `WHERE` contains a set of graph patterns that is called a basic graph pattern (BGP). Intuitively, SPARQL query processing amounts to returning the distinguished variable bindings that match the BGP to a given RDF graph. SPARQL processing is based on matching graph patterns. Additionally to sets I, B and L, the signature of a triple pattern (TP) in SPARQL requires V, an infinite set of variables. We can recursively define a SPARQL graph pattern as follows: (i) a triple gp ∈ (I ∪ B ∪ V) x (I ∪ V) x (I ∪ V ∪ B ∪ L) is a SPARQL graph pattern, (ii) if gp_1 and gp_2 are graph patterns, then ($gp_1 . gp_2$) represents a group of graph patterns that must all match, (gp_1 `UNION` gp_2), denoting pattern alternatives, are graph patterns and (iii) if gp is a graph pattern and C is a built-in condition then the expression (gp `FILTER` C) is a graph pattern that enables to restrict the solutions of a graph pattern match according to the expression C.

A Knowledge Base (KB) consists of an ontology, aka terminological box (TBox), and a fact base, aka assertional box (ABox). The least expressive ontology language of the Semantic Web is RDF Schema[1] (RDFS). Its goal is to provide a mechanism allowing to describe groups of related resources (concepts) and their relationships (properties). RDFS entailment can be computed using a set of (14) rules. But practical inferences can be computed with a subset of them. In this work, we consider the ρdf minimal RDF deductive system which has been defined and theoretically investigated in [12]. In a nutshell, ρdf considers inferences using `rdfs:subClassOf`, `rdfs:subPropertyOf`, `rdfs:range` and `rdfs:domain` properties. An RDF property is defined as a relation between subject and object resources. RDFS allows to describe this relations in terms of the classes of resources to which they apply by specifying the class of the subject (*i.e.*, the domain) and the class of the object (*i.e.*, the range) of the corresponding predicate. The corresponding `rdfs:range` and `rdfs:domain` properties allow to state that respectively the subject and the object of a given `rdf:Property` should be an instance of a given `rdfs:Class`. The property `rdfs:subClassOf` is used to state that a given class (*i.e.*, `rdfs:Class`) is a subclass of another class. Similarly, using the property `rdfs:subPropertyOf`, one can state that any pair of resources (*i.e.*, subject and object) related by a given property is also related by another property.

Other ontology languages, *e.g.*, OWL[2] (Web Ontology Language) of the Semantic Web stack are more expressive than RDFS and thus come at a higher computational cost for standard inference services.

2.2 Succinct Data Structures

In Streaming SuccinctEdge, we are using 2 types of Succinct Data Structures (SDS): BitMap (BM) and Wavelet Tree (WT)[13].

BM is the most basic type of SDS. It consists of a sequence of bits with extra information to support the execution of SDS operations. A WT uses 2 BMs at each level of its binary balanced tree. Both of them can be queried with a set of 3 operations: *Rank*, *Select* and *Access*. Given a sequence S, the operation $S.Access(i)$ (also denoted as $S[i]$) refers to the $(i+1)^{th}$ element in S. $S.Rank(i,c)$ returns the number of occurrences of c from S's beginning at index i. Finally, $S.Select(i,c)$ returns the index of the i^{th} occurrence of element c in S. Figure 1a provides an example of theses operations executed over a BM.

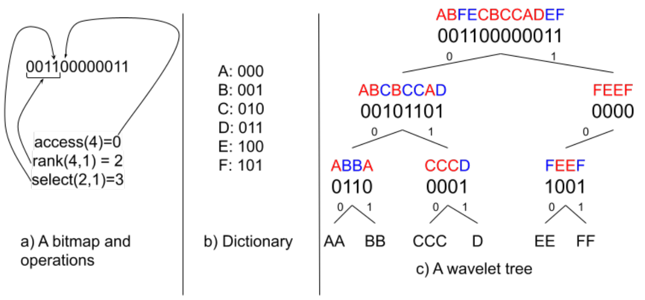

Figure 1: Wavelet Tree example with its dictionary

`Example 1`: Consider the *ABFECBCCADEF* string sequence, where each letter is mapped with an integer identifier in an incremental order,*e.g.*, *A* and *B* are resp. assigned values 0 and 1 (see dictionary in Figure 1b). Figure 1c displays the WT of this sequence according to this encoding. A tree structure is constructed from this sequence as follows: each level of this tree divides the sequence

[1]http://www.w3.org/TR/2014/REC-rdf-schema-20140225/Overview.html

[2]https://www.w3.org/TR/owl2-overview/

of previous nodes into two sub-sequences by the corresponding bit. For example, from root to the first level, *ABFECBCCADEF* is divided into *ABCBCCAD* and *FEEF* by the first bit of each identifier. This strategy is applied recursively until each leaf is computed. Using the RRR index[15], the SDS operations are computed in O(1) for BMs and O(log n) for WTs where n is the size of the vocabulary.

2.3 LiteMat

LiteMat is a semantic-aware encoding scheme that compresses RDF data sets and supports reasoning services associated to the RDFS ontology language. It focuses on the ρdf[11] subset of RDFS, *i.e.*, inferences associated to constructors such as `rdfs:subClassOf`, `rdfs:subPropertyOf`, `rdfs:domain`, `rdfs:range`. To address inferences drawn from these first two RDFS predicates, we attribute numerical identifiers to ontology terms, *i.e.*, concepts and predicates, that are supporting the semantic.

The semantic encoding of concepts and predicates supports reasoning services usually required at query processing time. For instance, consider a query asking for the pressure value of sensors of type S1. This would be expressed as the following two triple patterns: `?x pressureValue ?v. ?x type S1`. In the case sensor concept $S1$ has n sub-concepts, then a naive query reformulation requires to run the union of n+1 queries. With LiteMat's semantic-aware encoding, we are able, using two bit-shift operations and an addition, to compute the identifier interval, *i.e.*, [lowerBound, upperBound), of all direct and indirect sub-concepts of $S1$. And thus we can compute this query with a simple reformulation: (i) replacing the concept S1 with a new variable : `?x type ?newVar` and (ii) introducing a filter clause constraining values of this new variable: `FILTER (?newVar>=lowerBound && ?newVar<upperBound)`.

2.4 Eclipse Mosquitto

Eclipse Mosquitto[10] is a lightweight message broker that implements MQTT (Message Queuing Telemetry Transport), a publish-subscribe message protocol that generally uses the Internet protocol suite,*i.e.*, TCP/IP. Mosquitto has been designed to run on devices with limited resources such as sensors, *i.e.*, handling over 1000 clients on less than 3MB of RAM. Therefore, it is particularly well suited for Edge computing. In the context of our Streaming SuccinctEdge platform, Mosquitto supports the data exchange between SuccinctEdge's server and client as well as between SuccinctEdge client and sensors (see Figure 2).

3 MOTIVATING EXAMPLE

3.1 Data flow

Our running example is based on a frequent use case in sensor-based anomaly detection. In fact, it corresponds to a real world scenario encountered by our energy partner: ENGIE, a multinational company operating in fields such as energy transition, generation and distribution.

The setting of this use case corresponds to buildings where hundreds of sensors are installed and are continuously capturing measures. Sensor data is continuously analyzed to detect anomalies and/or identify risky situations, *e.g.*, gas and water leaks, energy over-consumption.

Figure 2 presents the typical data flow of this IoT setting. We consider the installation of a new sensor in the building (step 1). In step 2, the people responsible for this installation, noted the IoT Persons, declare to the administrators of the platform the schema associated with the measurements retrieved from this device as well as certain metadata, *e.g.*, the brand of the sensor, the location, the date of the last calibration. Note that this approach also applies when an existing sensor is replaced. Therefore, we consider that it is not possible for a sensor to be changed without the administrators being aware of it.

The Administrators then ask a team of domain experts to map this schema, *e.g.*, CSV, to a semantic representation, *e.g.*, OWL. A large set of ontologies are available to annotate the IoT and sensor domains, *e.g.*, Sensor, Observation, Sample, Actuator (SOSA[3]), Quantities, Units, Dimensions, and Types (QUDT[4]) or Smart Appliances Reference (SAREF[5]). The semantic representation of these data is an incentive to use the RDF data model. At ENGIE, the use of these ontologies simplified the task of describing, manipulating and connecting sensors and actuators. Domain experts also define relevant queries (using SPARQL) , *i.e.*, those enabling anomaly and risk detection (step 3).

Using LiteMat[6], administrators can encode ontologies required by this new graph (step 4) and transform the SPARQL queries into optimized queries expressed in terms of SDS operations (step 5). These physical query plans are sent to the appropriate SuccinctEdge client (step 6). These queries are continuously executed when receiving messages from the sensors (step 7). These queries can only be modified upon Administrators request, *e.g.*, when the connected sensor is changed. In case an anomaly is detected by a query, a message is sent from the SuccinctEdge client to the SuccinctEdge server with some context information such as device and query identifiers, abnormal data and event time of the anomaly (step 8). In order to perform several of these tasks, SuccinctEdge maintains some metadata, *e.g.*, query/sensor, client instance/sensor, mosquitto/client instance associations.

3.2 Semantic integration

Figure 3 presents an extract of a graph processed by a Streaming SuccinctEdge instance. It contains some measures related to the distribution of some commodities, *e.g.*, water, gas, in a building. Given such a graph, our system executes queries that can detect some anomaly patterns, *e.g.*, distribution network leaks.

In our experimentation at ENGIE, we found out that several types and brands of sensors are frequently being used to observe similar measures, *e.g.*, pressure, flow. These sensors may also produce measures expressed in different units, *e.g.*, Bar, $Pascal$, psi, $Torr$ for pressure measures; ft^3/min, gal/min, m/min, m^3/h for volume per time unit. In this context, it is necessary to integrate all retrieved information into a single information system.

The ability to associate KG concepts and properties to the measures produced by these sensors is the first step toward this semantic integration. Moreover, domain experts generally define concepts of observable properties, *e.g.*, AtmosphericTemperature, which can be

[3]http://www.w3.org/TR/ns/sosa
[4]http://qudt.org/schema/qudt
[5]https://ontology.tno.nl/saref.ttl

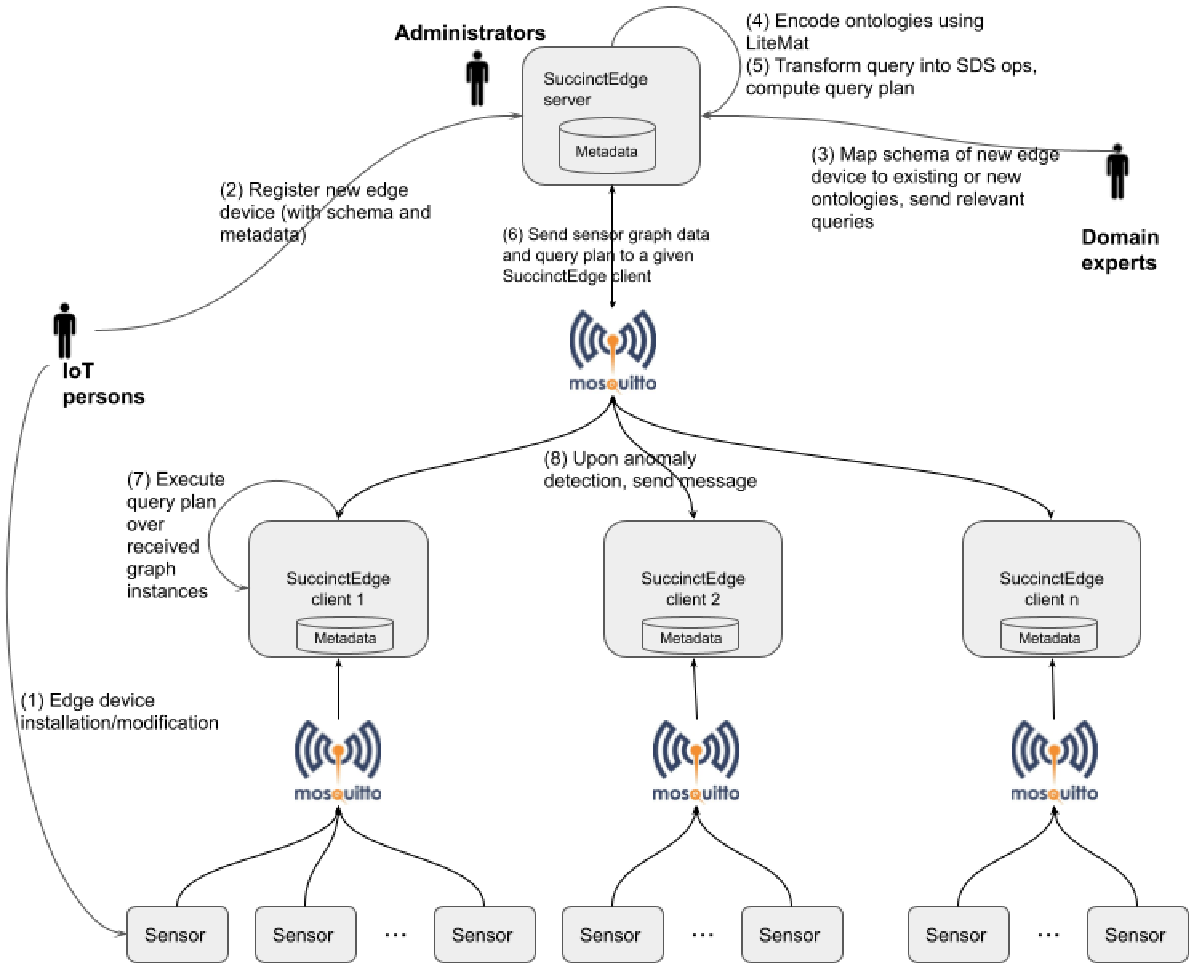
Administrators
SuccinctEdge server
Metadata
(4) Encode ontologies using LiteMat
(5) Transform query into SDS ops, compute query plan
(3) Map schema of new edge device to existing or new ontologies, send relevant queries
Domain experts
(2) Register new edge device (with schema and metadata)
(6) Send sensor graph data and query plan to a given SuccinctEdge client
IoT persons
mosquitto
(7) Execute query plan over received graph instances
(8) Upon anomaly detection, send message
SuccinctEdge client 1
SuccinctEdge client 2
SuccinctEdge client n
(1) Edge device installation/modification
Sensor
...

Figure 2: Data flow for setting a Streaming SuccinctEdge platform

organized into hierarchies and can hence be used for reasoning purposes. RDFS inferences are efficiently processed in SuccinctEdge, via query rewriting, thanks to our LiteMat encoding approach.

A second semantic integration step consists in making it easier to write SPARQL queries by automatically transforming queries to the characteristics of a given sensor, *e.g.*, based on concept annotations and units being used. To support these requirements, we encourage domain experts to express queries in relatively high concept terms. Expressing a query with abstract concepts, *i.e.*, high in the concept hierarchy, permits to write a single query that can tackle sensors performing similar measures but annotated with different concepts and measure units. Hence, domain experts do not have to worry about the inferences which are handled automatically by the system. This approach's simplicity was expected at ENGIE for productivity reasons. Indeed, it allows its sensor personnel to concentrate on their tasks and not on adapting a given query to a potentially large number of sensors in an industrial environment.

Let us consider two sensor platforms. The first station corresponds to the one described in Figure 3 where the pressure is typed as $qudt : PressureOrStressUnit$ and is expressed in the Bar unit. In the second one, a similar pressure measure is typed as $qudt : PressureUnit$ and is expressed in the HectoPA unit. Since, the QUDT ontology[6] states that: $qudt : PressureOrStressUnit \sqsubseteq qudt : PressureUnit$, a single SPARQL query (detailed in Section 4.2) can be written to address the specificity of each sensor at these stations.

4 STREAMING SUCCINCTEDGE OVERVIEW

4.1 Architecture

SuccinctEdge adopts a self-index approach based on the PSO permutation. This means that a single copy of RDF triples is stored in the system. Given our PSO indexing approach, we distinguish between datatype properties, *i.e.*, where the object is a literal, and object properties, *i.e.*, where the object is not a literal. In most use cases we have encountered, the relationships between instances in the RDF graphs rarely change because they represent the connections between physical objects, *e.g.*, platforms, sensors. We can thus represent all triples containing an object property (except *rdf:type*

[6]https://qudt.org/

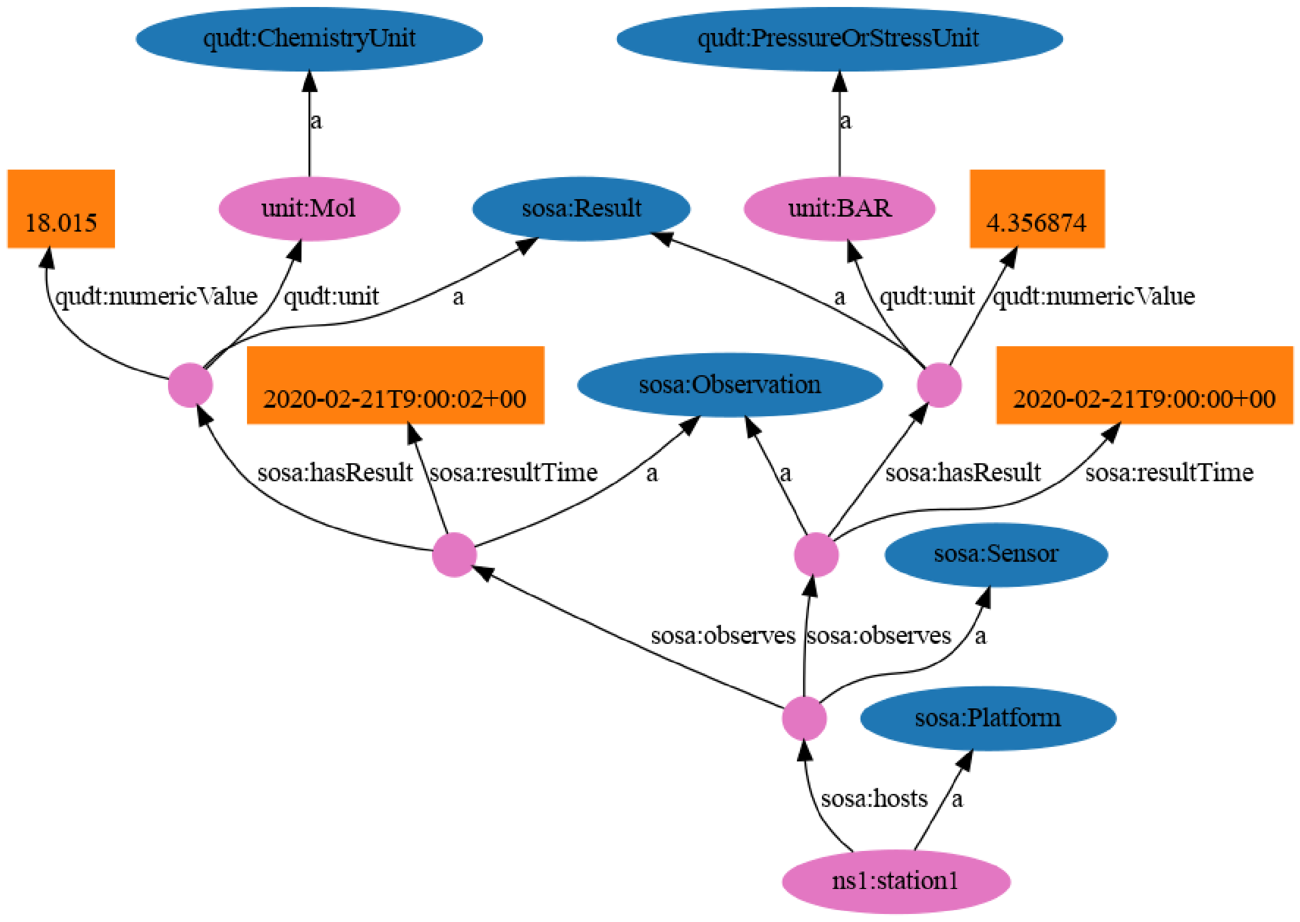

Figure 3: Graph extract of our use-case

for which a special storage is proposed) with a combination of WTs and BMs. Intuitively, each predicate, subject, object set is stored as a WT (resp. WT_p, WT_s and WT_o) and two BMs resp. connect the WT_p to WT_s and WT_s to WT_o.

Meanwhile, the data generated by sensors, *i.e.*, numerical measures, which are objects of some datatype properties, change continually. A high update rate is not adapted to a WT storage for the two following reasons: (i) each object value would require a single identifier but this is not reasonable since these values are mostly numerical and thus possibly infinite, (ii) updating a WT can not be performed efficiently.

Figure 4 shows the details of the data structures used for triples containing a datatype property. Properties and subjects are stored as in the object properties part, *i.e.*, with 2 WTs and a BM. A BM connects the subject WT to an object layer, denoted O. In this layer, each object represents a dynamic data which is timestamped and frequently appended. There, the system stores a pointer for each object which is pointed to a queue-like structure. When new data comes in, we push it to the front of the queue. These queue-like structures have some auxiliary functions to optimize the aggregation operations (*i.e.*, MIN, MAX, AVG, SUM, COUNT) present in a query. The corresponding function is activated on demand from the system, *e.g.*, it may depend on the streaming semantic (see next sub-section). When we execute a triple pattern (TP) with datatype property, we search the index interval of objects using the WTs and BMs of the first two layers, then for each object in this interval, we take the value in its corresponding data queue and compute the function indicated in the query.

`Example 2:` We consider the data transfer of 2 different sensors (S1 and S2) is given in Figure 4. We assume that each sensor measures a single value, *i.e.*, S1V and S2V. S1T (resp. S2T) represents the timestamp received from S1 (resp. S2) and S1V (resp. S2V) refers to its S1's measure (resp. S2). Hence, SuccinctEdge has distributed a queue-like structure for each data series.

Even though sensors can send messages with different frequencies, SuccintEdge can still handle the situation thanks to a map structure implementation to distribute each datatype object to a set of its corresponding sensor. By using this approach, we easily keep all the data sequences originating from one sensor in the same data contiguous structure. Moreover, sliding and tumbling streaming windows impose the maintenance of cursors on these structures.

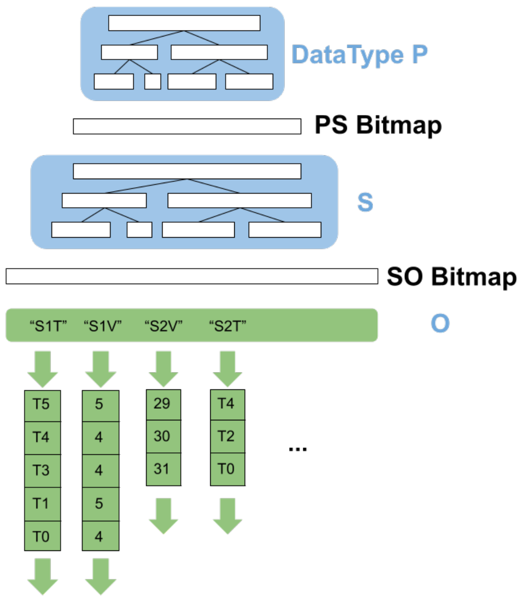

Figure 4: Data type property structure with two WTs and two BMs

4.2 Continuous SPARQL extension

Several projects have extended the SPARQL query language to support the continuous querying of RDF data streams. As a well-established approach, the syntax and semantics of C-SPARQL [3] has influenced our own SPARQL extension.

Compared to the C-SPARQL syntax, we are currently only supporting logical windows, *i.e.*, time-based, and our RANGE description block, specifying the length of the logical window, appears in the SELECT clause.

In the context of our experiments at ENGIE, we have not encountered use cases where a physical window, *i.e.*, based on counting triples, is needed. Moreover, we have not yet experienced a situation where named graphs are required. So, the FROM clause is at the moment not supported in our SPARQL extension. Nevertheless, we are aiming to address this limitation in the next version of SuccinctEdge. In addition to tumbling windows, we also support sliding windows, see Section 4.4, using the STEP keyword to specify the frequency of window sliding. We can also associate an aggregation operation to each query variable. Another extension we have made is that for each datatype variable, we can indicate an aggregation function we want to compute, *e.g.*, MAX, MIN and AVG.

The query presented in Figure 5 detects anomalies related to an incorrect pressure value (either expressed in Bar or HectoPascal) for sensors at stations 1 and 2. We can see that in the RANGE clause a tumbling window of 5 seconds is required. Moreover, the numerical variable ?v1 is followed by a [MAX] instruction which indicates that for each binding of ?v1 in the result set, we take the maximum in the data window. The FILTER clause detects anomalies, the BIND clause performs some data transformations (between the Bar and hectoPascal measure units).

```
SELECT ?x ?s ?ts ?newV[Max]
[RANGE 5000ms TUMBLING] WHERE {
?x rdf:type sosa:Platform; sosa:hosts ?s.
?s rdf:type sosa:Sensor; sosa:observes ?o.
?o rdf:type sosa:Observation;
sosa:resultTime ?ts; sosa:hasResult ?y.
?y rdf:type sosa:Result; qudt:unit ?u,
qudt:numericValue ?v.
?u rdf:type qudt:PressureOrStressUnit.
FILTER (?newV<3.00 || ?newV>4.50)
BIND(if(regex(str(?u),
"http://qudt.org/vocab/unit/BAR"),?v,
if(regex(str(?u),
"http://qudt.org/vocab/unit/HectoPA"),
?v/1000,0)) as ?newV)}
```

Figure 5: Physical pressure anomaly detection query

Such queries are continuously executed on some SuccinctEdge clients. Whenever this query returns a result, its distinguished variables, *i.e.*, variable binding of the SELECT clause, are sent to a SuccinctEdge server to alert on a potential anomaly. Note that the message to the server is also enriched with client metadata such as SuccinctEdge client and query identifications.

4.3 Query processing

The query processor described in [17] has been extended with a decomposition of a query's basic graph pattern (BGP). The motivation for this new query processor is two-fold and based on an observation of real-world IoT settings.

First, continuous queries analyzing streaming data are (i) generally highly selective, *i.e.*, returning rather small answer sets of around tens of tuples, and (ii) retrieve both static, *e.g.*, sensor and entity identifiers, and dynamic, *e.g.*, analysis of recent measures and their timestamps, information. Moreover, a sensor can produce a set of measures corresponding to different types of information, *e.g.*, pressure, flow, pH, etc. But most of the time, only one value is produced per type of information in a given sensor output. Note that this has also been observed in other industrial contexts, for example the Waves project [7].

Second, the semantic graph, *i.e.*, its TBox, associated to a sensor rarely changes except if the sensor is replaced. As stated in Section 3.1, the replacement of a sensor in our IoT ecosystem is necessarily notified to the team of administrators of the overall platform. Then, this forces to re-execute steps 3 to 6 of Figure 2 and thus prevents our semantic context to be become incoherent with our IoT setting.

Given these two observations, our new query processor distinguishes between a static and a dynamic subset of the BGP. Intuitively, when a sensor produces its first measures, the complete BGP of the continuous query is executed and the bindings of the distinguished variables of the static part of the BGP are cached by SuccinctEdge. Note that this query execution may involve some form of (RDFS) reasoning which are handled by LiteMat's rewriting facility. This rewriting mainly tackles the concept and property hierarchy inferences.

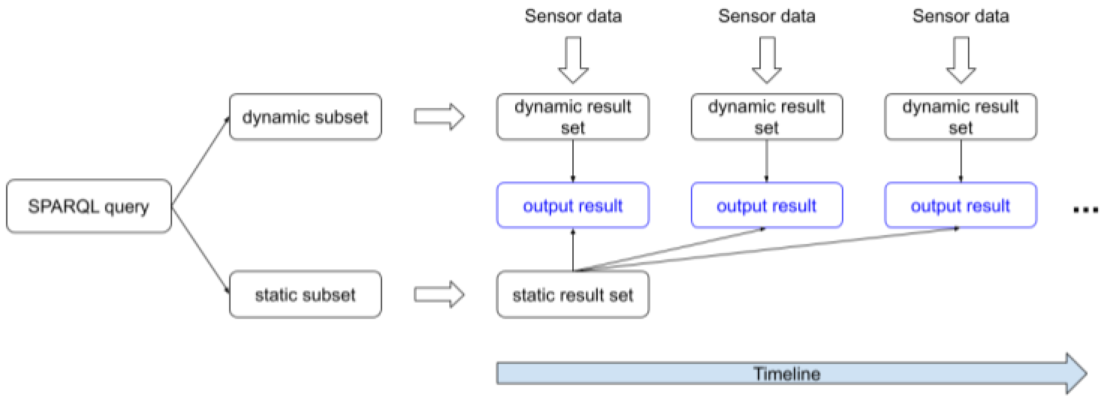

Figure 6: Static and dynamic query processing

Then, for successive measures produced by this same sensor, only the dynamic portion of the BGP needs to be executed and integrated in the query result set (see Figure 6). The dynamic distinguished variables correspond to objects of datatype properties and thus do not influence a query's graph pattern matching since they correspond to graph leaves. The execution of this dynamic BGP subset may require the computation of aggregation functions and some data transformation, *e.g.*, transform a pressure from Bar to Pascal.

The main design principle of our query processing component is to take advantage of this aspect and to compute a physical plan only once for a given query. This drastically improves query execution since in a continuous query processing setting, a query may be computed an undefined number of times. This approach is reminiscent to a parameterized query, *aka* prepared statement, where a query is pre-compiled and only needs some parameters to complete its execution. In our streaming context, the parameters correspond to the dynamic part of the BGP, *i.e.*, the objects associated to datatype properties (including timestamps or measures) or the result of applying an aggregation function over them. Since the static subset of the BGP is cached to optimize it's execution, we need to reassemble the two parts when an anomaly is found before its reporting. This is performed by searching through a series of hash map data structures where the key corresponds to a query and sensor identification pair.

In the query of Figure 5, the cached static part corresponds to the ?x and ?s variables, respectively the platform and sensor URIs, while the dynamic part corresponds to the ?ts and ?v1[MAX] variables, respectively the event timestamp and maximum pressure value of a pressure measure whose maximum value exceeds a certain threshold.

We have seen in Figure 2 that the computation of the physical plan is performed at a SuccinctEdge server which has generally more resources, in terms of CPU and memory, than an Edge device where a SuccinctEdge client is running, *e.g.*, a Raspberry Pi. SuccinctEdge's query optimizer mixes heuristics with a cost-based approach. The statistics of the latter are stored in the dictionaries

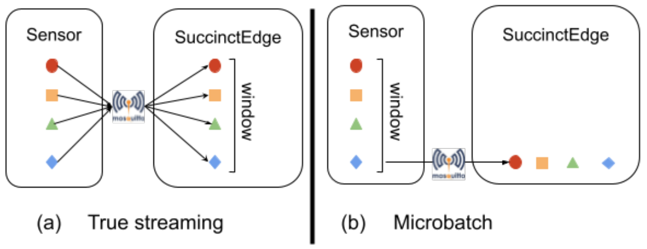

Figure 7: Supported exchange modes

of LiteMat which remain on the machine running the SuccinctEdge server. The remaining of the query evaluation is performed on a SuccinctEdge client.

4.4 Data stream exchange modes

Streaming SuccinctEdge supports two data stream exchange modes. In Figure 7, data events are represented as shapes. In the true streaming mode, each sensor is immediately sent to its SuccinctEdge client, via Mosquitto. The main advantage here is to limit data latency. But, it comes with low data throughput and high network cost due to frequent data exchange between the sensor and Mosquitto.

In the micro-batch mode, the sensor retains a certain amount of events, typically corresponding to the length of the query temporal window. Once the boundary of this window is attained, the complete set of data is sent to the SuccinctEdge client, also via Mosquitto. This mode limits the number of data exchanged over the network but it increases data latency.

These two modes can run under the sliding and tumbling window strategies which are both supported by SuccinctEdge. Both of these window strategies have a fixed duration and a slide interval. In the case of tumbling widows, the slide interval is equal to the fixed duration while in sliding windows the slide interval is different to the fixed duration. When the slide interval is inferior to the fixed duration, data contained in one window has some overlap with the previous window. SuccinctEdge provides some optimizations to intelligently manage this overlap. In the context of our experimentation at ENGIE, we have not encountered situation where having a slide interval superior to the fixed duration was relevant. In fact this last situation implies that some sensor data are not being processed.

5 RELATED WORK

RDF4Led[16] is an RDF database system that has been specifically designed for Edge computing. It adopts a data persistence approach, *i.e.*, all the data are stored on a SD card. This is partially motivated by a query execution component based on multiple indexes. Apart from being less efficient in terms of query execution and memory footprint compared to SuccinctEdge (see [17] for details), RDF4Led does not support reasoning services and certain SPARQL clauses that would enable an efficient query rewriting, *e.g.*, UNION clause. Additionally, RDF4Led is not supplied with stream processing capabilities. It is thus not able to execute queries over a set of incoming RDF graphs.

Fed4Edge [14] is a decentralized version of the CQELS engine that benefits from the work conducted on RDF4Led. It adopts the CQELS-QL [8] continuous query language based on SPARQL, while we are using C-SPARQL. Nevertheless, in terms of query processing and streaming features, both systems are quite comparable. Currently, Fed4Edge concentrates on SPARQL query federation [2] while Streaming SuccinctEdge has so far focused on query optimization and reasoning. Nevertheless, as a FEDS system, SuccinctEdge is also able to integrate and query data emitted by different sensors.

WaterFowl[5] is an RDF store based on SDS that is capable of performing RDFS reasoning services. It can hence provide useful services at the Edge of computing network. Nevertheless, WaterFowl lacks support for stream processing, misses the object storage design and the query processing optimization of Streaming SuccinctEdge. Finally, WaterFowl is not equipped with streaming functionalities, *e.g.*, continuous SPARQL querying.

Existing centralized RDF Stream Processing systems like C-SPARQL [3] and CQELS [9] have been around for some time. Each engine proposes its own continuous query language extension (generally based on the SPARQL syntax) to query time-annotated triples. So, they have influenced our own extension of SPARQL to continuously process queries. Though, none of these systems have been designed to run on an Edge computing device.

6 EVALUATION

In this section, we evaluate our Streaming SuccinctEdge system along the following dimensions: accuracy, robustness, scalability, query processing performance, latency and throughput. This evaluation is performed over different settings (tumbling as well as sliding windows, anomaly scenarios) in the context of both synthetic data and a real-life use case at one of ENGIE's building.

6.1 Experimental setting

The context of this evaluation is anchored in our running example (Section 3.1) where real-life measures are analyzed in a building of our research pattern. For confidentiality reasons, we can not detail the complete building setting. Nevertheless, we can state that each floor of this four storey building is equipped with a SuccinctEdge client connected to eight sensors. These sensors can capture different measures, *e.g.*, room temperature, physical pressure. The set of sensors was heterogeneous at the time of the experimentation, *i.e.*, some sensors were emitting pressures in the Bar unit while other ones were submitting values in HectoPascal. A Single SuccinctEdge server is receiving all messages from the four SuccinctEdge clients.

Since, we could not force anomalies in this real-world context, it was mainly used to assess the robustness of our streaming SuccinctEdge prototype. Concerning accuracy, query execution performance, latency and throughput, we needed to control the occurrence of anomalies. For this reason, we also processed some synthetic data characterizing different forms of anomalies.

The devices running SuccinctEdge clients are Raspberry Pi 3B+, *i.e.*, equipped with a Cortex-A53 (ARMv7l) 32-bit SoC 1.4GHz CPU and 1GB LPDDR2 SDRAM. SuccinctEdge is implemented in C++ (version 14) and uses the SDS-lite library[7]. Eclipse Mosquitto (version 2.0) and SuccinctEdge server run using a JDK (version 8). The server is also using the Eclipse Paho java library [8] (an MQTT client library). Data is transferred using MQTT which is a publish / subscribe protocol based on TCP/IP. In every scenario examples, we were using a Wi-Fi network. Installation details can be found on github[9].

In our use case, normal measures belong to the [4,5) interval (in Bar) . A large set of evaluated dimensions are experimented over 5 scenarios which are presented in Figure 8. They differ in the occurrence frequency of anomalies. Intuitively, there is no anomaly in Scenario 0 since all pressure values are in the [4,5) interval. Few anomalies are regularly occurring in Scenario 1, series of anomalies are followed by correct measures in Scenario 2. In scenarios 3 and 4, measures are drifting to a continuous anomaly states (complete pressure loss in Scenario 4).

6.2 System robustness, accuracy and client scalability

Considering robustness, we have conducted our tests over the building setting previously described. This evaluation was conducted for a week without any failure from our SuccinctEdge platform, *i.e.*, client, server and mosquitto instances.

During that week, we witnessed five anomalies. We double checked the data retrieved for that week with domain experts. They confirmed that only five anomalies occurred during that period. In terms accuracy, *i.e.*, whether the system detects all anomalies that were occurring during our experimentation and whether it does not detect false anomalies, SuccinctEdge identified the five anomalies and only those anomalies. In fact, as long as the anomaly detection query is correct, there is no reason for our system to detect a false anomaly or to miss one.

Of course, an incorrect calibration of a sensor can falsify our anomaly detection, but we consider that this is not the responsibility of the SuccinctEdge system. We are planning to enrich the functionality of our SuccinctEdge server to analyze historical measures in order to identify calibration issues following data drifting.

[7]https://github.com/simongog/sdsl-lite
[8]https://www.eclipse.org/paho/index.php?page=clients/java/index.php
[9]https://github.com/SuccinctEdge/SuccinctEdgePublic

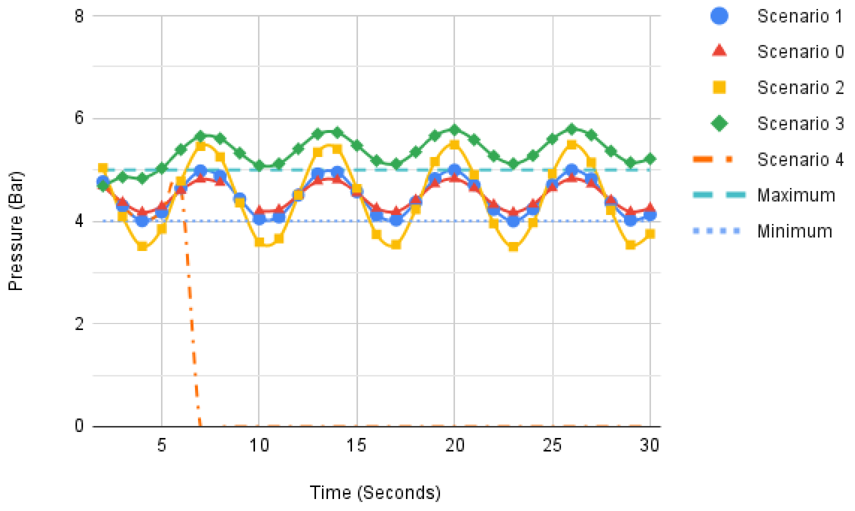

Figure 8: Experimentation scenarios on pressure measures

Another problematic situation occurs if the sensor is not able to send measures to Mosquitto or if the Mosquitto instance between a sensor and a SuccinctEdge client is down. We are currently implementing fault tolerance in SuccinctEdge to at least identify these situations. There already exists a heartbeat solution in MQTT which allows to know if the client or the server has lost connection to the Mosquitto Broker. However it does not allows us to monitor the status of our clients and servers.

We have evaluated our system with synthetic data to make sure that SuccinctEdge detects different anomaly patterns, *i.e.*, the ones from our 5 scenarios presented in Figure 8. We ran different window strategies (tumbling and sliding windows), stream communication modes (true streaming and micro-batch) and windows from 1 to 60 seconds. In all cases, we detected all anomalies and only real anomalies.

The scenarios have also been tested over a setting implying several sensors communicating with a single Streaming SuccinctEdge client, *i.e.*, up to 40, and with different frequencies, *i.e.*, with two sets of 20 sensors sending messages respectively every 200 and 300ms. The same 100% correctness has been observed. Finally, we evaluated a 40 sensors platform under sliding windows (from 5 sec to 5 min steps), true streaming (5 min to 1 hour) for over 3 days with no failure and the same accuracy.

6.3 Throughput and network usage

We begin by evaluating, on synthetic data, the impact of data exchanges over the network in an experimentation spanning over 24 hours in relatively extreme cases: an anomaly per half hour (Table 1) and an anomaly every 2 hours (Table 2).

While sending a Mosquitto packet, we only consider the size of the header and the payload size. The payload size is a multiple of 16 bytes (8 bytes for the value and 8 for the timestamp). We compare different modes of SuccinctEdge: true streaming and micro-batch (henceforth batch) using anomaly detection for each mode. Some anomaly detections in SuccinctEdge are implemented using the FILTER clause of SPARQL, hence it can be disabled if there is no FILTER in the query. Then all the sensors measures are sent directly to a SuccinctEdge server.

When in batch mode, SuccinctEdge only sends the relevant data once to the server for a given time. When the anomaly detection mode is enabled, the client only sends back measurements identified as anomalies by the query. We try multiple batch sizes: sending batches every hour or every 2 hours. Each batch correspond to a single Mosquitto packet assuming the payload size is below 256MB. Sensors are also producing one measurement per second.

In Tables 1 and 2, we see the network load to transfer 24 hour sensor payload from clients to server using different modes. In Table 1, we detect one anomaly per half hour, while in Table 2, we have one anomaly every 2 hours.

Obviously, we get a higher throughput when data is transferred as a batch and when we have more than one anomaly in the time frame of this batch. The streaming mode is a little less efficient, however it allows to get the data instantaneously.

When examining the latency and throughput properties, we do not consider the query execution that retrieves and caches the static part of the query as this is amortized by our query processing approach. Instead, we only consider the impact of receiving successive measurements, *i.e.*, the computation of aggregate functions, the detection of anomalies and the integration of the dynamic and static parts of the query.

6.4 SuccinctEdge client query run time

In this section, we are evaluating the impact of adding many sensors on a single SuccinctEdge client. Each sensor is sending 100 measures every second to the client, the evaluation has been conducted over a range of 1 to 40 sensors. Figure 9 highlights that for 100 measures per second, a SuccinctEdge client is able to handle up to 10 sensors without delay in it's query processing. It also shows that it could

Mode	Batch size	Data size	Mode	Batch size	Data size
Batch	1 hour	1.38MB	Batch	2 hours	1.38MB
Batch + detection	1 hour	240B	Batch + detection	2 hours	216B
Streaming	-	1.46MB	Streaming	-	1.46MB
Streaming + detection	-	288B	Streaming + detection	-	288B

Table 1: Network usage for 1 sensor, having 1 anomaly per half hour (detection means that we are sending the query result set to the SuccinctEdge server)

Mode	Batch size	Data size	Mode	Batch size	Data size
Batch	1 hour	1.38MB	Batch	2 hours	1.38MB
Batch + detection	1 hour	216B	Batch + detection	2 hours	216B
Streaming	-	1.46MB	Streaming	-	1.46MB
Streaming + detection	-	216B	Streaming + detection	-	216B

Table 2: Network usage for 1 sensor, having 1 anomaly per 2 hours (detection means that we are sending the query result set to the SuccinctEdge server)

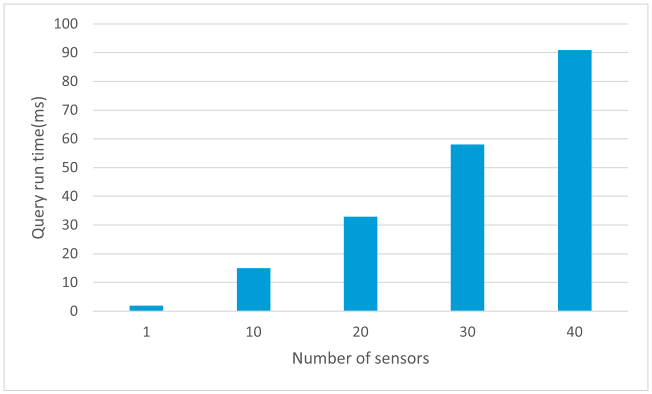

Figure 9: Average query run time (dynamic portion) in true streaming for a client over 1 to 40 sensors sending 100 measures per second

handle up to 40 sensors if they were only sending 10 measures per second. Therefore, for a setting where each sensor sends a measure every 10ms, a new SuccinctEdge client is needed every 10 or so sensors. Considering the cost of a Raspberry Pi, this is not a limitation of our overall streaming solution. Moreover, in the context of industrial anomaly detection expected at companies like ENGIE, a measure frequency in the range of seconds is more realistic than in milliseconds.

These results are only possible due to the optimization of BGP processing which is performed in two steps. The results in Figure 9 represent the execution time of the dynamic subset of the BGP while the execution time of the static subset is shown in Table 3, varying based on the number of triple patterns contained in the query BGP. We can see that the dynamic portion of the query processing (in ms) dominates the static one (sub ms).

6.5 SuccinctEdge server robustness and latency

In this section, we are evaluating the impact of adding many clients on a single SuccinctEdge server. As we didn't have enough Raspberry Pis for this synthetic data-based experimentation, we had to create a docker[10] image of the SuccinctEdge client to run it in multiple containers to stress test the SuccinctEdge server. In this scenario, each client has 5 sensors, sending data every second. These values are always considered as anomaly and are sent to the server.

We measure the latency for a SuccinctEdge client-server roundtrip, having the server returning an acknowledgement to the client for each value received. Latency is increasing linearly when adding client to a single server as we can see in figure 10. In this particular experimentation setting, beyond 10 clients, the server is too slow to process data in time, so the latency is increasing over time. Note that this experimentation context is rather extreme since it considers that all data received by a client is an anomaly. Hence, in a real setting, it is best to question the domain experts for average and worst case anomaly frequencies in order to define the number of clients per server. Note that in our ENGIE building setting, a SuccinctEdge server was connected to four clients.

Finally, in terms of latency measurement, we consider that the dockerization also has a non negligible negative impact.

7 LESSONS LEARNED

Running our experimentation on a real-life anomaly detection - driven use case helped us to confirm several accepted ideas. First, the latency imposed by a micro-batch solution is sufficient in most cases. In fact, the lower latency provided by a streaming approach is not really expected by the ENGIE staff: gaining few seconds is not that important compared to the time required by a human intervention to fix a problem, *e.g.*, fixing a leak in a distribution network.

The efficient setting of the number of sensors connected to a single client and of the number of clients connected to a single

[10]https://www.docker.com/

Number of triple patterns	5	8	10	15
Time (ms)	0.215	0.312	0.414	0.632

Table 3: Time used to process the static part of the query for different number of triple patterns

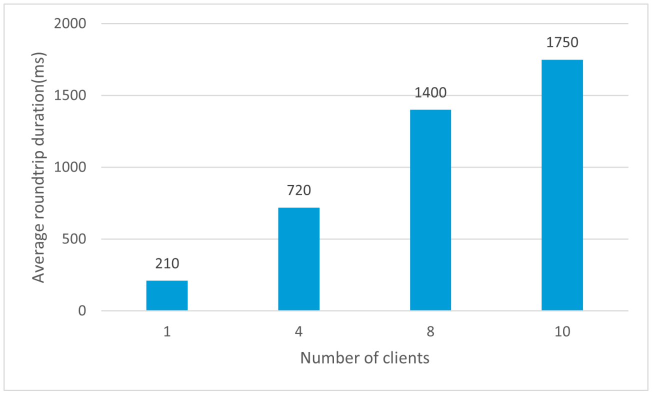

Figure 10: SuccinctEdge client-server average round-trip latency for client number ranging from 1 to 10

server require the knowledge of domain experts. Through our experimentation, we already discovered that up to 40 sensors can be connected to a single SuccinctEdge client and that in extreme cases, up to 10 clients can be connected to a single server. Note that this is more than what an energy actor like ENGIE is currently expecting.

Additionally over 24 hours of measurement, our system find many similar values in our micro-batches, this is not very important when sending the data in true streaming mode but when we accumulate this data in micro-batches it means that it can be compressed efficiently to allow for persistence of the data even with the limited hard drive capacity of edge devices. This could be achieved in future works by the integration of Apache Parquet[11], a columnar storage format focusing on saving storage space, in Streaming SuccinctEdge.

We also got the confirmation that most queries submitted in an anomaly detection context are quite selective and the answer set of low cardinalities. At ENGIE, we are searching for use cases requiring queries with a lower selectivity.

Finally, we got the confirmation that timed-based windows (as opposed to counting or session windows) are more relevant in risk/anomaly detection. Additionally, we have yet to find a scenario where sliding windows would make more sense than tumbling windows when running on an Edge device.

8 CONCLUSION AND FUTURE WORK

In this paper, we have presented an original attempt to process RDF stream at the edge of a computing infrastructure. The main characteristics of our system are compactness and the ability to infer implicit consequences on-the-fly. Our system, denoted Streaming SuccinctEdge, demonstrates relevant properties, *e.g.*, low latency and high throughput, expected in a Big Data context where data are produced at a high velocity. A thorough experimentation over the most frequently used streaming models (true streaming and micro-batch) and window strategies (sliding and tumbling) emphasized the accuracy, robustness and scalability of the system. In the near future, Streaming SuccinctEdge will execute in several buildings at our energy partner. Moreover, communication and cooperation across SuccinctEdge clients will be integrated.

[11]https://parquet.apache.org/

REFERENCES

[1] Y. Ai, M. Peng, and K. Zhang. Edge computing technologies for internet of things: a primer. *Digital Communications and Networks*, 4(2):77 – 86, 2018.
[2] C. B. Aranda, M. Arenas, Ó. Corcho, and A. Polleres. Federating queries in SPARQL 1.1: Syntax, semantics and evaluation. *J. Web Semant.*, 18(1):1–17, 2013.
[3] D. F. Barbieri, D. Braga, S. Ceri, E. D. Valle, and M. Grossniklaus. Querying DDF streams with C-SPARQL. *SIGMOD Rec.*, 39(1):20–26, Sept. 2010.
[4] K. M. Chandy and M. Olson. Specifications and architectures of federated event-driven systems. In *Intelligent Event Processing, Papers from the 2009 AAAI Spring Symposium, Technical Report SS-09-05, Stanford, California, USA, March 23-25, 2009*, pages 21–26. AAAI, 2009.
[5] O. Curé, G. Blin, D. Revuz, and D. C. Faye. Waterfowl: A compact, self-indexed and inference-enabled immutable RDF store. In *ESWC*, pages 302–316, 2014.
[6] O. Curé, W. Xu, H. Naacke, and P. Calvez. LiteMat, an encoding scheme with RDFS++ and multiple inheritance support. In *The Semantic Web: ESWC 2019 Satellite Events - Revised Selected Papers*, pages 269–284, 2019.
[7] H. Khrouf, B. Belabbess, L. Bihanic, G. Képéklian, and O. Curé. WAVES: big data platform for real-time RDF stream processing. In D. Dell'Aglio, E. D. Valle, T. Eiter, M. Krötzsch, M. Maleshkova, R. Verborgh, F. M. Facca, and M. Mrissa, editors, *Joint Proceedings of the 3rd Stream Reasoning (SR 2016) and the 1st Semantic Web Technologies for the Internet of Things (SWIT 2016) workshops co-located with 15th International Semantic Web Conference (ISWC 2016), Kobe, Japan, October 17th - to - 18th, 2016*, volume 1783 of *CEUR Workshop Proceedings*, pages 37–48. CEUR-WS.org, 2016.
[8] D. Le-phuoc, M. Dao-tran, J. X. Parreira, and M. Hauswirth. A native and adaptive approach for unified processing of linked streams and linked data. In *ISWC 2011*, pages 370–388. Springer, 2011.
[9] D. Le-Phuoc, M. Dao-Tran, J. X. Parreira, and M. Hauswirth. A native and adaptive approach for unified processing of linked streams and linked data. In *ISWC*, 2011.
[10] R. A. Light. Mosquitto: server and client implementation of the MQTT protocol. *The Journal of Open Source Software*, 2(13):265, 2017.
[11] S. Muñoz, J. Pérez, and C. Gutierrez. Simple and efficient minimal RDFS. *Web Semant.*, 7(3):220–234, Sept. 2009.
[12] S. Muñoz, J. Pérez, and C. Gutierrez. Minimal deductive systems for RDF. In E. Franconi, M. Kifer, and W. May, editors, *The Semantic Web: Research and Applications*, pages 53–67, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.
[13] G. Navarro. Wavelet trees for all. *J. Discrete Algorithms*, 25:2–20, 2014.
[14] M. Nguyen Duc, A. L. Tuán, J. Calbimonte, M. Hauswirth, and D. L. Phuoc. Autonomous RDF stream processing for iot edge devices. In *Semantic Technology - 9th Joint International Conference, JIST 2019*, pages 304–319. Springer, 2019.
[15] R. Raman, V. Raman, and S. R. Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Trans. Algorithms*, 3(4):43–es, Nov. 2007.
[16] A. L. Tuán, C. Hayes, M. Wylot, and D. L. Phuoc. RDF4Led: an RDF engine for lightweight edge devices. In *IOT*, pages 2:1–2:8, 2018.
[17] W. Xu, O. Curé, and P. Calvez. Knowledge graph management on the edge. In *EDBT*, pages 229–240, 2021.