Autonomous RDF Stream Processing For IoT Edge Devices

Manh Nguyen-Duc[1], Anh Le-Tuan[1,3], Jean-Paul Calbimonte[4], Manfred Hauswirth[1,2], and Danh Le-Phuoc[1]

[1] Open Distributed Systems, TU Berlin, Germany
[2] Fraunhofer Institute for Open Communication Systems, Berlin, Germany
[3] Insight Centre for Data Analytics, NUI Galway, Ireland
[4] University of Applied Sciences and Arts Western Switzerland HES-SO, Sierre, Switzerland

Abstract. The wide adoption of increasingly cheap and computationally powerful single-board computers, has triggered the emergence of new paradigms for collaborative data processing among IoT devices. Motivated by the billions of ARM chips having been shipped as IoT gateways so far, our paper proposes a novel continuous federation approach that uses RDF Stream Processing (RSP) engines as autonomous processing agents. These agents can coordinate their resources to distribute processing pipelines by delegating partial workloads to their peers via subscribing continuous queries. Our empirical study in "cooperative sensing" scenarios with resourceful experiments on a cluster of Raspberry Pi nodes shows that the scalability can be significantly improved by adding more autonomous agents to a network of edge devices on demand. The findings open several new interesting follow-up research challenges in enabling semantic interoperability for the edge computing paradigm.

Keywords: Autonomous systems, stream processing, cooperative sensing, query federation

1 Introduction

Over the last few years, Semantic Web technologies have provided promising solutions for achieving semantic interoperability in the IoT (Internet of Things) domain. Ranging from ontologies for describing streams and devices [10, 11], to continuous query processors and stream reasoning agents [8], these efforts constitute important milestones towards the integration of heterogeneous IoT platforms and applications. While these different technologies enable the publication of streams using semantic technologies (e.g., RDF streams), and the querying of streaming data over ontological representations, most of them tend to centralise the processing, relegating interactions among IoT devices simply to data transmission. This approach may be convenient in certain scenarios where the streams, typically time-annotated RDF data, are integrated following a top-down approach, for instance using cloud-based solutions for RDF Stream Processing

(RSP). However, in the context of IoT, decentralised integration paradigms fit better with the distributed nature of autonomous deployments of smart devices [22]. Moreover, moving the computation closer to the edge networks, such as sensor nodes or IoT gateways, will not only create more chances to improve performance and to reduce network overhead/bottlenecks, but also to enable flexible and continuous integration of new IoT devices/data sources [19].

Thanks to recent developments in the design of embedded devices, e.g., ARM boards [23], single board computers are getting cheaper and smaller while increasing their computational power. For example, a Raspberry computer costs less than 40 EUR and its size is just roughly as big as the size of a credit card. Despite the size, they are powerful enough to run a fully-functioning Linux distribution that provides both *operational* and *deployment* advantages. On the one hand, they are both power efficient and cost-effective, while computationally powerful. On the other hand, their small sizes make it easier to embed or bundle them with other IoT devices (e.g., sensors and actuators) as a processing gateway interfacing with outer networks, called *edge devices*.

RDF Stream Processing (RSP) [21] extends the RDF data model, enabling to capture and to process heterogeneous streaming sensor sources under a unified data model. An RSP engine usually supports a continuous query language based on SPARQL, e.g. C-SPARQL [3] and CQELS-QL [15]. Hence, an edge device equipped with an RSP engine could play the role of an autonomous data processing gateway. Such an autonomous gateway can coordinate the actions with other peers connected to it to execute a data processing pipe in a collaborative fashion. However, to the best of our knowledge, there has not been any in-depth study on how such a decentralised processing paradigm would work with edge devices. In particular, an edge device has 10-100 times less resources than those of a PC counter-part which is originally the expected execution setting for an RSP engine. Hence, this raises two main questions: how feasible would it be to enable such a paradigm for edge devices, and how it would affect the performance and scalability. Putting our motivation in the context of 100 billion ARM chips that have been shipped so far [4], enabling computational and processing autonomy along with semantic interoperability will have a huge impact even for a small fraction of this number of devices (e.g. 0.1% would account for 100s millions devices).

To this end, this paper investigates how to realise this edge computing paradigm by extending an RSP engine (i.e., CQELS) as a continuous query federation engine to enable a decentralised computation architecture for edge devices. A prototype engine was implemented to empirically study the performance and scalability aspects on "cooperative sensing" scenarios. Our experiment results on a realistic setup with the biggest network of its kind in Section 4 show that our federation engine can considerably scale the processing throughput of a network of edge devices by adding more nodes on demand. We believe this is the largest experiment setup of its kind so far. The main contributions of the paper are summarised below:

1. We propose a novel federation mechanism based on autonomous RSP Engines and distributed continuous queries.

2. We present technical details on how to realise such a federation mechanism by integrating an RSP engine and an RDF Store for edge devices.
3. We carry out an empirical study on performance and scalability on "cooperative sensing" that leads to various quantitative findings and then opens up several interesting follow-up research challenges.

The paper is outlined as follows. The next section presents our approach on continuous federation based on autonomous RSP. Section 3 describes the implementation details of our federated RSP engine for edge devices. The setup and results of the experiments is reported in Section 4. We summarise related work in Section 5 and Section 6 concludes the paper.

2 Continuous Federation with Autonomous RSP

2.1 Preliminaries: RDF Stream Processing with CQELS-QL

CQELS-QL is a continuous query language for RSP that extends SPARQL 1.1 with sliding windows [15]. As an example, the CQELS-QL query below continuously provides the "updates for the locations of 10 weather stations which have reported the highest temperatures in the last 5 minutes". This query then is also used as Query Q3 in the experiments of Section 4.

```
1  SELECT ?sensor ?maxTemp ?lat ?lon
2  WHERE {
3      {SELECT ?sensor (MAX(?temp) as ?maxTemp)
4         STREAM ?streamURI [RANGE 5m ON sosa:resultTime]{
5            ?observation sosa:hasSimpleResult ?temp.
6            ?sensor rdf:type <TempSensor>; made:Observation ?observation.}
7            GROUP BY ?sensor}
8      ?streamURI prov:wasGeneratedBy ?sensor. ?sensor sosa:isHostedBy ?station.
9      ?station wgs84:Point ?loc. ?loc wgs84:lat ?lat;wgs84:lon ?lon.}
10 ORDER BY ?maxTemp
11 LIMIT 10
```

Listing 1: Query Q3 in CQELS-QL (prefixes are omitted)

In the original centralised setting, the above query can be subscribed to a CQELS engine installed in one *processing node*. Stream data in RDF formats (e.g JSON-LD or Turtle) can be provided to it from data acquisition nodes, called *streaming nodes*. These streaming nodes collect data from sensors of weather stations that can be geographically distributed in different locations. In practice, an edge device can host both a CQELS node and a streaming node, but we can assume they communicate via an internal process. As soon as the data is collected, the sensor data is pushed to the CQELS engine via a streaming protocol such as Websocket or MQTT. The incoming data continuously triggers the processing pipeline compiled from Query Q3. Consequently, the computing node that hosts this CQELS engine needs to have enough resources (bandwidth, CPU and memory) to deal with the workload regardless of how many stream nodes exist in the network. Hence, if the CQELS engine is only hosted on an edge device, the physical limits of its hardware quickly becomes a bottleneck

as shown in Section 4. To create a more scalable processing system, we need to decentralise the processing pipelines of similar queries to a network of edge devices connected to these stream nodes. The following two sections describe our approach to enable this type of network.

2.2 Dynamic subscription & discovery for autonomous RSP engines

To enable a CQELS engine to work in a decentralised fashion, it would require the capability to operate as an autonomous agent which can collaborate with other peers to execute a distributed processing pipeline specified in CQELS-QL. An autonomous CQELS node can dynamically join a network of existing peers by subscribing itself to an existing node in the network, called a parent node, and it then notifies the parent node about the query service and streaming service it can provide to the network. These services can be semantically described by using vocabularies provided by VoCaLS [27]. For instance, VoCaLS allows describing the URIs of the streams and their related metadata (e.g. sensors that generated the streams), which are used in the query patterns of the query Q3. Hence, a subscription can be done by sending a RDF-based message via a REST API or Websocket channel. Listing 2 illustrates a snippet of a subscription message in RDF Turtle that is used in our experiments in Section 4.

```
1 <> a vocals:StreamDescriptor, vsd:CatalogService; dcat:dataset :NOAAWeather.
2 :NOAAWeather a vocals:RDFStream; prov:wasGeneratedBy :TemperatureSensor;
3 vocals:hasEndpoint :NOAAWeatherEndpoint; dct:title "Weather stream From Berlin".
4 :NOAAWeatherEndpoint a vocals:StreamEndpoint; dct:format frmt:JSON-LD;
5 dcat:accessURL "ws://192.168.178.5/noaa/berlin".
```

Listing 2: Example of subscription message in RDF

Based on the semantic description provided by the subscribed nodes, the parent node can carry the stream discovery patterns which use a variable in the stream pattern, as shown in line 4 of the query Q3. The variable *?streamURI* then can be matched in other metadata as shown in line 8. In this example, it is used to link with the sensors that generated this stream. Recursively, the subscription process can propagate the stream information upstream hierarchically, and vice versa, the discovery process can be recursively delegated to downstream nodes via sub-queries in CQELS-QL.

To this end, when an autonomous CQELS joins a network, it makes itself and its connected nodes discoverable and queryable to other nodes of the network. Moreover, each node can share its processing resources by executing a CQELS query on it. This will help us treat the query similar to query Q3 as a query to a sensor network whereby sensor nodes and network gateways collaborate as a single system to answer the query of this kind. Next, we will discuss how to federate such queries in “cooperative sensing” scenarios whereby such a network of autonomous CQELS processing nodes will coordinate each other to answer a CQELS-QL request in a decentralised fashion.

2.3 Continuous Query Federation Mechanism

With the support of the above subscription and discovery operations, a stream processing pipeline written in CQELS-QL can be deployed across several sites distributed in different locations: e.g., weather stations provide environmental sensory streams in various locations on earth. Each autonomous CQELS node gives access to data streams fed from streaming nodes connecting to it. Such stream nodes can ingest a range of sensors, such as air temperature, humidity and carbon monoxide. When the stream data arrives, this CQELS node can partially process the data at its processing site, and then forward the results as mappings or RDF stream elements to its parent node.

In this context, when a query is subscribed to the top-most node, called root node, it will divide this query to sub-query fragments and deploy at one or more sites via its subscribed nodes. A query fragment consists of one or more operators, and each fragment of the same query can be deployed on different processing nodes. Recursively, a sub-query delegated to a node can be federated to its subscribed nodes. All participant nodes of a processing pipeline can synchronise their processing timeline via a timing stream that is propagated from the root node. The execution process of sub-query fragments can use resources, i.e. CPU, memory, disk space and network bandwidth of participant nodes to process incoming RDF graphs or sets of solution mappings and generate output RDF graphs/sets of solution mappings. Output streams may be further processed by fragments of the same query, until results are sent to the query issuer at the root node. For example, the sub-query of the query Q3 in below Listing 3 can be sent down to the nodes closer to the streaming nodes, then the results will be recursively sent to upper nodes to carry out the partial top-k queries in lines 10 and 11 until it reaches the root node to carry out final computation steps to return the expected results.

```
1  SELECT ?sensor (MAX(?temp) as ?maxTemp)
2  WHERE{
3       STREAM ?streamURI [RANGE 5m ON sosa:resultTime]
4       { ?observation sosa:hasSimpleResult ?temp.
5         ?sensor rdf:type <TempSensor>; made:Observation ?observation.}
6       ?streamURI prov:wasGeneratedBy ?sensor.
7  }GROUP BY ?sensor
```

Listing 3: An Example of The Subquery of Q3

This federation process can be carried out dynamically thanks to the dynamic subscription and discovery capability above. Moreover, the processing topology of such as processing pipelines in our experiment scenarios of Section 4 can be dynamically configured by changing where and how participant nodes subscribed themselves to the processing networks. For example, we carried out five different federation topologies in Section 4. The biggest advantage of this federation mechanism is the ability to dynamically push some processing operations closer to the streaming nodes to alleviate the network and processing bottlenecks which often happen at edge devices. Moreover, this mechanism can significantly improve the

processing throughput by adding more processing nodes on demand as shown in the experiments in Section 4.

3 Design and Implementation

To enable the cooperative federation of RSP engines on edge devices, we built a decentralised version of the CQELS engine, called Fed4Edge. Fed4Edge was implemented by extending the algorithms and Java codebase of the original open sourced version of CQELS [15]. Thanks to the platform-agnostic design of its execution framework [14], the core components are abstract enough to be seamlessly integrated with different RDF libraries in order to port to different hardware platforms. To tailor the RDF-based data processing operations on edge devices (e.g ARM CPU, Flash-storage and the likes), we integrated the core components of CQELS with the counterparts of RDF4Led [17], a RISC style RDF engine for lightweight edge devices. The Fed4Edge system will be open-sourced at `https://github.com/cqels/Fed4Edge`.

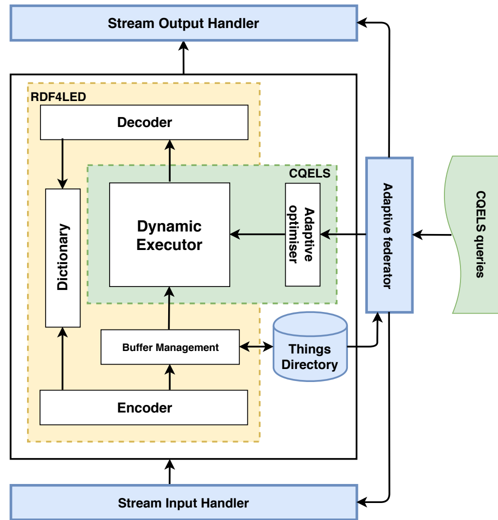


Fig. 1: Overview Architecture of Fed4Edge

The architectural overview of the system is depicted in Figure 1. The core components of CQELS and RDF4Led such as the *Dictionary*, *Encoder*, *Decoder*, *Dynamic Execution*, *Adaptive Query Optimiser*, *Buffer Manager* are reused in our Fed4Edge implementation. And the extension plugins of them such *Adaptive Federator*, *Thing Directory*, *Stream Input Handler* and *Stream Output Handler* are built to facilitate the federation mechanism proposed in Section 2. The technical details of these components are discussed next.

CQELS and RDF4Led share similar RDF data processing flows due to the fact that both systems apply the same RDF data encoding approach, which normalises RDF nodes into a fixed-size integer. By encoding the RDF nodes, most of the operators on RDF data can be executed on a smaller data structure rather than large variable-length strings. This approach is commonly used in many RDF data processors in order to reduce memory footprint, I/O time, and improve cache efficiency. The platform-agnostic design of CQELS allows the size of the encoded node to be tuned to adapt to a targeted platform without changing the implementation of other core components. Therefore, the Encoder, Decoder and Dictionary of RDF4Led can be easily integrated with CQELS for the RDF normalisation tasks. After receiving RDF data from RDF

stream subscriptions via the Stream Input Handler, the data is encoded by the Encoder. The encoded RDF triples are then sent to the Buffer Manager for further processing. The Decoder waits for the output from the Dynamic Executor, transforms the encoded nodes back to a lexical representation before sending them to the Stream Output Handler. The Encoder and Decoder share the Dictionary for encoding and decoding. Instead of using a 64-bit integer for encoding node as in the original version of CQELS, the Dictionary of RDF4Led uses 32-bit integers, which entails less memory footprint for cached data. Therefore, backed by RDF4Led, Fed4Edge can process 30 million triples with only 80MB of memory [17] on ARM computing architectures.

The Buffer Manager is responsible for managing the buffered data of windows and then feeding the data to the Dynamic Executor. Furthermore, the Buffer Manager also manages cached data for querying and writing the static data in the Thing Directory. Stream data is evicted from the buffer by the data invalidating policy defined by the window operators [15, 12]. Meanwhile, the flash-aware updating algorithms of RDF4Led are reused in order to achieve fast updating for static data [17].

The Dynamic Executor employs a routing-based query execution algorithm that provides dynamic execution strategies in each node [12, 13]. During the lifetime of a continuous query, the query plan can be changed by redirecting data flow on the routing network. The Adaptive Optimiser continuously adjusts the efficient query plan according to the data distribution in each execution step[15, 17]. RDF4Led and CQELS employ a similar query execution paradigm. While CQELS uses routing-based query execution algorithms, RDF4Led executes SPARQL with a one-tuple-at-a-time policy. Therefore, the same cost model of the Adaptive Optimiser can be applied when calculating the best plan for a query that has static data patterns. The Buffer Manager treats the buffer for join results of the static patterns as a window, and depending on the available memory, it will apply the *fresh update* or *incremental update* policy.

The Adaptive Federator acts as the query rewriter, which adaptively divides the input query into subqueries. For the implementation used in our experiments in Section 4, the rewriter will push down operators as close to the streaming nodes as possible by following the predicate pushdown practice in common logical optimisation algorithms. The Thing Directory stores the metadata subscribed by the other Fed4Edge engines (cf. Section 2) in the default graph. Similar to [7], such metadata allows endpoint services of the Fed4Edge engines to be discovered via the Adaptive Federator. When the Adaptive Federator sends out a subquery, it notifies the Stream Input Handler to subscribe and listens to the results returning from the subquery. On the other hand, the Stream Output Handler sends out the subqueries to other nodes or sends back the results to the requester.

4 Evaluation and Analyses

4.1 Evaluation Setup

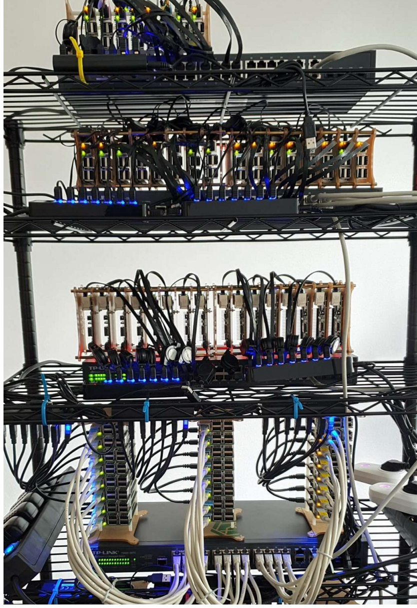


Fig. 2: The Evaluation Cluster of 85 Raspberry PI Nodes

Hardware & Software: The hardware for the experiment is a cluster of 85 Raspberry Pi model B nodes, each one is equipped with: Quad Core 1.2GHz Broadcom BCM2837 64bit CPU, 1GB RAM and 100 Mbps Ethernet. All nodes are connected to five TP-LINK Jet-Stream T2500-28TC switches. Each has 24 100 Mbps Ethernet ports 4 1000Mbps uplinks shown in Figure 2. As to the switching capacity, T2500-28TC has a non-blocking aggregated bandwidth of 12.8Gbps. Four switches for connecting streaming nodes will be connected to the fifth one via the 1000Mps links. This fifth switch is used to connected CQELS processing nodes. Every node uses Raspbian Jessie as the operating system and Open-JDK 1.7 for ARM as the JVM. We set 512MB as the maximum heap size for the Fed4Edge engine.

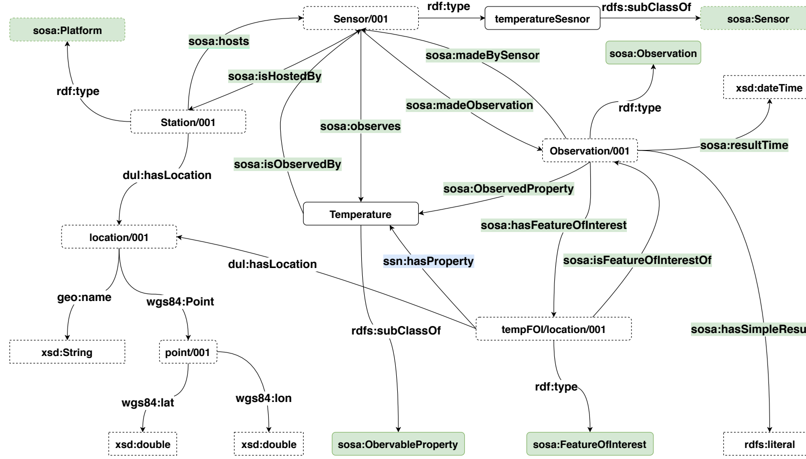


Fig. 3: RDF Stream Schema For NCDC Weather Data

Datasets and Queries: To prepare the RDF Stream dataset for the evaluation, we used the SSN/SOSA ontology [10] to map sensor readings of the NCDC Integrated Surface Database (ISD) dataset [5] to RDF. The ISD dataset is one of the most prominent weather datasets, which contains weather observation data

[5] https://www.ncdc.noaa.gov/

from 1901 to present, from nearly 20K stations over the world. A weather reading of a station produces an observation that covers measurements for temperature, wind speed, wind gust, etc. depending on the types of sensors equipped for that station. Each observation needs approximately 87 RDF triples to map its values and attributes to the schema illustrated in Figure 3. The data from different weather stations was split to multiple devices which acted as streaming nodes (i.e., the white nodes in Figure 4). Each streaming node hosts a Websocket server which manages WebSocket stream endpoints. The data is read from CSV files in local storage, then mapped to the RDF data schema in Figure 3 before streaming out.

We designed the following queries in order to show the advantages of cooperative federation for querying streaming data over a network of edge devices. Listings 4 and 5 respectively present the queries Q1 and Q2 that are used for measuring the improvement of the streaming throughput in simple federation cases. Q1 will return the updated temperature and the corresponding location and Q2 will answer the location where the latest temperature is higher than 30 degrees. The subquery of Q1 and Q2 contains only triple patterns for querying streaming data. With these simple join patterns, the behaviour of the system is mostly influenced by the behaviour of the network. The filter at line 7 of Q2 will reduce the number of intermediate results sent from the lower node, and therefore, it can highlight the benefit of pushing down processing operators closer to the data sources.

```
1 SELECT ?temp ?lat ?lon ?resultTime
2 WHERE {
3    STREAM ?streamURI [LATEST ON ssn:resultTime] {
4        ?obs sosa:hasSimpleResult ?temp; sosa:resultTime ?resultTime.
5        ?sensor rdf:type iot:TempSensor; made:Observation ?obs.}
6   ?streamURI prov:wasGeneratedBy ?sensor. ?sensor sosa:isHostedBy ?station.
7   ?station wgs84:Point ?loc. ?loc wgs84:lat ?lat;wgs84:lon ?lon.}
```

Listing 4: Q1: Return updated temperature and the corresponding location.

```
1 SELECT ?lat ?lon
2 WHERE {
3    STREAM ?streamURI [LATEST ON ssn:resultTime] {
4        ?obs sosa:hasSimpleResult ?temp; sosa:resultTime ?resultTime.
5        ?sensor rdf:type iot:TempSensor; made:Observation ?obs.
6        FILTER (?temp > 30) }
7    ?streamURI prov:wasGeneratedBy ?sensor. ?sensor sosa:isHostedBy ?station.
8    ?station wgs84:Point ?loc. ?loc wgs84:lat ?lat;wgs84:lon ?lon.}
```

Listing 5: Q2: Return the location where the latest temperature is higher than 30 degree.

For the queries that can show the collaborative behaviour of the participant edge nodes, we used the queries Q3 (as described in the example of Section 2) and query Q4 in Listing 6. The query Q3 has aggregation and top-k operators and the Q4 includes a complex join across windows.

```
SELECT ?city ?temp ?windspeed
WHERE {
   STREAM ?streamURI [RANGE 5m ON ssn:resultTime] {
       ?obs1 sosa:hasSimpleResult ?temp; sosa:resultTime ?resultTime.
       ?obs1 sosa:hasFeatureOfInterest ?foi1.
       ?foi1 ssn:hasProperty iot:Temperature. ?foi1 :hasLocation ?loc.
       FILTER (?temp > 30) }
   STREAM ?streamURI [RANGE 5m ON ssn:resultTime] {
       ?obs2 sosa:hasSimpleResult ?windspeed; sosa:resultTime ?resultTime.
       ?obs2 sosa:hasFeatureOfInterest ?foi2.
       ?foi2 ssn:hasProperty iot:WindSpeed. ?foi2 :hasLocation ?loc.
       FILTER (?windspeed > 15) }
   ?streamURI prov:wasGeneratedBy ?sensor. ?sensor sosa:isHostedBy ?station.
   ?station wgs84:Point ?loc. ?loc geo:city ?city.}
```

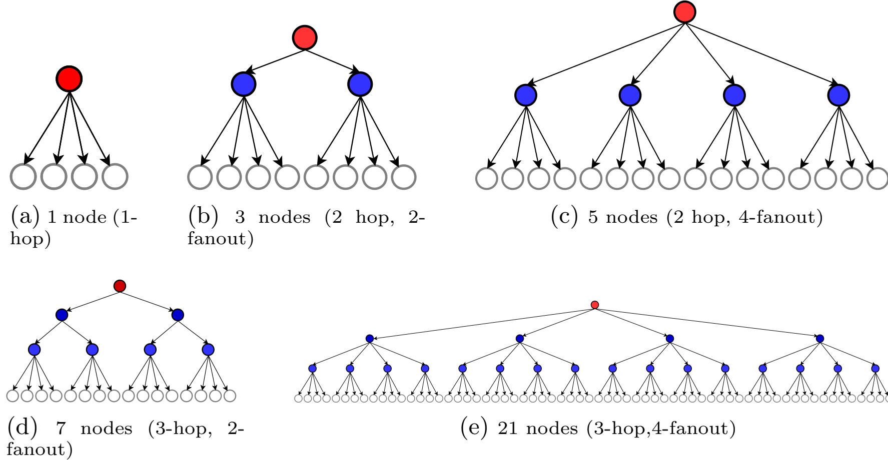
Listing 6: Q4: Return the city where the temperature is higher than 30 degree and the wind speed is higher than 15km in the last 5 minutes.

4.2 Experiments

Baseline Calibration (Exp1): In this experiment, we calibrated the maximum processing capability of a processing node as the baseline for the following federation experiment. We increased the number of stream nodes to observe the bottleneck phenomena whereby increasing more streaming nodes decreases the processing throughput of the network. Each streaming node will stream out recorded data as its maximum capacity. We will use Query 1 and its two variants as the testing queries. These two variants are made by reducing four triple patterns into 1 and 2 patterns respectively. The throughput is measured by using a timing stream whereby each streaming nodes will send timing triples indicating when each of them starts and finishes sending their data. In each test we will equally split 500k-1M observations among streaming nodes and record how much time to process these observations to calculate the average throughput. Note that we separated the streaming and processing processes in different physical devices to avoid performance and bandwidth interference which might have an impact on our measurements.

Fan-out Federation (Exp2): To test the possibility of increasing the processing throughput by increasing more edge nodes as autonomous agents to the network, we carried out the tests on five topologies as shown in Figure 4. The first topology (1-hop) in Figure 4a was the configuration that gave the peak throughput in Exp1. Let k be the number of hops the data has travel to reach to the final destination, we will increase k to add more intermediate nodes to this topology to create new topologies. As a result, we can recursively add n nodes to the root node (k=2, namely 2-hop) and then n nodes to the root node's children nodes ($k = 3$, namely 3-hop) whereby n is called the fanout factor (denoted as n-fanout). Then, we have $\sum_{i=0}^{k-1} n^i$ as the number of nodes of a topology with k hops and fanout factor n. We choose $n = 2$ and $n = 4$ (corresponding to the number of streaming nodes at the maximum throughput reported in *Exp1* below), thus, we have four new topologies with 3, 5, 7 and 21 processing nodes in Figure 4b, Figure 4c, Figure 4d, and Figure 4e. In each processing topology, the lowest processing nodes are connected with 4 streaming nodes. We will record

the throughput and delay for processing three queries (Q1, Q2, Q3 and Q4) on these five topologies in a similar fashion to Exp1.



(a) 1 node (1-hop) (b) 3 nodes (2 hop, 2-fanout) (c) 5 nodes (2 hop, 4-fanout)

(d) 7 nodes (3-hop, 2-fanout) (e) 21 nodes (3-hop,4-fanout)

Fig. 4: Federation Topologies

4.3 Results and Discussions

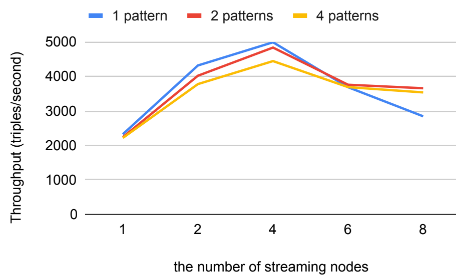


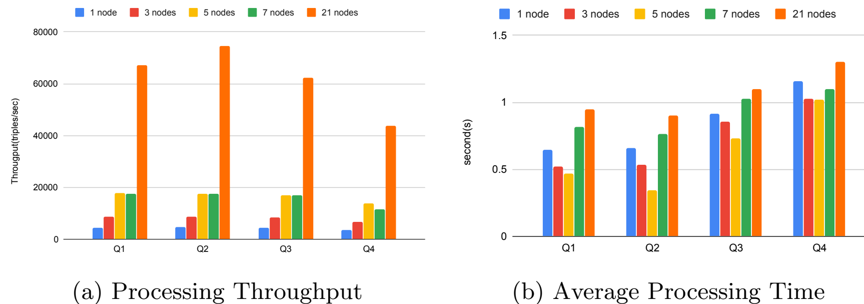
Fig. 5: Baseline Calibration

Figure 5 reports the results of the experiment Exp1. The maximum processing throughput for three variants of Query 1 on one single edge device is from 4200-5000 triples corresponding to 4 streaming nodes. It is interesting that increasing the number of streaming nodes more than 4 will gradually decrease the overall processing throughput. The results are consistent with different complexities of the variants of Query 1. We observed that the CPU usages were around 60-70% and the memory consumption is around 270-300MB in all tests. Therefore, we can conclude that the bottleneck was caused by the bandwidth limitations. We also carried out a similar test with Q1 on a PC (Intel Core i7 i7-7700K, 4GHz, 1GBb Ethernet and 16GB RAM) as the root node which has more than 10 times of processing power, memory and network bandwidth than those of a Raspberry Pi model B. As we expected, the PC's maximum throughput is approximately 36k triples/second, around 8-10 times the one with a Raspberry Pi. Note that this PC costs more than the price of 40 Raspberry Pi nodes.

Figure 6a shows the results of throughput improvements via federating the processing workload on other intermediate nodes in four proposed topologies. The results show that adding more nodes will increase the processing throughput in general. Most queries have their processing throughput consistently boosted

up as a considerable amount of processing load were done at the intermediate nodes. However, the increase is not consistently correlated with the total number of processing nodes. In fact, the topology with 5 nodes in Figure 4d gives a slightly higher throughput than those of the topology with 7 nodes in Figure 4c. This can be explained by the fact that both topologies have 4 processing nodes at the lowest levels (called leaf processing nodes, i.e, connecting to streaming nodes) but the data in the latter topology has to travel 1 more hop in comparison with the former. Due to our pushing down rewriting strategy presented in Section 3, these two upper blue nodes in Figure 4c did not significantly contribute to the overall throughput but on the other hand cause more communication overhead.

Look closer to the reported figures, we see a high correlation between the number of leaf processing nodes, i.e. n^{k-1}, and the processing throughput in all topologies. This shows that our proposed approach is able to linearly scale a network of IoT devices by adding the more devices on demand. In particular, a network of 21 Raspberry Pi nodes can collaboratively process up to 74k triples/seconds or equivalent to roughly 8500 sensor observations/second that are streamed from other 64 streaming nodes. Hence, the above 20K weather stations across the globe of NCDC can be queried via such a network with the update rate 20-30 observations per minute which are much faster than the highest update rate currently supported by NCDC [6], i.e. ASOS 1-minute data. Moreover, the processing capacity of this network is twice more than that of the above PC but it only costs roughly a half of the PC. Regarding the energy consumption, each Raspberry Pi only consumes around 2W in comparison of 240W of the above PC.



(a) Processing Throughput (b) Average Processing Time

Fig. 6: Federation Topologies

Figure 6b reports the average time for each observation to travel through a processing pipeline specified by each query on different topologies, i.e., average processing time. It shows that adding more intermediate nodes for query Q1 and Q2 can lower the average processing time as it can reduce queuing time at some nodes. That means communication time might be a dominant factor for the delay in these processing pipelines. In queries Q3 and Q4, we witness the consistent increase in processing time wrt. the number of hops which explains the nature of query Q3 and Q4 that needs more coordination among nodes.

[6] `https://www.ncdc.noaa.gov/data-access/land-based-station-data`

However, it is interesting that increasing 1 hop in organising a network topology just adds 10-15% delay while the maximum throughput gain is linear to n^{k-1}.

4.4 Follow-up Challenges

We observed the CPU, memory consumption and bandwidth in our experiments. It is interesting that all tests used 60-70% of CPU (across 4 cores), 25-30% of physical memory and 20-40% of Ethernet bandwidth (i.e., 100Mbps). Our reported performance figures show that edge devices have enough resources to enable semantic interoperability for the edge computing paradigm. From our analyses of hardware and software libraries, the most potential suspects for the processing bottleneck are related to the communication among the nodes. Hence, there is a lot of room to make our approach much more efficient and scalable. In this context, to help 100 billions and more edge devices to reach their full potential, we outline some interesting research challenges below.

The first challenge is how to address the multiple optimisation problems that such a federated processing pipeline entails. The first one is how to optimise an RSP engine for edge devices which have distinctive processing and I/O behaviours from those of PC/workstations due to their own design philosophies. The second challenge is about how to find optimal operator placements on very dynamic execution settings. The subsequent challenge is how to define cost models which are no longer limited to processing time/throughput, but need to cover several cost metrics such as bandwidth, power consumption and robustness.

Looking beyond database-oriented optimisation goals, another relevant research challenge would be how to model socioeconomic aspects as the control or optimisation scheme for such a cooperative system. In particular, autonomous RSP processing nodes can be operated by different stakeholders which have different utility functions dictating when and how to join a network and to share data and resources. To this end, the coordination strategies become related to the game theory which inspired some relevant proposals in both the stream processing and Web communities. For instance, [1] proposed a contract-based coordination scheme based on mechanism design, a field in economics and game theory that designs economic mechanisms or incentives toward desired objectives. Similarly, [9] also proposed to use mechanism design for establishing an incentive-driven coordination strategy among SPARQL endpoints. Inspired by this line of work, we also proposed an architecture for in-cooperating blockchain with RDF4Led [18] to pave the way to in-cooperate with such incentive and contract-based coordination strategies.

Regarding cooperation and negotiation among RSP autonomous agents, a potential research challenge is the study and exploration of protocols and strategies that follow the multi-agent system paradigm. Although early works on the topic [26] point at potential opportunities in this area, several aspects have not been studied yet. These include the usage of individual contextual knowledge for local decision making (potentially through reasoning) and for a resource-optimised distribution of tasks among a set of competing/associated nodes. The dynamics of these federated processing networks would need to adapt to changing

conditions of load, membership, throughput, and other criteria, with emerging behaviour patterns on the sensing and processing nodes.

5 Related work

Semantic interoperability in the IoT domain has gained considerable attention both in the academic and industrial spheres. Beyond syntactic standards such as SensorML, semantically rich ontologies such as SSN-O/SOSA [10] have shown a significant impact in different IoT projects and solutions, such as OpenIoT [24], SymbIoTe [25], or BigIoT [5]. Other related vocabularies, such as the ThingsDescription ontology, have also recently gained support from different IoT vendors, aiming at consolidating it as a backbone representation model for generic IoT devices and services. Regarding the representation of data streams themselves, the VoCaLS vocabulary [27] has been designed as a means for the publication, consumption, and shared processing of streams. Although these ontology resources provide different and complementary ways to represent IoT and streaming data, they require the necessary infrastructure and software components (or agents) able to *interpret* the stream metadata, and apply coordination/cooperation mechanisms for federated/decentralised processing, as shown in this paper.

The processing of continuous streaming data, structured according to Semantic Web standards has been studied in the last decade, generally within the fields of RDF Stream processing (RSP) and Stream Reasoning [8]. A number of RSP engines have been developed in this period, focusing on different aspects including incremental reasoning, continuous querying, complex event processing, among others [3, 15, 6, 20]. However, most of these RDF stream processors lack the capability of interconnecting with each other, or to establish cooperation patterns among them. The coordination among RDF stream processing nodes is sometimes delegated to a generic cloud-based stream processing platform such as Apache Storm (e.g [16]) or Apache Spark (e.g [20]). In contrast, in this paper, we investigate a more decentralised environment whereby participant nodes can be distributed across different organisations. Moreover, the hardware capabilities of such processing nodes are different from the cloud-based setting, i.e. resource-constraint edge devices.

Regarding the distributed processing and integration of RSP engines on a truly decentralised architecture, different aspects and building blocks have surfaced in the latest years. Initial attempts to provide HTTP-based service interfaces for streaming data were explored in [3]. Other contributions in this line are the RSP Service Interface[7], and the SLD Revolution framework [2]. These propose the establishment of distributed workflows of RSP engines, using lazy-transformation techniques for optimised interactions among the engines. Further conceptualisations of RDF stream processing over decentralised entities have been presented in works such as WeSP [7][8], which advocates for a community-driven definition of stream vocabularies and interoperable interfaces. Cooper-

[7] http://streamreasoning.org/resources/rsp-services
[8] http://w3id.org/wesp/web-data-streams

ation strategies among RDF stream processors, or *stream reasoning agents* is discussed in [26], introducing potential challenges and opportunities for federated processing through negotiation established across multi-agent systems.

6 Conclusion

This paper presented a continuous query federation approach that uses RSP engines as autonomous processing agents. The approach enables the coordination of edge devices' resources to process query processing pipelines by cooperatively delegating partial workload to their peer agents. We implemented our approach as an open source engine, Fed4Edge, to conduct an empirical study in "cooperative sensing" scenarios. The resourceful experiments of the study show that the scalablity can be significantly improved by adding more edge devices to a network of processing nodes on demand. This opens several interesting follow-up research challenges in enabling semantic interoperability for the edge computing paradigm. Our next step will be investigating on how to adaptively optimise the distributed processing pipeline of Fed4Edge. Another interesting step is studying how the communication will effect its performance and scalability on an Internet-scale setting whereby the processing nodes are distributed across different networks and countries.

Acknowledgements This work was funded in part by the German Ministry for Education and Research as BBDC 2 - Berlin Big Data Center Phase 2(ref. 01IS18025A), Irish Research Council under Grant Number GOIPG/2014/917, HES-SO RCSO ISNet grant 87057 (PROFILES), and Marie Skodowska-Curie Programme H2020-MSCA-IF-2014 (SMARTER project) under Grant No. 661180.

References

1. M. Balazinska, H. Balakrishnan, and M. Stonebraker. Contract-based load management in federated distributed systems. In *NSDI'04*, 2004.
2. M. Balduini, E. Della Valle, and R. Tommasini. Sld revolution: A cheaper, faster yet more accurate streaming linked data framework. In *ESWC*, 2017.
3. D. F. Barbieri, D. Braga, S. Ceri, and M. Grossniklaus. An execution environment for C-SPARQL queries. In *EDBT 2010*, 2010.
4. Enabling mass iot connectivity as arm partners ship 100 billion chips. http://tiny.cc/uiefcz.
5. A. Bröring, A. Ziller, V. Charpenay, A. S. Thuluva, D. Anicic, S. Schmid, A. Zappa, M. P. Linares, L. Mikkelsen, and C. Seidel. The big iot api-semantically enabling iot interoperability. *IEEE Pervasive Computing*, 17(4):41–51, 2018.
6. J.-P. Calbimonte, O. Corcho, and A. J. G. Gray. Enabling ontology-based access to streaming data sources. In *ISWC*, pages 96–111. Springer, 2010.
7. D. Dell'Aglio, D. L. Phuoc, A. Le-Tuan, M. I. Ali, and J.-P. Calbimonte. On a web of data streams. In *DeSemWeb@ISWC*, 2017.
8. D. DellAglio, E. Della Valle, F. van Harmelen, and A. Bernstein. Stream reasoning: A survey and outlook. *Data Science*, 1(1-2):59–83, 2017.

9. T. Grubenmann, A. Bernstein, D. Moor, and S. Seuken. Financing the web of data with delayed-answer auctions. In *WWW '18*, 2018.
10. A. Haller, K. Janowicz, S. J. D. Cox, M. Lefrançois, K. Taylor, D. Le-Phuoc, J. Lieberman, R. García-Castro, R. Atkinson, and C. Stadler. The modular SSN ontology: A joint W3C and OGC standard specifying the semantics of sensors, observations, sampling, and actuation. *Semantic Web*, 10(1):9–32, 2019.
11. S. Kaebisch, T. Kamiya, M. McCool, and V. Charpenay. Web of things (wot) thing description. *W3C, W3C Candidate Recommendation*, 2019.
12. D. Le-Phuoc. Operator-aware approach for boosting performance in RDF stream processing. *J. Web Sem.*, 42:38–54, 2017.
13. D. Le-Phuoc. Adaptive optimisation for continuous multi-way joins over rdf streams. In *Companion Proceedings of the The Web Conference 2018*, WWW '18, pages 1857–1865, 2018.
14. D. Le-Phuoc, M. Dao-Tran, C. Le Van, A. Le Tuan, T. T. N. Manh Nguyen Duc, and M. Hauswirth. Platform-agnostic execution framework towards rdf stream processing. In *RDF Stream Processing Workshop at ESWC2015*, 2015.
15. 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'11*, pages 370–388, 2011.
16. D. Le-Phuoc, H. N. M. Quoc, C. L. Van, and M. Hauswirth. Elastic and scalable processing of linked stream data in the cloud. In *ISWC*, pages 280–297, 2013.
17. A. Le-Tuan, C. Hayes, M. Wylot, and D. Le-Phuoc. Rdf4led: An rdf engine for lightweight edge devices. In *IOT '18*, 2018.
18. A. Le-Tuan, D. Hingu, M. Hauswirth, and D. Le-Phuoc. Incorporating blockchain into rdf store at the lightweight edge devices. In *Semantic '19*, 2019.
19. A. Munir, P. Kansakar, and S. U. Khan. Ifciot: Integrated fog cloud iot: A novel architectural paradigm for the future internet of things. *IEEE Consumer Electronics Magazine*, 2017.
20. X. Ren and O. Curé. Strider: A hybrid adaptive distributed rdf stream processing engine. In *The Semantic Web – ISWC 2017*, 2017.
21. S. Sakr, M. Wylot, R. Mutharaju, D. Le Phuoc, and I. Fundulaki. *Processing of RDF Stream Data*. Springer International Publishing, Cham, 2018.
22. M. Satyanarayanan. The emergence of edge computing. *Computer*, 2017.
23. B. Smith. Arm and intel battle over the mobile chip's future. *Computer*, 2008.
24. J. Soldatos et al. Openiot: Open source internet-of-things in the cloud. In *Interoperability and open-source solutions for the internet of things*. Springer, 2015.
25. S. Soursos, I. P. Žarko, P. Zwickl, I. Gojmerac, G. Bianchi, and G. Carrozzo. Towards the cross-domain interoperability of iot platforms. In *2016 European conference on networks and communications (EuCNC)*, pages 398–402. IEEE, 2016.
26. R. Tommasini, D. Calvaresi, and J.-P. Calbimonte. Stream reasoning agents: Blue sky ideas track. In *AAMAS*, pages 1664–1680, 2019.
27. R. Tommasini, Y. A. Sedira, D. DellAglio, M. Balduini, M. I. Ali, D. Le Phuoc, E. Della Valle, and J.-P. Calbimonte. Vocals: Vocabulary and catalog of linked streams. In *International Semantic Web Conference*, 2018.