Towards Cascading Reasoning for Generic Edge Processing

Pieter Bonte[1], Femke Ongenae[1]

[1]Ghent University - imec, Technologiepark-Zwijnaarde 126, B-9052 Ghent, Belgium

Abstract

Many data-intensive applications, including smart logistics, smart cities, intelligent factories, etc. have been made possible by the Internet of Things (IoT). Due to the large amounts of data produced, it is no longer viable to upload all data to the cloud, as it results in network congestion, low throughput, high latency, and privacy issues. Edge processing provides a solution to these problems and aims to process data as close to the source. However, a generic approach to edge processing that can decide which components to offload is still missing. Cascading Reasoning (CR) is a vision that aims to realize highly expressive reasoning over high-frequency data by introducing a layered approach. The lowest layers employ low-complexity techniques and select relevant parts of the data. Going up in the layers, the complexity increases while the data volume decreases as each layer selects relevant parts for further processing. CR aligns with Edge processing, as lower layers can be processed closer to the source, i.e. the Edge, and the expressive reasoning allows to handle the data variety and domain knowledge imposed by the IoT. However, until now, CR approaches required to manually define the various layers. In this paper, we propose a generic CR Edge processing platform that can decide which parts of the queries to offload to the Edge and how to optimize the queries and reasoning rules for efficient Edge processing.

Keywords

Stream Reasoning, Edge Processing, Cascading Reasoning

1. Introduction

The Internet of Things (IoT) has given rise to many data-intensive applications such as smart logistics, smart cities, pervasive health, intelligent factories, etc. [1]. Many of these applications deploy large quantities of sensors and have low tolerance for delays [1]. Even with the rise of 5G networks, it has become unfeasible to upload all data to the cloud for processing as too much bandwidth will be consumed, the operational cost will rise and the overall latency will be too high [1]. Furthermore, critical and time-sensitive tasks such as emergency shutdowns in smart factories or monitoring of critical patients cannot rely on the connection with the cloud to make local decisions [1]. Edge processing provides a solution to the problems described above by processing the data as close to its source as possible [2]. However, deciding which parts of the processing can be offloaded to the Edge for processing is to date only possible for tailored applications that have been hard-coded to enable edge computing [2]. A generic approach towards edge computing is still missing.

pieter.bonte@ugent.be (P. Bonte)
0000-0002-8931-8343 (P. Bonte); 0000-0003-2529-5477 (F. Ongenae)

The generated data in the IoT imposes data variety, as it results from a range of different sources, each possibly describing their data differently on both structural, syntactical and semantical level [3]. Semantic Web technologies are the preferred technology to solve this data integration problem [4]. Semantic reasoning is often required to interpret the domain knowledge, something very prominent in the IoT [5]. However, there is a mismatch between the complexity of reasoning algorithms and the rate data is produced in the IoT. The Stream Reasoning community and more particular the RDF Stream Processing (RSP) community have focused on processing RDF streams through the use of continuous declarative queries, such as dialects of SPARQL [6], disregarding much of the reasoning challenges. Cascading Reasoning (CR) is a vision that emerged from the Stream Reasoning community to tackle this mismatch between reasoning complexity and frequency of changes in data streams [7]. CR proposes a layered approach where starting from the lowest layers, close to where data is produced, low complexity techniques are employed that can handle huge amounts of data and high velocity updates, going up the the layers, each layer selects parts of the data that might be of interest and process this selection with techniques that increase in complexity. At the highest layers, techniques that use highly complex reasoning algorithms can be employed as the lower layers already filtered out the majority of data. From a top-down perspective, high-level queries, using abstracted concepts are issued to the CR platform. The CR vision aligns nicely with the Edge processing paradigm as lower layers can be employed at the Edge and Fog. However, deciding how to automatically split up the high-level queries and offload part to the Edge, i.e. lower layers, is still an open problem. CR solutions have so far relied on an manual definition of the different layers [4].

In this paper we propose a generic CR Edge processing platform that can decide how to offload the processing to the Edge by analyzing both the queries and the data produced at the Edge. In order to achieve semantic interoperability, an annotation step is required to convert the raw data produced by the sensors to semantic data, e.g. through the use of RML. Recent developments in annotation research allows to extract the shape of these mapping [8], which allows to interpret what kind of data will be produced by certain streams. By combining a bottom-up and top-down analysis, i.e. analyzing the shapes of the data and the queries registered to the platform, the platform can really understand the data generated and requested by the user. This unique situation allows to optimize the Edge data processing pipeline and by understanding both data generated and queries, optimizing the offloading towards the cloud of the registered queries.

This paper is structured as follows: Section 2 introduces the needed background to understand the remainder of the paper, while Section 3 introduces a running example. Section 4 discusses our proposed CR approach and Section 5 details the evaluation. Section 6 discusses the related work and Section 7 concludes the work and discusses future work directions.

2. Background

In this section, we summarize the knowledge necessary to understand the remainder of this paper. We start with the definition of an (RDF) stream:

Definition 1. A data stream S is a possibly infinite multiset of elements $\langle s, \tau \rangle$, where s is a data item and $\tau \in \mathcal{T}$ is a timestamp of the element, with $\mathcal{T}$ a time domain. An **RDF Stream** is

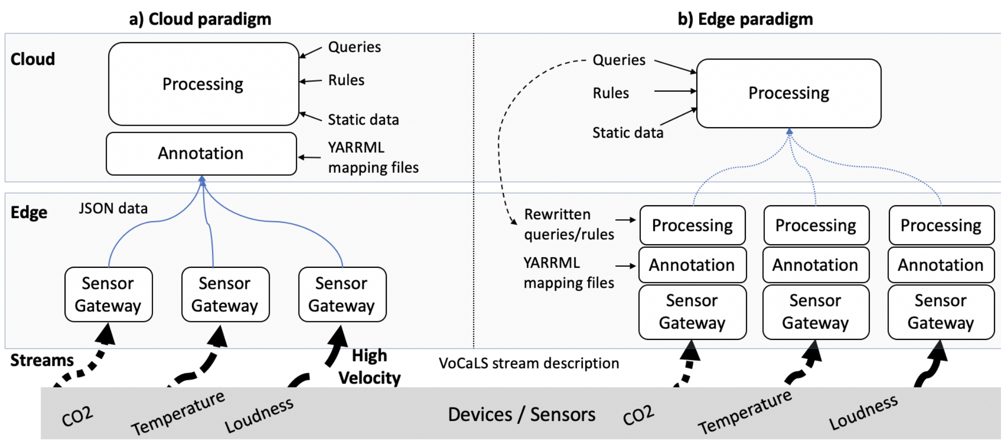

Figure 1: Left (a): Overview of the running example when pushing all data to the cloud. Right (b): Overview of the running example when most data is processed at the Edge

a stream where the data item s is an RDF object.

Every element in a (RDF) stream is also called an *event*. Processing these (RDF) streams requires a special class of queries that run under continuous semantics:

Definition 2. Under continuous semantics, the result of a query is the set of results that would be returned if the query were executed at every instant in time.

We call this class of queries *continuous queries* or *registered queries* as they are registered upfront and continuously evaluated. As streams are possible infinite, special mechanisms are needed to cut these streams in processable chunks. Time-based windows allow to extract parts of these streams for processing:

Definition 3. The time-based window operator $\mathcal{W}$ is a triple (α, β, t^0) that defines a series of windows of width (α) and that slide of (β) starting at t^0.

Events in a stream typically describe a minimal information unit, as transmission can be costly. Therefore, streaming data often needs to be combined with static data that describes contextual information related to the event that does not change very frequently over time. This process is called *Enrichment* and requires a join between events in the stream and static data [9].

To describe the domain knowledge, we focus on *Datalog*, i.e. if-then rules, where the if-part is called the *body* of the rule and the then-part the *head* of the rule. We focus on rules that do not introduce new variables or disjunctions in the head of the rule and no support for negation. We will use a subset of *Notation3* rules[1] that relate to the expressivity of *Datalog*. *Notation3* (N3) is a superset of RDF that allows the definition of rules in the following syntax: *{body of the rule.}=>{head of the rule}* where both body and head can define triple patterns.

3. Running Example

A Smart Building use case will be used as running example. Each room is equipped with various sensor, each measuring comfort parameters such as temperature, humidity, loudness and CO2

[1]https://w3c.github.io/N3/spec/

values. Each room contains a local sensor gateway that has some processing capabilities. In the cloud scenario, the gateway annotates the raw sensor data to the semantic model and transmits the sensed values to the cloud. A CO2 sensor produces the following raw sensor data in JSON-format:

```
{   "sensorID": "1234",
    "observationID": "5674",
    "value": 765 }
```

Listing 1: Raw CO2 sensor data in JSON

Detailing the ID of the sensor, the ID of the observation and the actual sensed value. Note that the observed property, i.e. CO2, is not part of the event as it does not change over time. Therefore it is described in the static data.

In order to convert the raw sensor data to RDF, typically an annotation step is defined using some sort of mapping language. In this example we are using YARRRML [10], i.e. a human readable text-based representation of RML.

```
mappings:
  sensor:
    sources:
    s: iot:$(observationID)
    po:
      - [a, ssn:Observation]
      - [ssn:madeBySensor, iot:$(sensorID)]
      - [sosa:hasSimpleResult, $(value)]
```

Listing 2: YARRRML mapping file describing the annotation to RDF

The mapping defines how the JSON data can be converted to the SSN ontology, creating an *Observation* with a certain *value* that was made by a certain *Sensor*. The various streams can be described and discovered using VoCaLS [11], i.e. a vocabulary for describing RDF stream meta-data. We envision that the annotation as described above is part of said meta-data.

Static and contextual information is stored in the cloud and describes various properties of the different rooms of the building, which sensors are installed in each room, what kind of activities does each room host, the layout of the room, the positioning of each room within the larger building, etc. For example:

```
...
:sensorX  a  sosa:Sensor;
          sosa:observes  :temp, :loudness, :co2, humidity;
          :hasLocation  :officeY.
:officeY  :connectedTo  :officeZ, officeQ;
          :hasName  "200.009"^^xsd:string.
...
```

Listing 3: Extract of static data

The building manager wants a unified view over all data produced by various heterogeneous IoT sensors. For example, the following query (Query1) retrieves all observations from a specific room. In order to evaluate this query, the data from the sensor streams needs to be combined with the static data:

```
?obs a Observation;            // stream
    hasSimpleResult ?value;    // stream
    madeBySensor ?sensor.      // stream
?sensor hasLocation ?loc.      // static
?loc hasName "200.009"         // static
```

Listing 4: Query1: retrieve all sensor data for a particular room

Query2 retrieves all Comfort Observations, which is defined as a high-level definition in the domain knowledge:

```
?obs a ComfortObservation;    // stream + domain knowledge
    hasSimpleResult ?value.   // stream
```

Listing 5: Query2: retrieve all CO2 sensor data

Where the domain knowledge is defined through the following N3 rules:

```
R1: {?x a :TempObservation} => {?x a :ComfortObservation}
R2: {?x a :LoudnessObservation} => {?x a :ComfortObservation}
R3: {?x a :Observation, ?x :madeBySensor ?s, ?s a :TempSensor}
              => {?x a :TempObservation}
R4: {?x a :Observation, ?x :madeBySensor ?s, ?s a :LoudnessSensor}
              => {?x a :LoudnessObservation}
R5: {?s a :Sensor, ?s :observes ?p, ?p a :Temperature}
              => {?s a :TempSensor}
R6: {?s a :Sensor, ?s :observes ?p, ?p a :Loudness}
              => {?s a :LoudnessSensor}
```

Listing 6: Domain knowledge defined in N3 rules

The domain knowledge defines that both *TempObservation* and *LoudnessObservation* are types of *ComfortObservation* and further defines what a *TempObservation* or *LoudnessObservation* is. The former is an *Observation* made by a *TempSensor*, while the later is made by a *LoudnessSensor*. Furthermore, a *TempSensor* is a *Sensor* that *observes* the property *Temperature* and a *LoudnessSensor* is a *Sensor* that *observes* the property *Loudness*.

Instead of uploading all the sensor data to the cloud, enrich it with static data and interpret the domain knowledge, the building manager want to process as much as possible at the Edge, i.e. on the local sensor gateways, in order to ensure privacy, lower latency, reduce network congestion and decrease the load on the cloud infrastructure. Figure 1 a) visualizes the flow when pushing the data to the cloud, while Figure 1 b) visualizes the desired scenario where most data is processed at the Edge.

4. Cascading Reasoning for Edge Processing

In order to enable CR for Edge processing, a combination of a bottom-up and top-down optimization is proposed. Section 4.1 describes the first step needed for the bottom-up analysis. This first step fixes the structure of the events in the streams. Section 4.2 uses this extracted structure and introduces the first step of the top-down analysis which starts from the registered queries, available static data and the extracted event structures to rewrite queries to be optimized to be evaluated at the Edge. Section 4.3 extend the algorithm of Section 4.2 to realize the optimized evaluation of the defined domain knowledge through rules.

4.1. Shape extraction from Annotation

In the first phase, an interpretation is done of what shape data resulting from the stream will have. This can be done as a pre-processing step by analyzing the RML mappings from the annotation phase. Through the use of recent developments, these RML mappings can be converted to SHACL shapes [8]. Once the shape has been extracted, a so-called *blueprint* of the events in the stream is created, e.g. a *blueprint* of the sensor observations. We define N3 rules to define how the *blueprint* should be extracted from the converted shapes:

```
{?x a NodeShape. ?x targetClass ?clss} => {_:t a ?clss}
{?x a NodeShape. ?x sh:property [ sh:path ?prop]}
                                   => {_:s ?prop _:o}
...
```

Executing the conversion rules with creates the following *blueprint*:

```
_:t  a ssn:Observation
_:t  ssn:madeBySensor _:q
_:t  sosa:hasSimpleResult _:z
_:t  :observedProperty _:p
```

This *blueprint* fixes the structure of the events and will be used in the top-down optimizations. Note that the blank nodes follow the original interpretation of blank nodes in RDF and can be seen as existential quantifiers or variables [12].

4.2. Query rewriting

Algorithm 1 Query Rewrite Algorithm ***rewriteBody***

```
Require: Static Data $S$, BluePrint $BP$, Query $Q$
1: $B_{all} \leftarrow []$    ▷ Stores the bindings of the query evaluation
2: $B_{origin} \leftarrow []$    ▷ Stores the origin of the bindings, either static or streaming data
3: $Q_{new} \leftarrow []$    ▷ Stores the rewritten triples patterns of the Query
4: for TP in Q do    ▷ First bindings and their origin are retrieved
5:     $B \leftarrow eval(TP, S, BP)$    ▷ Evaluates the Triple Pattern (TP)
6:     $B_{origin} \leftarrow B_{origin} \cup getOrigin(B, S, BP)$    ▷ Retrieves the origin of the bindings
7:     $B_{all} \leftarrow B_{all} \bowtie B$    ▷ Joins the bindings
8: end for
9: for TP in Q do ▷ Patterns on static data are dropped, on streaming data are kept and that joins static and streaming data are rewritten
10:    $B_{tp} \leftarrow B_{all}[TP]$    ▷ Retrieve the bindings for the TP
11:    if $isStaticStreamJoin(B_{origin}, B_{tp}$ then
12:        $TP' \leftarrow injectStatic(TP, B_{origin}, B_{tp})$    ▷ Inject static data in TP
13:        $Q_{new} \leftarrow Q_{new} \cup TP'$
14:    end if
15:    if $isStreamOnly(B_{origin}, B_{tp}$ then
16:        $Q_{new} \leftarrow Q_{new} \cup TP$    ▷ Patterns on streaming data only are kept
17:    end if
18: end for
19: return $Q_{new}$
```

Once the event *blueprint* has been extracted, we can optimize the registered queries. The RSP-QL query language makes a clear distinction between what is executed on either static or streaming data, however, not all RSP dialects do. For example, in the CSPARQL language [13] this distinction is not explicit, nor is it in Streaming MASSIF [4]. Thus as a first step, it is

	?obs	?value	?sensor	?loc
value	_:t	_:z	:sensorX	:officeY
origin	event	event	event/static	static

Table 1
Bindings static query evaluation

necessary to extract which parts of the query need to be evaluated on the static data and which parts on the streaming data. Since the static data is typically available during a pre-processing phase and the *blueprint* of the sensors observations have been extracted, it is possible to split up the query in a streaming and static evaluation. This approach is similar to the basic federated querying algorithms [14] where each triple pattern is evaluated on the available data sources. In our case, each triple pattern is valuated on the static data set and the *blueprints*. Algorithm 1 shows the query rewrite algorithm in pseudo code. It takes the static data (S), the blueprint (B) and the query (Q) and returns the optimized query. Line 4 till 7 evaluates all triple patterns and retrieves the origin that correspond with the bindings that resulted from the evaluation of each triple pattern. Note that the blank nodes in the data are treated as existential variables. Thus while performing the joins between the bindings (line 7), blank nodes do not require an exact match but are treated as existential variables. Next, we try to rewrite the query such that as much of the part that needs to be evaluated on the static data can be injected directly in the query (starting at line 9). This can be done by identifying the query variables (or bindings) that require a join between the streaming and static data (line 11). Once these variables have been identified, the bindings can be inspected and if the number of bindings is low, this information can be directly injected in the query (line 12). Triple patterns that are evaluated only on the *blueprint* are kept (line 15), and those evaluated on the static data are dropped. This results in a query that can be evaluated directly on the stream, disregarding the need to combine the streaming data with the static data. This also means that this query can be directly evaluated at the Edge.

Example 1 (Rewriting)**.** Evaluating the triple patterns from *Query1*, results in the following bindings, showing the binding values and the origin, i.e. static, streaming or a join between both:

Since *?sensor* has been identified as a variable that joins static and streaming data, its bindings can be injected in the query. This results in an optimized query that can be evaluated directly on the stream and thus on the Edge:

```
?obs a Observation;            // stream
    madeBySensor :sensorX;     // stream (static injection)
    hasSimpleResult ?value;    // stream
```

Listing 7: Query1-Edge: rewritten query that retrieve all sensor data for a particular room where :sensorX is installed.

4.3. Reasoning-enabled rewriting

Rewriting becomes more challenging when inference rules are in place. Our platform supports rewriting for standard Datalog rules, i.e. rules without negation and no introduction of new variables in the head of the rule. The goal is to identify a minimal subset of necessary rules required to successfully evaluate the registered query, given the stream *blueprint*, the static data

Algorithm 2 Rule Rewrite Algorithm ***rewriteRule***

```
Require: Static Data $S$, BluePrint $BP$, Rule set $R$, Rule Rewrite Candidate $r$, Pruned Rules $R_{new}$
 1: $B_{all} \leftarrow []$                                         ▷ Stores the bindings of the rule evaluation
 2: for TP in body(r) do
 3:     if TP in RuleHeads(R) then
 4:         $r_{TP} \leftarrow ruleForHead(TP)$
 5:         $B \leftarrow rewriteRule(S, BP, R, r_{TP}, R_{new})$                        ▷ Recursion call
 6:         if $isHeadVariablesBoundToStream(B, head(r_{TP}))$ then
 7:             $body' \leftarrow rewriteBody(S, BP, body(r_{TP}))$                       ▷ Rewrites body
 8:             $R_{new} \leftarrow R_{new} \cup body' \rightarrow head(r_{TP})$
 9:         end if
10:         $B_{all} \leftarrow B_{all} \bowtie B(head(r_{TP}))$        ▷ Joins the bindings for the head of the rule
11:     else
12:         $B \leftarrow eval(TP, S, BP)$
13:         $B_{all} \leftarrow B_{all} \bowtie B$
14:     end if
15: end for
16: return $B_{all}$
```

and the whole set of rules. First the relevant rules are identified through a backward chaining process, i.e. the query is analyzed and each concept/property in the query is matched against the heads of the various rules. Algorithm 2 provides the algorithm for the rewriting of the identified rules, which uses the algorithm of Section 4.2 to rewrite certain rule bodies. For each of the identified rules, the process investigates the patterns in the body of the rule (line 2) and either expands the pattern in another rule through a recursion call if the pattern is also a head of another rule (line 3-5). When the recursion returns and we can identify that the resulting bindings that correspond to the variables in the head of the rule are bound to data from the stream *blueprint*, we rewrite the rule body such that the bindings from the body related to static data can be injected in the rule body and add the new rule (line 6-8). For the rewriting process, the algorithm from Section 4.2 is used. When the variables in the head relate to the static data, this rule is skipped and only the resulting head bindings are further used in the recursive process (line 10). If the pattern in the body is not also a rule head, no recursion is needed and we can simply evaluate the pattern and try to make the join with the previous available bindings (line 11-13).

We thus make a distinction between rules where the head contains variables that are bound to static data or to data from the stream *blueprint*. Only rules with head variables bound to data from the stream *blueprint* are kept and rewritten such that the static data is injected in the rules, similar as for the queries as described in Section 4.2.

Example 2. For this example, we will use the rules as defined in the running example and use abstract query *Query2*, that asks for all *ComfortObservations* instead of defining each of their low level details.

The process starts by analyzing the query and finds the concept *ComfortObservation* as the head of rules R1 and R2 which can be used in the backward chaining process. Doing so, both *TempObservation* and *LoudnessObservation* need to be verified in the recursive call. First, evidence for any *TempObservation* is checked directly in the data, as there are none, the process tries to recursively unfold the *TempObservation* as it is also the head of R3 into *?x a Observation, ?x madeBySensor ?s, ?s a TempSensor*. Evidence for *?x a Observation, ?x madeBySensor ?s* can be detected in the *blueprint*, but for the triple pattern *?s a TempSensor*, the process has to further unfold as the pattern is also the head of rule R5 which can be unfolded into *?s a Sensor, ?s observes ?p, ?p a Temperature*, for which evidence is found in the static data. The backward process can bind *:sensorX* to the variable *?s* and thus infer *:sensorX* as a *TempSensor.* As the variables in the head of this rule are not bound to any data from the stream *blueprint*, the rule is not kept but the bindings are used as a return value for the recursive backward chaining process. This allows to completely evaluate the body of the *TempObservation* rule (R3). As for this rule, the variables in the head of the rule are bound to the data from the stream *blueprint*, the body of the rule is rewritten as described in Section 4.2 and added to the resulting rule set. This results in the following rules:

```
R1:  {?x a TempObservation} => {?x a ComfortObservation}
R3': {?x a Observation, ?x madeBySensor :sensorX.}
                => {?x a TempObservation}
```

Listing 8: Identified and Rewritten rules

As a final step, the rules are further pruned to simplify the resulting rules. A process similar to the C-Sprite algorithm [15] is used to prune rules such that intermediate results that do not directly contribute to answering the query are removed, this results resuls in further pruned rule:

```
R3'': {?x a Observation, ?x madeBySensor :sensorX.}
                => {?x a ComfortObservation}
```

Listing 9: Pruned rules

If only a small number of rules remain, these can be directly injected in the query, which allows to extract the following query:

```
    ?obs a :Observation;
        madeBySensor :sensorX;
        hasSimpleResult ?val.
```

Listing 10: Query2-Edge: rewritten query based on optimized rules.

This query can be pushed to the Edge, without the need for any additional inference rules are combination with static data.

4.4. Implementation

We have implemented the rewriting algorithms on top of our RoXi reasoner [16][2], i.e. a modular reasoner that focuses on reactive reasoning applications. RoXi is written in Rust and thus ideal to run both on low resource devices, as found at the Edge and in the cloud, while using the same code base. We reused the forward and backward chaining algorithms from RoXi and built the rewriting algorithms presented in Section 1 and Section 2 on top. The resulting algorithms can be found on our Github page[3].

[2]https://github.com/pbonte/roxi
[3]https://github.com/pbonte/SHAROQ

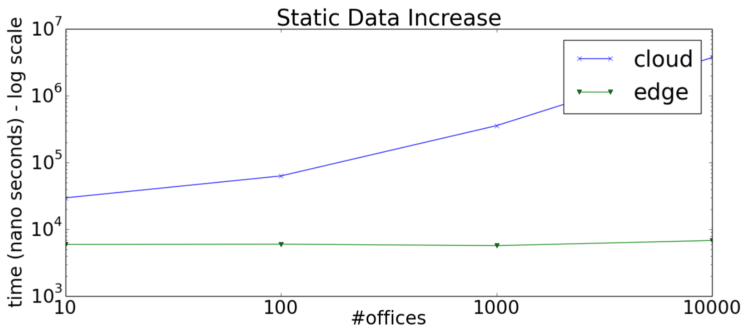

Figure 2: Scalability evaluation when increasing the static data, i.e. the number of offices.

5. Evaluation

For the evaluation, we focus on the scalability in function of the response time in a cloud environment compared to offloading to the Edge. The evaluation is naive in the sense that we do not take network delays, or any heterogeneity in the available resources at the Edge into account. The main focus is demonstrating that the proposed algorithms can rewrite queries/rules and inject the needed static data directly inside the queries/rules such that the query evaluation can be directly done on the data produced at the Edge and thus improving scalability. We focus on three scalability investigations: 1) increasing the static data, 2) increasing the number of concurrent observations and 3) increasing the reasoning complexity. All evaluations, both cloud and Edge, were done on a MacBook Pro, running Big Sur, with 2,7 GHz Quad-Core Intel Core i7 CPU and 16GB of RAM.

5.1. Static Data Increase

In order to increase the size of the static data, we have generated buildings with increasing number of rooms. Each room has a number of properties, the rooms it is adjacent to, the function of the room, its name, etc. Note that even though we generated data for this evaluation, all data is inspired on real building data [17]. The generation process is to simplify the scalability evaluation. We use *Query1* for the cloud evaluation and the rewritten query *Query1-Edge* from Section 4.2 is used for the Edge evaluation. Figure 2 shows the response time in nano seconds, in log scale, when increasing the number of offices. The figure compares the query evaluation time for a cloud scenario compared to the query evaluation time for the rewritten query at the Edge. We can clearly see that the Edge scenario is not subject to any performance degradation due to the increase of the static data set. This is because the subset needed static data is injected directly inside the Edge query. We note that this is the response time for a single Edge device. However, as the evaluation at the Edge can be done in parallel when multiple Edge devices are present, the comparison still holds.

5.2. Sensor Observations Increase

This evaluation investigates the influence of installing more sensors that are producing more observations. In RSP, we typically use some kind of windowing to cut the streams in processable chunks. As more sensors are being installed and more observations are being transmitted to

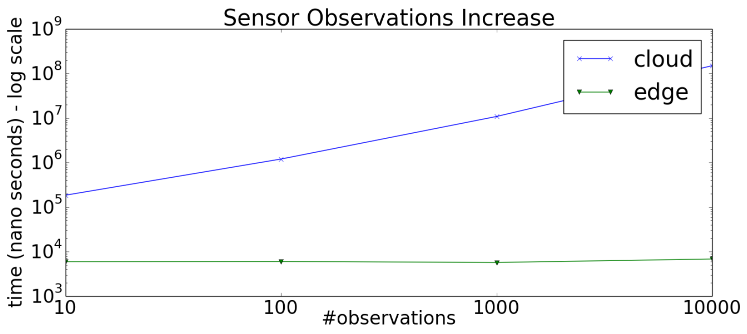

Figure 3: Scalability evaluation when increasing the number of sensor observations.

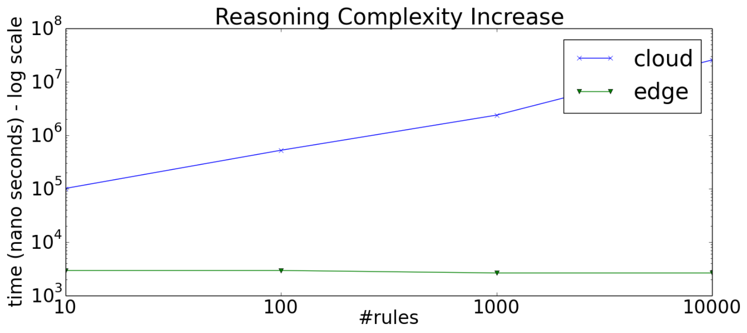

Figure 4: Scalability evaluation when increasing the number of reasoning rules.

the cloud, these windows will contain larger amounts of observations, resulting in performance degradation. In this evaluation we scale the number of sensors and transmitted observations and compare the query evaluation time for the cloud and Edge scenario. We will use *Query1* and *Query1-Edge* again for this evaluation. Figure 3 shows the response time in nano seconds, in log scale, when increasing the number of concurrent observations. We can clearly see that the Edge scenario is in this case also not subject to any performance degradation, because it does not require to combine the observations from all sensors and can filter the data produced by a single sensor. We note that we assumed that each sensor would have its own gateway that is able to perform the offloaded processing.

5.3. Reasoning Complexity Increase

The last evaluation investigates the influence of increasing the reasoning complexity by increasing the number of rules that need to trigger in order to correctly evaluate *Query2*. This part of the evaluation focuses on the rewriting of the rules, as presented in Section 4.3. In order to increase the number of rules that trigger, we artificially generate a hierarchy of concepts that is evaluated on the data from the sensors. This is done by injecting a hierarchy of variable length between the *LoudnessObservation* and *ComfortObservation* concepts. As the latter concept is being queried in *Query2*, the whole hierarchy needs to be evaluated, regardless of using forward or backward chaining. Figure 4 shows the response time in nano seconds, in log scale, when increasing the number the number of reasoning rules. The figure compares the reasoning and query evaluation time for a cloud scenario compared to the reasoning and query evaluation time for the rewritten rules/query at the Edge. Note that for this evaluation, we also measure the materialization time, i.e. the time to evaluate all the rules and store their results. The more rules need to evaluated and store the results of these rules, the longer the processing time. This is clearly visible for the cloud scenario. For the Edge scenario, we can optimize the rules and prune all intermediate steps that are not contributing to answering the query. Furthermore, necessary static data is injected inside the remaining rules, such that the static data is not necessary any more.

Pruning the rules and injecting the static data inside the rules/queries makes the evaluation less prone to scalability in terms of static data, concurrent observations or reasoning complexity.

6. Related Work

Various approaches to realize the vision of CR have been proposed in the past. StreamRule [18] proposes a layered approach that combines RSP with Answerset Programming, however, there is no connection between the layers and each layer needs to be manually defined. In the CityPulse project [19], StreamRule is extended with Complex Event Processing (CEP) capabilities, however, inherits the same drawbacks as StreamRule, while the CEP rules cannot be defined declaratibly and require a programming effort. StreamingMASSIF is the first platform that really integrates all layers into a single declarative language [4], however, even though the language allows to target the different layers, each layer still has to be defined manually.

In RSP, StreamQR [20] proposes a query rewriting approach that is able to inject the domain knowledge directly into the query, resulting in very larges queries with large numbers of UNIONs, which can still be very expensive to evaluate. Compared to our approach, StreamQR takes a top-down approach and does not prune the domain knowledge that needs to be injected in the query based on a bottom-up analysis of the generated data. Our bottom-up analysis allows to further prune the needed domain knowledge such that that the evaluation at the Edge can be as efficient as possible. Furthermore, our approach also investigates the injection of static data into the query next to the injection of domain knowledge, eliminating many of the stream and static joins that would otherwise need to happen in the cloud.

C-Sprite [15] is a hybrid rewriting approach that analyses a registered query and optimizes the rules in the domain knowledge that represent concept hierarchies. Similar to our work, C-Sprite does a top-down analysis of the queries and prunes part of the rules. However, C-Sprite only takes simple hierarchies (subclassOf) definitions into account, cannot inject any static data and also disregards the bottom-up analysis making it inefficient for Edge offloading.

Nguyen-Duc et al. [21] propose a Query Federation procedure for Edge processing. This is done by incorporating the exact query that needs to be evaluated at the Edge as a nested query in a larger CQELS-QL query. The focus of this work is on the automated federation of the defined Edge queries based on the stream descriptions and does not allow to rewrite the queries to take domain knowledge and static data into account as we propose here.

In the realm of Semantic Web of Things, many efforts have been conducted to semantically describe, discover and search Sensors and sensor data [22]. While their focus is typically on annotation, query and (in some cases) processing, their focus has not been on pushing generic processing capabilities down to the Edge as we propose in our work.

To conclude, little work has been done to automatically translate high-level queries to specific queries that can be automatically offloaded to the Edge to enable generic Edge processing.

7. Conclusion & Future Work

In this paper, we present a first automated CR approach that allows to automatically rewrite high-level queries to specific ones that can be evaluated directly at the Edge. These high-level queries typically combine multiple sensors streams with static data and domain knowledge (in the form of rules). We propose to employ a top-down and bottom up analysis of both the registered queries and the annotation of the sensor data. These high-level queries can be optimized and rewritten such that the reasoning-phase and enrichment with static data can be bypassed as much as possible. The later is realized by rewriting the high-level queries to Edge specific queries that contain the needed static data and domain knowledge to evaluate the query. This enables privacy, low-latency and autonomous processing directly at the Edge. Furthermore, as less data needs to be sent to the cloud, higher throughput at the cloud and less network congestion are obtained as well.

In our future work, we will fully formalize our work and prove that correctness and completeness is maintained after splitting up the high-level queries into multiple smaller ones. We will also investigate more complex queries, i.e. containing nested queries, union or optional operators. Furthermore, we aim to investigate the other required Edge processing building blocks that were disregarded in this work, such as discovery and description of streams and agents, offloading and resource matching to place the processing modules at right agents taking resources and Qualities of Service into account.

Our work brings us one step closer to realizing the CR vision which aligns directly with the Edge processing paradigm. This allows to specify high-level queries over the cloud and Edge network, while the CR platform optimizes the various querying and reasoning operators across the network to realize efficient and fully generic question answering at the Edge.

Acknowledgement This work is funded by Pieter Bonte's postdoctoral fellowship of Research Foundation Flanders (FWO) (1266521N) and by SolidLab Vlaanderen (Flemish Government, EWI, and RRF project VV023/10).

References

[1] T. Qiu, et al., Edge computing in industrial internet of things: Architecture, advances and challenges, IEEE Communications Surveys & Tutorials 22 (2020) 2462–2488.

[2] H. Lin, S. Zeadally, Z. Chen, H. Labiod, L. Wang, A survey on computation offloading modeling for edge computing, Journal of Network and Computer Applications 169 (2020) 102781.

[3] P. Bonte, F. D. Turck, F. Ongenae, Bridging the gap between expressivity and efficiency in stream reasoning: a structural caching approach for iot streams, Knowledge and Information Systems (2022) 1–35.

[4] P. Bonte, R. Tommasini, E. Della Valle, F. De Turck, F. Ongenae, Streaming massif: cascading reasoning for efficient processing of iot data streams, Sensors 18 (2018) 3832.

[5] P. Bonte, F. Ongenae, F. De Turck, Subset reasoning for event-based systems, IEEE Access 7 (2019) 107533–107549.

[6] D. Dell'Aglio, E. Della Valle, F. van Harmelen, A. Bernstein, Stream reasoning: A survey and outlook, Data Science 1 (2017) 59–83.

[7] H. Stuckenschmidt, S. Ceri, E. Della Valle, F. Van Harmelen, Towards expressive stream reasoning, in: Dagstuhl Seminar Proceedings, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2010.

[8] T. Delva, B. De Smedt, S. Min Oo, D. Van Assche, S. Lieber, A. Dimou, Rml2shacl: Rdf generation taking shape, in: Proceedings of the 11th on Knowledge Capture Conference, 2021, pp. 153–160.

[9] P. Bonte, R. Tommasini, Streaming linked data: A survey on life cycle compliance, Journal of Web Semantics (2023) 100785.

[10] P. Heyvaert, B. De Meester, A. Dimou, R. Verborgh, Declarative Rules for Linked Data Generation at your Fingertips!, in: Proceedings of the 15th ESWC: Posters and Demos, 2018.

[11] R. Tommasini, Y. A. Sedira, D. Dell'Aglio, M. Balduini, M. I. Ali, D. Le Phuoc, E. Della Valle, J.-P. Calbimonte, Vocals: Vocabulary and catalog of linked streams, in: ISWC, Springer, 2018, pp. 256–272.

[12] A. Mallea, M. Arenas, A. Hogan, A. Polleres, On blank nodes, in: The International Semantic Web Conference, Springer, 2011, pp. 421–437.

[13] D. F. Barbieri, D. Braga, S. Ceri, E. D. VALLE, M. Grossniklaus, C-sparql: a continuous query language for rdf data streams, International Journal of Semantic Computing 4 (2010) 3–25.

[14] O. Hartig, C. Bizer, J.-C. Freytag, Executing sparql queries over the web of linked data, in: International semantic web conference, Springer, 2009, pp. 293–309.

[15] P. Bonte, R. Tommasini, F. De Turck, F. Ongenae, E. D. Valle, C-sprite: efficient hierarchical reasoning for rapid rdf stream processing, in: DEBS, 2019, pp. 103–114.

[16] P. Bonte, F. Ongenae, Roxi: a framework for reactive reasoning, in: ESWC 2023, Springer, 2023.

[17] J. Vanhaeverbeke, E. Deprost, P. Bonte, M. Strobbe, J. Nelis, B. Volckaert, F. Ongenae, S. Verstockt, S. Van Hoecke, Real-time estimation and monitoring of covid-19 aerosol transmission risk in office buildings, Sensors 23 (2023) 2459.

[18] A. Mileo, A. Abdelrahman, S. Policarpio, M. Hauswirth, Streamrule: a nonmonotonic stream reasoning system for the semantic web, in: Web Reasoning and Rule Systems, Springer, 2013, pp. 247–252.

[19] D. Puiu, et al., Citypulse: Large scale data analytics framework for smart cities, IEEE Access 4 (2016) 1086–1108.

[20] J.-P. Calbimonte, J. Mora, O. Corcho, Query rewriting in rdf stream processing, in: ESWC, Springer, 2016, pp. 486–502.

[21] M. Nguyen-Duc, A. Le-Tuan, J.-P. Calbimonte, M. Hauswirth, D. Le-Phuoc, Autonomous rdf stream processing for iot edge devices, in: Joint International Semantic Technology Conference, Springer, 2020, pp. 304–319.

[22] Y. Zhou, S. De, W. Wang, K. Moessner, Search techniques for the web of things: A taxonomy and survey, Sensors 16 (2016) 600.