DIVIDE: Adaptive Context-Aware Query Derivation for IoT Data Streams

Mathias De Brouwer[1,3][0000−0001−8769−6861], Dörthe Arndt[2], Pieter Bonte[1][0000−0002−8931−8343], Filip De Turck[1][0000−0003−4824−1199], and Femke Ongenae[1][0000−0003−2529−5477]

[1] Ghent University – imec, IDLab, Department of Information Technology iGent Tower, Technologiepark-Zwijnaarde 126, B-9052 Ghent, Belgium
[2] Ghent University – imec, IDLab, Department of Electronics and Information Systems – AA Tower, Technologiepark-Zwijnaarde 122, B-9052 Ghent, Belgium
[3] mrdbrouw.DeBrouwer@UGent.be

Abstract In the Internet of Things, it is a challenging task to integrate & analyze high velocity sensor data with domain knowledge & context information in real-time. Semantic IoT platforms typically consist of stream processing components that use Semantic Web technologies to run a set of fixed queries processing the IoT data streams. Configuring these queries is still a manual task. To deal with changes in context information, which happen regularly in IoT domains, queries typically require reasoning on all sensor data in real-time to derive relevant sensors & events. This can be an issue in real-time, as expressive reasoning is required to deal with the complexity of many IoT domains. To solve these issues, this paper presents DIVIDE. DIVIDE automatically derives queries for stream processing components in an adaptive, context-aware way. When the context changes, it derives through reasoning which sensors & observations to filter, given the context & a use case goal, without requiring any more reasoning in real-time. This paper presents the details of DIVIDE, and performs evaluations on a healthcare example showing how it can reduce real-time processing times, scale better when there are more sensors & observations, and can run efficiently on low-end devices.

Keywords: Internet of Things · Context-aware query derivation · Reasoning · RDF stream processing · N3.

1 Introduction

In the Internet of Things (IoT), there exists a large collection of internet-connected devices and sensors. IoT-enabled sensors constantly generate data. The advantage of the IoT is that this data can be easily integrated and combined with existing domain knowledge and context information. In this way, devices and applications are able to process and analyze the combined sensor & context data in order to perform context-aware monitoring of the environment [18].

The data generated by IoT devices is typically voluminous, heterogeneous, and has a high velocity [1]. As such, it is a challenging task to integrate and analyze this data on the fly, in order to extract meaningful insights and actuate on it.

To deal with these challenges, Semantic Web technologies can be deployed [18]. Typical semantic IoT platforms consist of one or more streaming components that use queries to continuously process the generated data streams. The heterogeneous data is modeled in ontologies, and existing stream reasoning techniques are used to perform the advanced data stream processing [12].

In different IoT applications domains, relevant information about the context regularly changes. For example, in healthcare, the information contained in a patient's Electronic Health Record (EHR) continuously evolves throughout a patient's hospital stay. In the smart cities domain, changing contextual information heavily impacts applications such as traffic management. This information updates on a regular basis, as it includes unavailable traffic routes due to road works, current music or sporting events, whether it is a holiday or not, etc.

The application context has an influence on how the components of an IoT platform process and actuate on the generated sensor data. This context directly impacts the sensors of which the observations should be monitored in detail by the streaming components, and possibly filtered for further processing by other platform components. For example, a patient's diagnosis implies which sensors in his/her hospital room require special attention, while blocked traffic roads impact which intersection traffic streams should be closely monitored.

In existing semantic IoT platforms, the configuration of queries that run on the streaming components is a manual, labor-intensive task. To deal with context changes, two approaches are possible. The first approach uses fixed generic queries. These queries reason on all sensor observations, to derive in real-time which are the relevant sensors, and which observations of these sensors should be filtered, given the current context. In this way, the queries should not be updated when the context changes. However, ontologies in IoT domains are typically complex. This requires expressive reasoning, which is computationally expensive [14]. This might imply problems in a real-time system, especially when the component monitors many sensors, or when a high query frequency is required. The second approach is to run queries that filter the individual sensors that are relevant with the given context. These queries require less to no real-time reasoning, which solves the issues of the first approach. However, designing and reconfiguring them should be done manually upon each context change. This is highly impractical and infeasible if this needs to be maintained for a full-fledged IoT network, such as in a hospital. Hence, this approach is almost never applied.

To solve the presented issues, this paper presents the DIVIDE system. In general, DIVIDE can be seen as an additional component for a semantic IoT platform, which allows to derive relevant queries for the platform's streaming components, based on the context and a defined use case goal. These queries are derived by performing reasoning when the application context changes. Hence, complex ontology concepts can be filtered in real-time from the observations of the relevant sensors, without the need to perform any real-time reasoning on all data. As DIVIDE is able to adaptively derive the individual, newly relevant queries when the context changes, it actually removes the complexity issues of the first approach by applying the second approach in an automated way.

The remainder of this paper is organized as follows. In Section 2, related work is discussed. Section 3 explains all details of the DIVIDE system. The set-up and results of the system evaluation are presented in Sections 4 & 5. These results are further discussed in Section 6. Finally, Section 7 concludes the paper.

2 Related Work

To deal with the presented challenges of the IoT, multiple platforms exist that adopt different Semantic Web technologies [22,6,13,17]. Most of these platforms consist of both stream processing components and semantic reasoning components. They all use different existing technologies for these components, but all have in common that the configuration of queries on the streaming components is not automated in an adaptive and context-aware way.

Stream Reasoning is the research area that focuses on the adoption of Semantic Web technologies for streaming data [12]. Different RDF Stream Processing (RSP) engines exist [19], such as C-SPARQL, CQELS, and Yasper. These engines require the registration of a set of fixed queries, which are used to continuously filter the streaming data in real-time. Recently, a unifying semantic query model, RSP-QL, has been designed by the W3C RSP Community Group [11].

To infer new knowledge from the data, RSP engines try to incorporate semantic reasoning techniques. The complexity of these techniques depends on the expressivity of the underlying ontology [14]. Different ontology languages exist, ranging from RDFS to OWL 2 DL, with increasing expressivity.

Existing RSP engines support at most RDFS reasoning [19]. To perform more expressive reasoning, dedicated semantic reasoners exist. Examples are RDFox [15] and VLog [20], which are OWL 2 RL reasoners. OWL 2 RL contains all constructs that can be expressed by simple Datalog rules. By design, these engines are not able to handle streaming data. By adopting techniques from RSP engines such as windowing, this could be possible. However, reasoning complexity may be too high to provide real-time answers to high velocity data streams [12]. StreamQR [7] is an alternative approach that rewrites continuous RSP queries to multiple parallel queries, supporting ontologies expressed in the $\mathcal{ELHIO}$ logic.

3 DIVIDE System

The goal of DIVIDE is the context-aware, adaptive derivation of continuous queries running on the stream processing components of a semantic IoT platform, filtering (possibly complex) ontology concepts from the IoT data streams, without requiring real-time reasoning. This section details the DIVIDE system step by step, but first starts with the introduction of a running example, that will be used throughout the remainder of this paper.

3.1 Running Example

In a pervasive health context, smart hospitals of the future consist of ambient-intelligent care rooms. These rooms are equipped with many IoT enabled devices,

which contain sensors that continuously generate data. Examples are environmental sensors (e.g., light and sound sensors) and body sensors (e.g., for heart rate). Moreover, the existence of intelligent smart home devices allows to control and automate the lighting, room temperature, and much more.

A smart hospital typically has a set of medical domain knowledge which is spread out in a back-end database network. This includes, among others, known diagnoses and corresponding medical symptoms, i.e., sensitivities. For example, it may state that a concussion diagnosis implies sensitivities to light and sound, with a maximum exposure to values of respectively 170 lumen and 30 decibels. Moreover, all information the hospital knows about a patient, e.g., his diagnosis, is contained in the patient's EHR. These EHRs are also stored in this database network, as well as other context information about room set-up, care staff etc.

Consider a semantic IoT platform set-up in a smart hospital that consists of a back-end database network, and a local processing device in each room. Assume that the domain knowledge & context information, including EHRs, is available from a knowledge base on a central server, accessing this database network.

To filter all data generated by the sensors in the room, each local device runs an RSP engine. The relevant sensors that should be monitored in each room, and thus the relevant continuous RSP queries, depend fully on the context: which patient is accommodated in the room, what his diagnosis is, what sensitivities this diagnosis implies, and what thresholds are associated to these sensitivities. Moreover, changes to the context occur frequently. Examples are updates to a patient's EHR, or changes in room occupation. From the viewpoint of a hospital room, this may imply other relevant queries. Therefore, to automatically and adaptively derive the relevant RSP queries based on the context, DIVIDE can be used. Specifically, DIVIDE will look for all queries that filter observations which require a certain action, corresponding to a crossed threshold. This action will imply the automatic control of *local* devices influencing the involved property. Locally handling the action and propagating it into the system, e.g., sending it to the back-end to notify a nurse of the event, is left out of scope for this example.

3.2 Building Blocks

DIVIDE is built upon several existing building blocks, which are detailed below.

Ontology For the running example, the medical domain knowledge is described by the `CareRoomMonitoring` ontology of the ACCIO continuous care ontology [16], including all imports of other ACCIO ontologies and external ontologies such as SAREF [10], SOSA and SSN [8].[4] This ACCIO ontology contains a pattern that links observations with certain types of actions. It defines four generic ontology classes: `Observation`, `Symptom`[5], `Fault` and `Action`. To illustrate how these are linked, consider the following ontology definitions:

[4] The corresponding ontology files are available at https://github.com/IBCNServices/DIVIDE/tree/master/saw2019/ontology. This page also contains a figure and additional explanation about the described ontology observation pattern.

[5] Note the difference between `Symptom` (e.g. `ThresholdSymptom`) and `MedicalSymptom`.

```
LightIntensityAboveThresholdFault ⊑ Fault
LightIntensityAboveThresholdFault ≡ Observation and
  (hasSymptom some LightIntensityAboveThresholdSymptom) and
  (madeBySensor some (isSubsystemOf some (hasLocation some
    (isLocationOf some (
      (hasDiagnosis some (hasMedicalSymptom some SensitiveToLight))
      and (hasRole some PatientRole))))))
LightIntensityAboveThresholdSymptom ≡
  ThresholdSymptom and (forProperty some LightIntensity)
HandleHighLightInRoomAction ⊑ AboveThresholdAction
HandleHighLightInRoomAction ≡ LightIntensityAboveThresholdFault
  and (madeBySensor some (isSubsystemOf some (hasLocation some
    (isLocationOf some LightingDevice))))
HandleHighLightInRoomAction ≡
  AboveThresholdAction and (forProperty some LightIntensity)
```

Logic and Reasoner DIVIDE uses the rule-based Notation3 Logic (N_3) [5]. N_3 is a superset of RDF/Turtle [9], which means that the RDF/Turtle representation of the ACCIO ontology is valid N_3. A reasoner supporting N_3 can reason within the OWL profile OWL 2 RL [14]. DIVIDE uses the EYE reasoner, which runs in a Prolog virtual machine [21].

To run the EYE reasoner, a goal can be defined that tells EYE for which RDF/Turtle triples it should look for evidence. This goal is defined as a rule, which serves as a filter for EYE. When EYE reasons on its N_3 inputs, it constructs a proof where this rule is the last rule applied. Within DIVIDE, the reasoner goal should specify the ontology concept that the eventual queries should filter, which in real-time would require reasoning to derive from an `Observation`.

For the running example, the goal is to filter observations which require an action corresponding to a crossed threshold. Hence, it is defined as follows.

```
{ ?x a AboveThresholdAction . } => { ?x a AboveThresholdAction . } .
```

3.3 Sensor Query Rule

To use DIVIDE to derive the queries that need to run on a stream processing engine, a generic formalism has been designed. This formalism defines the generic pattern of such a query, together with information on when and how to instantiate it. Each such description is called a *sensor query rule*.

The presented formalism builds further on SENSdesc [3], which is the result of previous research. The theoretical SENSdesc work has initiated the idea and format to describe sensor queries in such a way that they can be combined with formal reasoning to retrieve queries contributing to a user defined goal. In this paper, this format is further generalized and improved, in order to be practically usable for generic use cases in DIVIDE.

A sensor query rule consists of three parts. To explain this with an example, consider the sensor query rule for the running example as defined in Listing 1.

Relevant Context In the antecedence of the rule, the context in which the query might become relevant is described in generic fashion. In Listing 1, this part is described in lines 1–10. It looks for a patient who has a certain diagnosis

Listing 1. Sensor query rule for the running example. Prefix declarations are omitted.

```
1  { ?p DUL:hasRole [ a RoleCompetenceAccio:PatientRole ] ;
2       DUL:hasLocation ?l ;
3       CareRoomMonitoring:hasDiagnosis [
4         CareRoomMonitoring:hasMedicalSymptom [
5           SSNiot:hasThreshold [
6             DUL:hasDataValue ?threshold ;
7             SSNiot:isThresholdOnProperty [ a ?prop ] ] ] ] .
8    ?sensor a sosa:Sensor ; sosa:observes [ a ?prop ] ;
9            SSNiot:isSubsystemOf [ DUL:hasLocation ?l ] .
10   ?prop rdfs:subClassOf sosa:ObservableProperty . }
11 =>
12 { _:q a sd:Query ; sd:pattern :pattern-1 ;
13       sd:inputVariables (("?th" ?threshold) ("?s" ?sensor) ("?prop" ?prop)) ;
14       sd:outputVariables (("?v" _:v) ("?o" _:o)) .
15
16   _:o a sosa:Observation ; sosa:madeBySensor ?sensor ; sosa:hasResult
17         [ a SSNiot:QuantityObservationValue ; DUL:hasDataValue _:v ] ;
18       SSNiot:hasSymptom [
19         a SSNiot:ThresholdSymptom ; ssn:forProperty [ a ?prop ] ] . } .
20
21 :pattern-1 a sd:QueryPattern ; sh:prefixes :prefixes ; sh:construct """
22     CONSTRUCT { ?o a CareRoomMonitoring:AboveThresholdAction ;
23                    ssn:forProperty ?prop . }
24     FROM NAMED WINDOW :win ON <http://idlab.ugent.be/grove>
25       [RANGE PT1S TUMBLING]
26     WHERE { WINDOW :win {
27       ?o a sosa:Observation ; sosa:madeBySensor ?s ;
28          sosa:hasResult [ DUL:hasDataValue ?v ] ; sosa:resultTime ?t ;
29          General:hasId [ General:hasID ?id ] .
30       FILTER (xsd:float(?v) > xsd:float(?th)) } }
31     ORDER BY DESC(?t) LIMIT 1""" .
```

that is linked to a `MedicalSymptom`. This `MedicalSymptom` (sensitivity) needs to be linked with a threshold on a specific property, e.g., `LightIntensity`. If there exists a sensor in the same room that is observing that specific property, the query described in the next step might be relevant.

Generic Query In the first part of the rule's consequence, the generic query is described. This query is written in RSP-QL format, and is defined using the SHACL standard. In addition, the query's input variables are defined, which need to be instantiated to make the query specific for the relevant context. This will happen through the rule evaluation during the query derivation.

In Listing 1, lines 12–14 and 21–31 describe the generic query. Lines 22–31 describe the actual RSP-QL query that should run on an RSP engine. The WHERE clause specifies that the query filters observations made by a *certain* sensor (`?s`), that are higher than a *certain* threshold (`?th`). For any filtered observation individual, new triples are constructed specifying that it is of type `AboveThresholdAction` linked to a *certain* property (`?prop`). Note that this class exactly matches the class specified in the reasoner's goal in Section 3.2. This makes sense, as the goal is used to specify the ontology concepts that the queries need to filter. If this certain linked property is for example `LightIntensity`, it follows from the ontology definitions in Section 3.2 that this is equivalent to a `HandleHighLightInRoomAction`. In addition to the RSP-QL query, line 13

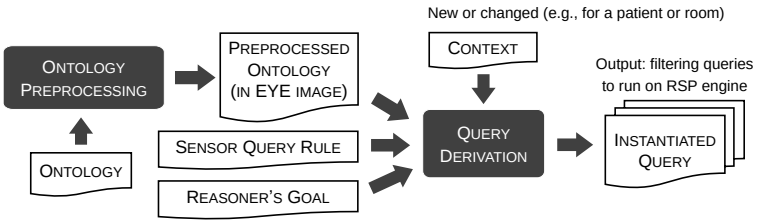


Figure 1. Overview of the components, inputs and outputs of the DIVIDE system

of the sensor query rule defines that the specific sensor, threshold and property variables should be substituted into the query to instantiate it. During the query derivation process, the actual values for these variables will depend on the matching query context defined in the rule's antecedence.

Ontology Consequences The second part of the rule's consequence describes the effects of a query result. A result is obtained when the rule's antecedence holds, and an instantiated version of the query actually filters *an* observation. This part defines the consequences of this observation in terms of the ontology.

In Listing 1, this part is in lines 16–19. If *a* sensor observation above a defined threshold is filtered, represented by the blank node `_:o`, this `Observation` is linked to a `ThresholdSymptom` for the considered property. For `LightIntensity`, this is equivalent with a `LightIntensityAboveThresholdSymptom`.

Note that the sensor query rule of the running example is generic in the sense that it can be used for any property that is threshold-based, as all steps use the variable `?prop`. In this way, the rule should only be defined once, in order to be used generically in a hospital context. Moreover, when defining the context for the query derivation, it should not be explicitly stated that a patient is sensitive to this property. By defining the diagnosis of a patient, the relevant ontology definitions and sensor query rule will enable a rule-based reasoner to automatically derive the associated sensitivities.

3.4 Context-Aware Query Derivation with DIVIDE

Given the generic sensor query rule defined in the previous section, the DIVIDE system can be used to automatically derive relevant RSP queries in a context-aware fashion. Figure 1 shows the different system components, including their inputs and outputs. Two components can be distinguished: apart from the actual query derivation, DIVIDE can first be used to preprocess the ontology.

Ontology Preprocessing The domain ontology is considered not to change throughout the lifetime of the application, in contrast with the context data. Therefore, this ontology can be preprocessed upfront by DIVIDE using the EYE reasoner, in order to speed up the actual query derivation process.

The preprocessing process consists of three steps. First, an N_3 copy of the full ontology is created. Second, specialized ontology-specific rules are created from the original rules taken from the OWL 2 RL profile description[6]. Starting the EYE reasoning process from these specialized rules will reduce the computational complexity of the reasoning [2]. Third, an image of the EYE reasoner, which has already loaded the ontology and specialized rules, is compiled within Prolog. In this way, they do not need to be loaded into the reasoner each time it is called.

Query Derivation The DIVIDE query derivation process is called each time a part of the context in the central knowledge base changes, e.g., the context related to a specific patient or room in the hospital use case.

The first process step starts from the EYE Prolog image compiled in the preprocessing step. It reads in the considered context, the sensor query rule and the reasoner goal. Given these inputs, EYE constructs a proof that derives all instances of the ontology concept defined in the goal. To get from the context to the goal, the evaluation of the sensor query rule is crucial. If the antecedence of the rule holds one or multiple times, it means that the rule *can* be evaluated. If the triples in the rule's consequence also allow the reasoner to derive the ontology concept defined in the goal, the rule *will* actually be evaluated for the antecedence's context and will appear in the proof. This means that the generic query will also be evaluated, with the query's input variables being instantiated.

Once the proof has been constructed by EYE, the second step looks for all queries in the proof. This is done by a simple reasoning step in EYE, looking for all proof steps that include the generic pattern in lines 12–13 of Listing 1.

In a third and final step, the system transforms the generic RSP-QL query, defined in the sensor query rule, into an instantiated query. This happens for each pattern extracted from the proof in step 2, through another forward reasoning step with EYE. As such, the system outputs all queries that filter the ontology concept defined in the goal: if this query filters an observation, it can immediately be concluded that this observation is an instance of this concept, without the need to perform the reasoning step anymore. This holds as long as that part of the context does not change. When it does change, the DIVIDE query derivation process should run again to (possibly) update the relevant queries.

Considering the running example, the goal of the EYE reasoner is to derive instances of the concept `AboveThresholdAction`. To do so, it follows from the definitions in Section 3.2 that the reasoner will – among others – try to look for individuals of the subclass `HandleHighLightInRoomAction`. To derive that an `Observation` individual is of this type, the reasoner requires – among other triples – a `LightIntensityAboveThresholdSymptom` linked to the `Observation` via the `hasSymptom` object property. This is equivalent to the second part of the consequence of the sensor query rule in Listing 1 (lines 16–19). Hence, if all other requirements are fulfilled to derive an instance of `HandleHighLightInRoomAction`, the rule will be evaluated for each situation in the input context where the antecedence holds for `?prop` being `LightIntensity`.

[6] https://www.w3.org/TR/owl2-profiles/#OWL_2_RL

For example, in `CareRoomMonitoring`, the `Concussion` diagnosis is linked to a sound sensitivity with threshold 30, and a light sensitivity with threshold 170. Consider the context of a hospital room consisting of a patient diagnosed with concussion, containing a light sensor A0, and at least one lighting device. For this context, the output of the query derivation process will contain a query filtering observations of sensor A0 higher than 170. If the room also contains a sound sensor A1 and at least one device influencing the room's sound level, the output will also contain a query filtering observations of sensor A1 above 30. If a new patient is brought into the room that has a different diagnosis with other sensitivities, rerunning the query derivation process will no longer output these queries, but others depending on the exact context and ontology definitions.

4 Evaluation Set-up

In this section, the DIVIDE system is evaluated. Three evaluations are performed, which all consider the use case and ontology of the running example described in Section 3.1. The context considered in each evaluation is one single-person hospital room, containing a patient diagnosed with concussion.

4.1 DIVIDE Performance Evaluation

To assess the performance of the DIVIDE system presented in Section 3.4, the duration of the ontology preprocessing and query derivation processes is measured. The evaluation considers the described evaluation context, with 10 sensors in the concussion patient's room, including one light sensor and one sound sensor. The reasoner goal and sensor query rule are as described in Section 3.2 and Listing 1. Given these inputs, two queries will be outputted: one for the light sensor and one for the sound sensor.[7] The evaluation is performed on a device with a 2800 MHz quad-core Intel Core i5-7440HQ CPU and 16 GB DDR4-2400 RAM.

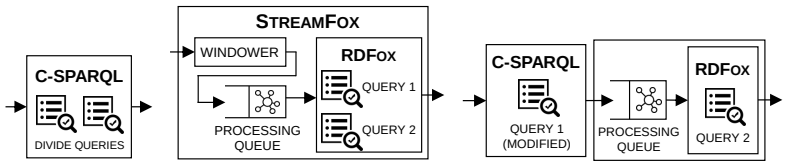
4.2 Comparison of DIVIDE with Real-Time Reasoning Approaches

The DIVIDE approach allows the detection of complex events in the sensor stream, without performing real-time reasoning. Alternatively, one could use other traditional approaches, which do require real-time reasoning. Therefore, the real-time filtering approach used in DIVIDE is compared with two real-time reasoning approaches, both using the same reasoning profile as DIVIDE, i.e., OWL 2 RL. Both approaches use RDFox, as this is known as one of the fastest OWL 2 RL reasoning engines [15]. For each approach, the goal is to detect any `AboveThresholdAction` individual in the sensor stream.

The following set-ups are considered[8], which are visualized in Figure 2:

[7] All evaluation files (scripts, inputs & outputs) are available at https://github.com/IBCNServices/DIVIDE/tree/master/saw2019/evaluations/divide-performance.

[8] The queries running on each set-up are available at https://github.com/IBCNServices/DIVIDE/tree/master/saw2019/evaluations/real-time-comparison. For the windowing, Esper (https://www.espertech.com/esper) is used.



(a) DIVIDE (1) (b) StreamFox (2) (c) C-SPARQL & RDFox (3)

Figure 2. Overview of the compared evaluation set-ups

1. **DIVIDE approach using C-SPARQL without reasoning:** regular C-SPARQL engine [4]. No ontology or context data is loaded into the engine, and no reasoning is performed during the continuous query evaluation. The two RSP-QL queries outputted by DIVIDE (see Section 4.1) are translated to the C-SPARQL syntax and running continuously on the C-SPARQL engine, each with their own logical tumbling window of 1 second.
2. **StreamFox:** streaming version of RDFox. Consists of one engine that pipes Esper for windowing with RDFox for reasoning, via a processing queue. Initially, the ontology and context data are loaded into the data store of the RDFox engine, and a reasoning step is performed. Two generic SPARQL queries are registered. Query 1 looks for observations above a threshold within the valid context, and creates a `ThresholdSymptom` for them; it can be seen as the SPARQL alternative for the generic sensor query rule in Listing 1. Query 2 retrieves any (derived) `AboveThresholdAction` individual. Windowing is performed with a logical tumbling window of 1 second. On each window trigger, the window content is added as one event to a processing queue. When available, RDFox takes an event from the queue, incrementally adds it to the RDFox data store (i.e., it performs incremental reasoning with the event scheduled for addition), and executes the registered queries in order. If query 1 yields a non-empty result, this is incrementally added to the store, before query 2 is executed. Finally, RFDox performs incremental reasoning with the event scheduled for deletion (i.e., incremental deletion).
3. **C-SPARQL piped with (non-streaming) RDFox:** Initially, the RDFox data store contains the ontology and context data, and a reasoning step is performed. For the C-SPARQL engine, query 1 of set-up 2 is modified to run as a continuous C-SPARQL query on a logical tumbling window of 1 second of the observation stream, and on the ontology and context triples. C-SPARQL does not perform reasoning during the query evaluation. It sends each query result to the event stream of the non-streaming RDFox engine, which adds it to a processing queue. Upon processing time, it incrementally adds the event to the data store, executes query 2 of set-up 2, and incrementally deletes the event from the data store.

The amount of sensors in the context depends on the evaluated scenario. During each scenario run, every sensor produces one observation per second, for a duration of 25 seconds. In all evaluated scenarios, there is always exactly one light

sensor that consistently produces a value higher than 170 lumen, which is the threshold for concussion patients. Hence, this `Observation` will always be an `AboveThresholdAction`, given the considered context. Regardless of the exact amount of sensors, no observations by any other sensor are filtered by any query.

For all evaluated scenarios, the *total execution time* metric has been calculated for each window. This time refers to the time starting from the Esper window trigger until the moment where the found `AboveThresholdAction` individuals are outputted by the corresponding query. For the DIVIDE set-up 1, the maximum time over the two filtering queries is taken. For set-up 2 and 3, this total execution time ends when RDFox yields the results of query 2.

All evaluations for each set-up are run on a processing device suited for the IoT: an Intel NUC, model D54250WYKH. It has a 1300 MHz dual-core Intel Core i5-4250U CPU (turbo frequency 2600 MHz) and 8 GB DDR3-1600 RAM.

4.3 Real-Time DIVIDE Performance on a Raspberry Pi

To evaluate how well the filtering approach of the DIVIDE system performs on a low-end device, the DIVIDE set-up 1 of Section 4.2 is also evaluated on a Raspberry Pi 3, Model B. This Raspberry Pi model has a Quad Core 1.2GHz Broadcom BCM2837 64bit CPU, 1GB RAM and MicroSD storage. Besides the physical machine, the same evaluation conditions as in Section 4.2 apply.

5 Evaluation Results

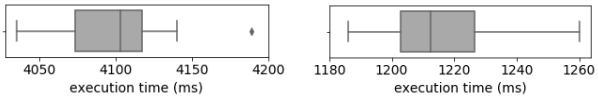
This section presents the results for the evaluation set-ups described in Section 4. All results are averaged over 30 runs, excluding 3 warm-up and 2 cool-down runs.

5.1 DIVIDE Performance Evaluation

Figure 3 shows the distribution of the execution times of the ontology preprocessing and query derivation processes of the DIVIDE system, over the evaluation runs. On average, the ontology preprocessing takes 4.100 seconds, and the query derivation process 1.215 seconds. Both processes have a quite constant duration.

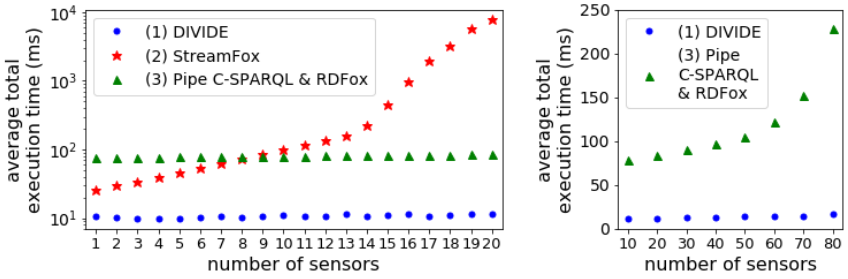
5.2 Comparison of DIVIDE with Real-Time Reasoning Approaches

Figure 4 shows the comparison of the total execution time for different amounts of sensors, averaged over multiple runs, and over the executions within the engine's runtime during each run. Looking at 1 to 20 sensors (Figure 4a), the average total execution times for the DIVIDE set-up and the C-SPARQL–RDFox pipe set-up remain more or less constant, respectively in the range 9–12 ms and 75–85 ms. For the StreamFox set-up, the average total execution time increases exponentially for an increasing amount of sensors: 26 ms for 1 sensor, 100 ms for 10 sensors, and 7861 ms for 20 sensors. Looking at the results for up to 80 sensors (Figure 4b), the DIVIDE set-up only slightly increases to 17 ms for 80



(a) Ontology preprocessing (b) Query derivation

Figure 3. Performance results of the DIVIDE system



(a) (b)

Figure 4. Comparison of the total execution time for different numbers of sensors, averaged over the engine's runtime and over multiple runs (on Intel NUC)

sensors, while the C-SPARQL–RDFox pipe set-up goes up to 228 ms. No results were measured for StreamFox for more than 20 sensors, due to the infeasibility of properly measuring the exponentially increasing total execution times.

To further inspect the set-up behaviors over the engines' runtime, Figure 5 shows a timeline comparing the total execution time, averaged per window number, for a context with 20 sensors. For StreamFox, this shows that for each runtime, the total execution times also exponentially increase over the windows. This shows the accumulation of the event processing: for window 1, this time is only 486 ms; for window 10, it is 3430 ms; and for window 20, it is 15844 ms. For the other two set-ups, the total execution times are somewhat higher at the start, caused by starting up C-SPARQL. Note that the last windows are omitted from the results, as StreamFox did not finish the processing of these windows.

5.3 Real-Time DIVIDE Performance on a Raspberry Pi

Figure 3 shows the results of the evaluation of the DIVIDE set-up on the Raspberry Pi. It shows a distribution of the total execution times for scenarios with different amounts of sensors. Going from 1 to 80 sensors, there is a small increase in average total execution time from 38 ms to 81 ms. In general, there are some outliers with a higher total execution time, especially for higher amounts of sensors. Comparing the results with the results in Figure 4, where the device was the only difference in evaluation conditions, the average total execution times on the Raspberry Pi are always a factor 3 to 4 times those on the Intel NUC.

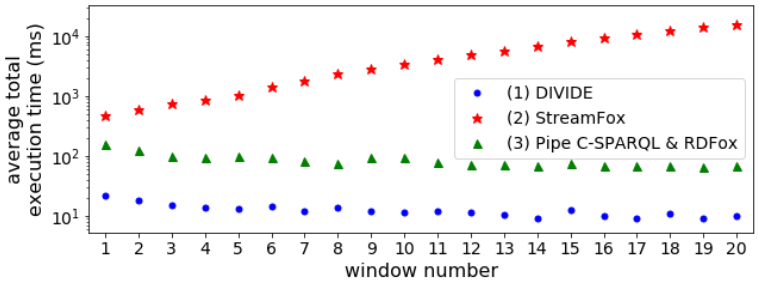


Figure 5. Timeline with the comparison of the total execution time, averaged per window number, for a set-up with 20 sensors (on Intel NUC)

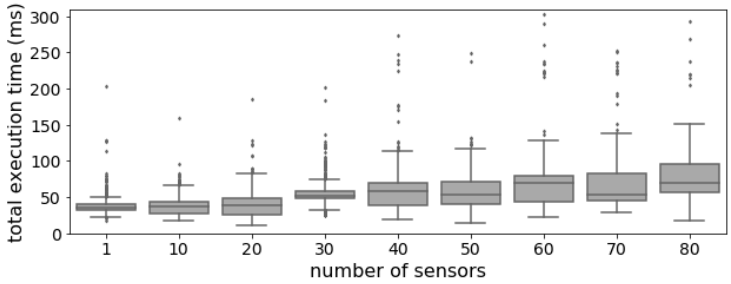


Figure 6. Total execution time distribution (over engine's runtime & multiple runs) for DIVIDE set-up, with different numbers of sensors (on Raspberry Pi 3, Model B)

6 Discussion

An important advantage of the DIVIDE system, is the removal of the need to perform real-time reasoning. The evaluation results in Sections 5.2 & 5.3 prove that this advantage has a significantly positive impact on the total execution time to derive the conclusions relevant for the use case. Applied on the running healthcare example, the usage of DIVIDE in combination with the well-known RSP engine C-SPARQL significantly outperforms the evaluated alternatives.

Before discussing these results in more detail, note that they show the performance of the compared set-ups for multiple amounts of sensors. In all set-ups, each individual measure is calculated on the window content of a 1 second tumbling window. As each sensor had an event rate of 1 observation per second, the amount of sensors always equaled the amount of observations in each window. Hence, apart from the amount of sensors in the context, the results generalize to other situations with more or less sensors, but a lower or higher event rate, leading to the same amount of incoming observations per second. Therefore, they also give an idea of the general throughput of each set-up.

Considering the StreamFox set-up, the results show how its performance degrades with an increasing amount of sensors. Inspecting the timeline for 20

sensors in Figure 5, it is clear that the total execution time exponentially increases as the scenario goes on. This happens because the RDFox reasoning time increases per evaluated window. When the amount of observations in the window content increases, the initial incremental reasoning step to add the event takes longer. In incremental reasoning, the most expensive operation is however the removal of facts [15]. Hence, the duration of the incremental deletion step increases the most. As this step happens *after* query 2 has outputted the actions, it does not influence the total execution time for that event. However, for a larger amount of sensors, the total processing time of one event, including the removal, surpasses the 1 second threshold. Hence, when the next window triggers and the event is added to the processing queue, its processing cannot immediately start. In this way, the removal of the previous event impacts the total execution time of this new event. As the scenario goes on, this impact accumulates, and the waiting time for windowed events in the queue gets longer and longer. Regarding this, two things should be noted. First, as a consequence, the average total execution times over the StreamFox runtime, as reported in Figure 4a, are highly dependent on the amount of consecutive non-empty windows, i.e., the scenario duration. This was 25 seconds for this evaluation, but increasing or decreasing this will also increase or decrease these average values. Second, the large increase in reasoning times, especially for the event removal, is caused by the large amount of rules extracted from the ACCIO ontology by RDFox. However, this is realistic for complex IoT domains such as healthcare, where large bodies of complex domain knowledge are required to correctly analyze the sensor streams.

Inspecting the results of the set-up piping C-SPARQL with RDFox, the main conclusion is that this set-up does not scale as badly with an increasing amount of sensors. An increase in total execution time is noticeable, but slightly for up to 60 sensors. Nevertheless, it consistently takes at least 6 times longer than with the DIVIDE set-up. Important to note here is that the RDFox execution time does not depend on the amount of sensor observations, as only one observation is filtered by C-SPARQL in all evaluation cases. The main difference is in the C-SPARQL query execution times, which take longer because they are executed on a model that also contains all triples in the context and the ontology.

In contrast to the two alternatives, the DIVIDE set-up on the Intel NUC does almost not suffer from an increasing amount of sensors. This is because more sensors in the context do not influence the queries derived by DIVIDE, given no other context changes. As the queries are only executed on the streams, do not take into account ontology or context data, and not require reasoning, the impact on the actual query execution time is also minimized. In addition, the results in Figure 6 show that C-SPARQL can also run the DIVIDE queries efficiently on a low-end device like a Raspberry Pi. The total execution times are larger than on the Intel NUC, but for up to 80 sensors, most still remain below 150 ms. This is an advantage when deploying the system, for example in all rooms of a hospital, as no large scale investment in expensive high-end hardware is required.

The reasoning required by DIVIDE is not performed on the observations stream, but only on the ontology and context data. The query derivation pro-

cess is triggered by context changes, which typically have a frequency that is several factors smaller than the observation data frequency. Hence, the amount of reasoning steps is significantly reduced. The evaluation results in Section 5.1 show that such a query derivation takes approximately 1.2 seconds for a realistic context of a hospital room with 1 patient and 10 sensors, on a normal pc. This time is of course highly dependent on the input data, but optimizations are always possible. By doing the ontology preprocessing, the query derivation process duration can also be largely reduced; for this paper's example, this reduction was approximately 77 %. Note as well that when using DIVIDE in a real set-up, this reasoning will be performed on a central server with many resources, introducing possibilities for parallelization and process acceleration. By using EYE and N_3, the flexibility also exists to extend the rule set beyond OWL 2 RL.

Importantly, the usage of DIVIDE also has other benefits that do not relate to execution times. Being able to locally derive certain conclusions, e.g., actions the system should take, gives an IoT set-up the local autonomy to react on certain events in a responsive way. Moreover, in contrast to other set-ups, no context information should be known and kept up to date locally. This removes synchronization issues, but also avoids potential privacy and security concerns.

The DIVIDE system produces RSP-QL queries for a given context, which can be translated to the correct RSP engine syntax to continuously run on it. By adding a module to DIVIDE that automatically calls the query derivation process upon context changes and performs this translation, the whole query configuration of RSP engines could be fully automated and adaptive. Hence, with DIVIDE, this will no longer be a manual, labor-intensive task.

7 Conclusion

In this paper, the DIVIDE system is presented, which can serve as a component of a semantic IoT platform. The main goal of DIVIDE is to automatically derive queries for an IoT platform's stream processing components, which filter the data streams, in an adaptive and context-aware way. Whenever the application context changes, DIVIDE can derive the queries that filter the observations of interest for the use case, based on this changed context. By performing the reasoning upon context changes, relevant sensors & their observations can be filtered without the need to perform reasoning while evaluating the continuous queries. The evaluation results show that this approach allows to greatly reduce the real-time processing times, and scales much better when the amount of events or sensors in the data stream increases. In this way, the real-time filtering can be performed efficiently on low-end devices. When used in a IoT platform, DIVIDE can divide the amount of queries that need to be deployed at any given time and thus conquer the scalability & performance issues of reasoning on large data streams.

Future work consists of further generalizing the sensor query rule description, to reduce the configuration required for using DIVIDE. One possibility is to integrate dynamic observation patterns into the queries, that could be part of the stream metadata. In addition, it should be researched how the query instan-

tiation could be extended to other query parameters, such as the window parameters, possibly by integrating context metadata such as device information.

Acknowledgements. F. Ongenae is funded by a UGent BOF postdoc grant. Part of this research was funded by the FWO SBO grant 150038 (DiSSeCt).

References

1. Aggarwal, C.C., et al.: The Internet of Things: A survey from the data-centric perspective. In: Managing and mining sensor data, pp. 383–428. Springer (2013)
2. Arndt, D., et al.: Improving OWL RL reasoning in N3 by using specialized rules. In: OWLED 2015. pp. 93–104. Springer (2015)
3. Arndt, D., et al.: SENSdesc: Connect Sensor queries and Context. In: BIOSTEC 2018. pp. 1–8 (2018)
4. Barbieri, D.F., et al.: C-SPARQL: a continuous query language for RDF data streams. International Journal of Semantic Computing **4**(1), 3–25 (2010)
5. Berners-Lee, T., et al.: N3logic: A logical framework for the world wide web. Theory and Practice of Logic Programming **8**(3), 249–269 (2008)
6. Bonte, P., et al.: Streaming MASSIF: Cascading Reasoning for Efficient Processing of IoT Data Streams. Sensors **18**(11), 3832 (2018)
7. Calbimonte, J.P., et al.: Query rewriting in RDF stream processing. In: ESWC 2016. pp. 486–502. Springer (2016)
8. Compton, M., et al.: The SSN ontology of the W3C Semantic Sensor Network Incubator Group. Web Semantics **17**, 25–32 (2012)
9. Cyganiak, R., et al.: RDF 1.1 concepts and abstract syntax. W3C Recomm. (2014)
10. Daniele, L., et al.: Created in close interaction with the industry: the smart appliances reference (SAREF) ontology. In: FOMI 2015. pp. 100–112. Springer (2015)
11. Dell'Aglio, D., et al.: RSP-QL semantics: A unifying query model to explain heterogeneity of RDF stream processing systems. IJSWIS **10**(4), 17–44 (2014)
12. Dell'Aglio, D., et al.: Stream reasoning: A survey and outlook. Data Science (Preprint), 1–25 (2017)
13. Mileo, A., et al.: StreamRule: a nonmonotonic stream reasoning system for the semantic web. In: RR 2013. pp. 247–252. Springer (2013)
14. Motik, B., et al.: OWL 2 web ontology language profiles. W3C Recomm. (2009)
15. Nenov, Y., et al.: RDFox: A highly-scalable RDF store. In: ISWC 2015. pp. 3–20. Springer (2015)
16. Ongenae, F., et al.: An ontology co-design method for the co-creation of a continuous care ontology. Applied Ontology **9**(1), 27–64 (2014)
17. Puiu, D., et al.: CityPulse: Large scale data analytics framework for smart cities. IEEE Access **4**, 1086–1108 (2016)
18. Su, X., et al.: Adding semantics to Internet of Things. Concurrency and Computation: Practice and Experience **27**(8), 1844–1860 (2015)
19. Su, X., et al.: Stream reasoning for the Internet of Things: Challenges and gap analysis. In: WIMS 2016. ACM (2016)
20. Urbani, J., et al.: Column-oriented datalog materialization for large knowledge graphs. In: AAAI 2016 (2016)
21. Verborgh, R., et al.: Drawing conclusions from linked data on the web: The EYE reasoner. IEEE Software **32**(3), 23–27 (2015)
22. Ye, J., et al.: Semantic web technologies in pervasive computing: A survey and research roadmap. Pervasive and Mobile Computing **23**, 1–25 (2015)