Bridging the Gap Between Expressivity and Efficiency in Stream Reasoning: a Structural Caching Approach for IoT Streams

Pieter Bonte · Filip De Turck · Femke Ongenae

Abstract In today's data landscape, data streams are well represented. This is mainly due to the rise of data-intensive domains such as the Internet of Things (IoT), Smart Industries, Pervasive Health, and Social Media. To extract meaningful insights from these streams, they should be processed in real-time, while solving an integration problem as these streams need to be combined with more static data and their domain knowledge. Ontologies are ideal for modeling this domain knowledge and facilitate the integration of heterogeneous data within data-intensive domains such as the IoT. Expressive reasoning techniques, such as OWL2 DL reasoning, are needed to completely interpret the domain knowledge and for the extraction of meaningful decisions.

Expressive reasoning techniques have mainly focused on static data environments, as it tends to become slow with growing datasets. There is thus a mismatch between expressive reasoning and the real-time requirements of data-intensive domains. In this paper, we take a first step towards bridging the gap between expressivity and efficiency while reasoning over high-velocity IoT data streams for the task of event enrichment. We present a structural caching technique that eliminates reoccurring reasoning steps by exploiting the characteristics of most IoT streams, i.e. streams typically produce events that are similar in structure and size. Our caching technique speeds up reasoning time up to thousands of times for fully-fledged OWL2 DL reasoners and even tenths and hundreds of times for less expressive OWL2 RL and OWL2 EL reasoners.

Keywords Stream Reasoning, Event-Based, Caching, IoT

1 Introduction

In today's data landscape, data streams are omnipresent [14]. Data-intensive domains, such as the Internet of Things (IoT), Smart Industries, Smart Manufacturing, Pervasive Health, financial sector, and Social Media, are still gaining popular-

P. Bonte
IDLab, Ghent University - imec, Technologiepark-Zwijnaarde 126, B-9052, Gent, Belgium
E-mail: pieter.bonte@ugent.be

ity. This results in huge amounts of produced heterogeneous data streams. These streams should be processed in real-time, as extracted insights change continuously due to the velocity of the stream. Stream processing has proven successful in continuous processing of data streams [28]. However, in order to extract meaningful insights from these data streams, a data integration problem needs to be solved. This is especially true in the IoT, where a variety of sensor can be used, each using their own protocols, syntax and data model [1]. Various data streams need to be combined or integrated with more static data sources. For example, IoT sensors might transmit their data in JSON format, while their meta-data is stored in a relational database or a key-value store. The integration with the meta-data is often necessary to extract the exact location of the sensor, the observed properties or its position/function within a larger system.

Semantic web technologies, such as ontologies, are the preferred model for the integration of heterogeneous data [1,9]. Ontologies can serve as a unifying model, allowing various data sources to be aligned and integrated. An ontology formally describes concepts, properties, and their relations, within a certain domain [22]. By defining the relations between various concepts, a model can also incorporate the knowledge about a certain domain. For example, defining the ranges the produced values of an IoT sensor should be contained in, when its correctly functioning. By incorporating this domain knowledge, malfunctioning sensors can automatically be detected. Reasoning techniques are needed to interpret the modeled domain knowledge and thus allow to extract meaningful decisions [10]. The more elaborate the modeling of the domain knowledge, the more expressive the reasoning needs to be to fully interpret the domain. To extract meaningful decision from data streams, the events in the streams should be *enriched*, i.e. they need to be integrated with both static data sources and their domain knowledge [11,15].

In this paper, we specifically focus on OWL2 as ontological language to define the domain knowledge and expressive reasoning within the OWL2 DL profile, as it is a web standard. This expressivity is necessary as previous research [10] has shown that almost all IoT ontologies within the Linked Open Vocabularies repository[1] require OWL2 DL reasoning to be fully interpreted.

The Stream Reasoning (SR) community states that event enrichment can be achieved through reasoning [16], however, current reasoners were not designed for event enrichment [42]. Expressive OWL2 reasoners have mostly focused on rather static data [42]. There is still a mismatch between expressive reasoning and the real-time requirements of data stream intensive domains [16]. Scalable expressive stream reasoning is still an open issue in the SR community [16], while expressive ontologies are in fact widely used within the field of IoT [10]. Note that expressive OWL2 DL reasoners can have up to NEXPTIME complexity [27], resulting in slow reasoning times. This is in contradiction with the fact that streams should be processed in real-time. In this paper, we set the first steps for bridging the gap between expressiveness and efficiency in IoT Stream Reasoning applications and focus specifically on *Event enrichment*.

Event enrichment is the task of combining (joining) the events with the static knowledge base, and reasoning over these events in order to infer and add all the available rich semantic information [43]. Once these events have been enriched, they can be used by querying tasks for further processing and analysis. However,

---

[1] `https://lov.linkeddata.es/dataset/lov/`

the challenge lies in enriching these events in a timely manner, as the reasoning process is a time-consuming task and existing reasoners were not designed to enrich streams. We investigate how we can speed up the enrichment process by exploiting the characteristics of the IoT streams, i.e. events in IoT streams are typically similar in structure and limited in size [11]. These characteristics have the benefit that it loosens particular reasoning constraints due to the reoccurring nature of the events and their limited size. Current expressive reasoners will perform the same reasoning steps over and over again over these types of reoccurring events, i.e. the events are different observations, however, their structure is the same and so are the reasoning results.

For example, in an Industry 4.0 setting, a sensor will continuously report the state of one specific machine [19]. In a Smart City, sensors will continuously report the state of traffic at a certain location [37]. In a Smart Air Quality case, sensors will continuously report the air quality in a specific room [36]. In all three cases, the structure of the produced observation show limited variation, except for the sensor values themselves. The integration with background data and domain knowledge allows to meaningfully interpret and process these observations.

To allow *event enrichment* through the use of expressive reasoning, we set the following *objectives*:

1. *Expressive reasoning:* In order to correctly interpret the domain, expressive reasoning is required to correctly analyze and capture the domain knowledge.
2. *Eliminate reoccurring reasoning steps*: Previous reasoning steps over similar events, i.e. in shape and size, should be exploited to speed up reasoning performance.
3. *Scalable*: In order to enable high throughput event enrichment for a variety of stream rates, the solution should be able to scale to a large number of events and high velocity data streams.
4. *Integrate static knowledge:* In order to perform event enrichment, it is mandatory to link events to additional data, e.g., the type of measurement linked to the sensor and the location of the sensor, since the sensory data typically only describe the sensor readings.

In this paper, we investigate a special form of caching that exploits the reoccurring nature of IoT events and limit the number of reasoning steps that need to be performed on similar events. The cache is able to differentiate between streaming and static data in order to correctly enrich events in IoT streams. The cache itself is scalable, and imposes limited overhead for events that are limited in size.

We show that this special form of caching allows to speed up event enrichment up to thousands of times for expressive OWL2 DL ontologies with large static knowledge bases. OWL2 contains three lesser expressive profiles, i.e. EL, QL and RL. We show that our caching technique is even up to 10 times faster for less expressive ontologies that fall within the OWL2 RL profile and up to hundreds of times for ontologies within the OWL2 EL profile.

This paper is structured as follows: Section 2 introduces a running example. Section 3 provides some background on expressive reasoning and streams to understand the remainder of the paper, while Section 4 details the domain knowledge for the running example. In Section 5 we introduce the IoT architecture used for processing IoT streams and detail how the *event enrichment* process fits within the complete architecture. Section 6 details the related work. In Section 7, we provide all the details of our caching technique and in Section 8, we detail the evaluation.
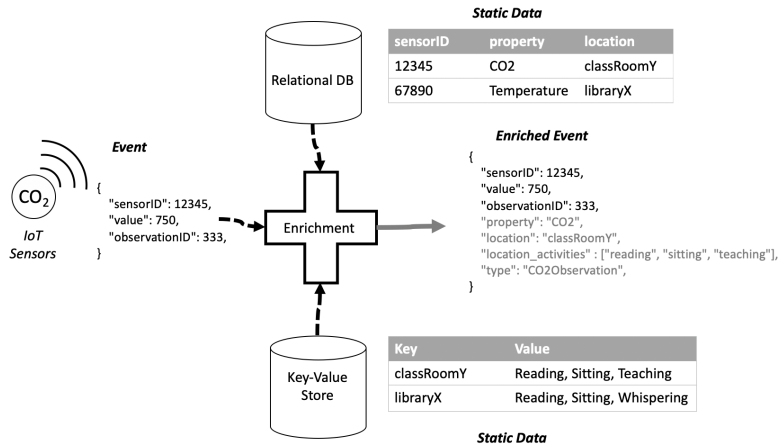
Fig. 1 Example of event enrichment: enriching of CO2 sensor observation with sensor meta-data from a relational database to include the location of the sensor and with the data from a key-value store to include the activities of the location.

Section 9 provides insights into the benefits and limitations of the approach and Section 10 summarizes the contributions and sets a vision for future work.

2 Running Example

We will use a Smart Air Quality case as a running example and focus specifically on the fact that indoor CO2 concentrations correlate with the probability of COVID-19 infection. One could alert the people in the room when the observed concentrations of CO2 exceed a certain threshold. However, the threshold is not static but rather depends on the function of the room [36]. For example, rooms, where people are standing, speaking loudly, or exercising, have lower thresholds than where people are resting, due to the larger amount of exhaled particles. This requires the CO2 observations to be integrated with static background data (the information of the room) and domain knowledge (the COVID-19 infection details). Figure 1 gives an example of an enrichment of a CO2 sensor observation, produced in one of the monitored rooms. Note that the event itself only details that it was made by a certain sensor, not the full details of the sensor or its surroundings. In this case, only the observed value and the ID of the sensor that made the observation. By combining the observation with the static data (containing the sensor details), we can determine that the observation is a high CO2 observation in a classroom, where activities such as *Reading, Sitting and Teaching* can take place. Note that IoT data is often stored using different technologies and in different formats or data models. The sensor produces data in *JSON* format, while the sensor meta-data is stored in a relational database and the functionality of the locations is stored in a key-value store. There is clearly an integration problem, and the domain knowledge itself still needs to be integrated as well as we still need a means to infer that the event is alarming in terms of COVID-19 infections. This problem is typical in the IoT, where data is often very heterogeneous [6]. Therefore, we will rely on ontologies to facilitate the integration of all this heterogeneous data.
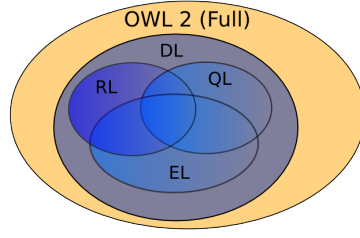
Fig. 2 Relations between the different OWL2 profiles (EL, QL, RL, and DL). (Source: W3C).

3 Background

In this section, we will provide more details on ontologies and their relation with the Resource Description Framework (RDF). Additionally, we give some definitions on the topic of streaming data and explain how ontologies can model IoT streams.

3.1 Ontologies and Description Logic

OWL2 is an ontological language that defines three profiles, i.e. EL, QL, and RL, each optimized for certain reasoning applications. OWL2 DL comprises the union of the three subprofiles, as visualized in Figure 2. This paper focuses on improving the streaming performance of OWL2 DL reasoning. OWL2 DL is the most expressive profile and built upon Description Logic (DL) [3]. As OWL2 DL is the most expressive, the proposed optimizations for OWL2 DL reasoning translated to all the other subprofiles. We introduce the syntax of a simplified DL, explaining the basic notions to understand the remainder of the paper. We refer the reader to Horrocks et. al. [26] for a more thorough description.

DL defines *concepts* C_i, to represent the classes of individuals a_i and *roles* R_i to represent binary relations between individuals or an individual and data values.

Concepts C_i are constructed from two special primitive concepts $\bot$ (bottom) and $\top$ (top) or concept names and roles using the following grammar:

$$C ::= C_i | \top | \bot | \neg C | C_1 \sqcap C_2 | C_1 \sqcup C_2 | \exists R_1.C_1 | \forall R_1.C_1$$

A Terminological Box (TBox) $\mathcal{T}$, is a finite set of concept (C) and role (R) inclusion axioms of the form: $C_1 \sqsubseteq C_2$ and $R_1 \sqsubseteq R_2$. Concept equations ($C_1 \equiv C_2$) denote that both C_1 and C_2 include each other. An Assertion Box (ABox) $\mathcal{A}$ is a finite set of concept and role assertions of the form: $C(a)$ and $R(a_i, a_j)$. While $ind(\mathcal{A})$ denotes the set of individuals occurring in $\mathcal{A}$. A *Knowledge base* $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ combines $\mathcal{T}$ and $\mathcal{A}$. $\mathcal{I}$ is an *interpretation* for $\mathcal{K}$, we say that $\mathcal{I}$ is a model of $\mathcal{K}$, if it satisfies all concept and role inclusions of $\mathcal{T}$ and all concept and role assertions of $\mathcal{A}$. This can be written as $\mathcal{I} \models \mathcal{K}$.

Definition 1 Reasoning is the process of interpreting the domain knowledge and inferring implicit facts. Different reasoning tasks exist, i.e.

- *Consistency checking*: the task of checking if there exist a model that satisfies all the definitions in the knowledge base.

- *Classification*: the task of computing the complete view on the TBox.
- *Realization*: the task of inferring implicit concept and role assertions for an individual defined in the ABox.

In the remainder of the paper, we will focus on the reasoning task of *Realization* and more specifically on *Materialization*, which computes all possible implicit concept and role assertions for all the individuals in the ABox.

Definition 2 We call $\mathcal{M} = \mathcal{K}^\infty$ the **materialization** of $\mathcal{K}$, i.e. all inferred axioms w.r.t. explicit individuals in $\mathcal{K}$ are computed and explicitly stored. This means that, based on the knowledge defined in $\mathcal{T}$, additional axioms regarding $\mathcal{A}$ can be extracted through the use of a reasoner.

Example 1 The TBox defines the concepts within a certain domain. In our use case, we could define various locations using the following concept inclusions, which defines that an *Office*, *Library* and *ClassRoom* are subclasses of the concept *Location*:

$$Office \sqsubseteq Location$$
$$Library \sqsubseteq Location$$
$$ClassRoom \sqsubseteq Location$$
$$...$$

The TBox could also define that a *CO2Sensor* is certain type of *Sensor*:

$$CO2Sensor \sqsubseteq Sensor$$

Furthermore, the role *hasLocation* links *Sensor*s to *Location*s. The ABox can define an instance of a *CO2Sensor* called *sensor1* as $CO2Sensor(sensor1)$ and a specific *ClassRoom room1* as $ClassRoom(room1)$. We can link them together using the role assertion $hasLocation(sensor1, room1)$. Through the use of reasoning and the definitions in the TBox, the materialization process will infer that *sensor1* is of the type *Sensor* and that *room1* is of the type *Location*: $Sensor(sensor1)$, $Location(room1)$.

3.2 Relation to RDF

RDF [35] is a standard to model interchangeable data on the web and allows the serialization of ontologies. It represents data as triples, consisting of a *subject*, *predicate* and *object*. A set of triples forms a *graph*. We give the relation between DL *Concepts* and *Roles* and RDF triples and how they can be visualized as graphs.

As depicted in Figure 3, *Concept* and *Roles* in DL can be converted to RDF triples. Concept assertions of the form *C(a)*, with *C* a concept and *a* an individual can be represented as the RDF triple *:a rdf:type :C*. Role assertions of the form *R(a,b)* can be represented as the RDF triple *:a :R :b*. Subsumption of concepts of the form $C_1 \sqsubseteq C_2$ can be represented as the RDF triple *:C1 rdfs:subClassOf :C2*. Note that in RDF everything is represented as an URI. Prefixes can be used to minimize the URI definitions, for example the prefix *rdf:* is short for `http://www.w3.org/1999/02/22-rdf-syntax-ns#`, thus rdf:type is short for `http://www.w3.org/1999/02/22-rdf-syntax-ns#type`. *:a, :b, :R* and *:C* use the default prefix.

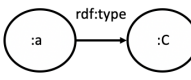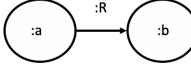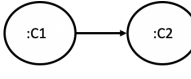Each of the triples can be visualized as a graph. Combining multiple triples results in a larger graph.

DL	RDF Triples	RDF Graph
C(a)	:a rdf:type :C	
R(a,b)	:a :R :b	
C1 ⊑ C2	:C1 rdfs:subClassOf :C2	

Fig. 3 Relations between the different DL concept and roles, RDF Triples and RDF Graphs

3.3 Events and Streams

As the topic of this paper is about enrichment of IoT streams, we provide the definitions of *Streams*, *Events* and the *Enrichment* procedure.

Definition 3 A data stream $\mathcal{S}$ is an infinite sequence of pairs $\langle d_i, t_i \rangle$ where, d_i is a data item, and t_i is a time timestamp. An **Ontology Stream** is a stream where the data item d_i is an ABox and t_i a timestamp.

Each data item in a stream can also be considered an *event*:

Definition 4 We define an **event** E_i as an element of an Ontology stream describing a self-contained observation or state in the form of a limited size ABox. One needs to combine the event with a static Abox $\mathcal{A}$ and domain knowledge described in the TBox $\mathcal{T}$ in order to interpret the event.

Thus to fully interpret these *events*, some kind of *enrichment* is necessary:

Definition 5 Event Enrichment is the materialization of events E_i, while taking the Abox data $\mathcal{A}$ that describes the static knowledge, e.g. sensor metadata, and the TBox $\mathcal{T}$ that describes the domain knowledge into account.

More formally, $Enrichment(E_i, \mathcal{T}, \mathcal{A}) = E_i \cup \{C(e_j) | (\mathcal{T}, \mathcal{A} \cup E_i) \models C(e_j), \forall e_j \in E_i\}$, with e_i the individuals defined in the event E_i.

Static enrichment, which is the materialization of the static data based on the inferred influence of the events, is out of scope for this paper.

3.4 Semantic Sensor Network Ontology

The Semantic Sensor Network (SSN) [13][2] is an ontology for the description of sensors and their observations. SSN is a modular ontology, consisting of a lightweight

---
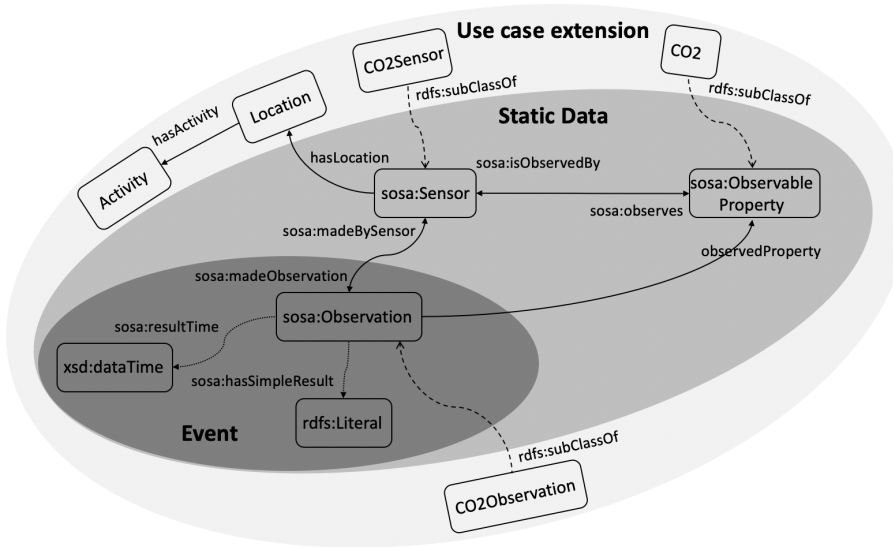
[2] https://www.w3.org/TR/vocab-ssn/

Fig. 4 Overview of the SOSA ontology, the event core, concepts of the static data and the use case extensions.

self-contained core ontology called SOSA (Sensor, Observation, Sample, and Actuator) for the modeling of its elementary concepts and roles. SSN further extends SOSA with additional concepts but also more elaborate domain knowledge.

Figure 4 visualizes the SOSA ontology with its core concepts. At the center is the *Observation* concept which models the observed values and the time of the observation as literal (data) values, through the properties *hasSimpleResult* and *resultTime*. This defines the conceptualization for the *Event*s themselves. The *Observation*s are linked to the *Sensor* that made the observation, and the property that was observed, e.g. CO2. This is the more *Static Data*. Figure 4 shows the distinction between the conceptualization for events and more static data. Furthermore, it shows that for the running example, the SOSA concepts are being extended with *CO2Sensor*s and *CO2Observation*s. Moreover, *Sensor*s are positioned at certain *Location*s, which each can have an number of *Activities*.

3.5 Conversion to the ontology model

Special techniques exist to convert raw data, e.g. JSON, CSV, data from a database, etc. to an ontology model. RDF Mapping Language (RML) [17] allows to convert the raw data to the ontology model through a mapping definition. These mapping definitions specify how parts of the raw data need to be extracted and converted to the ontology model. Figure 5 gives a visual overview of the mapping process for the sensor data from the running example. The values are extracted from the raw sensor data, in this case in JSON format, and converted to a graph according to the ontology model. The figure also gives an example of the mapping definition itself. Note that we used YARRRML [24] for the definition of the mapping, as it is a more concise and human-readable RML format.
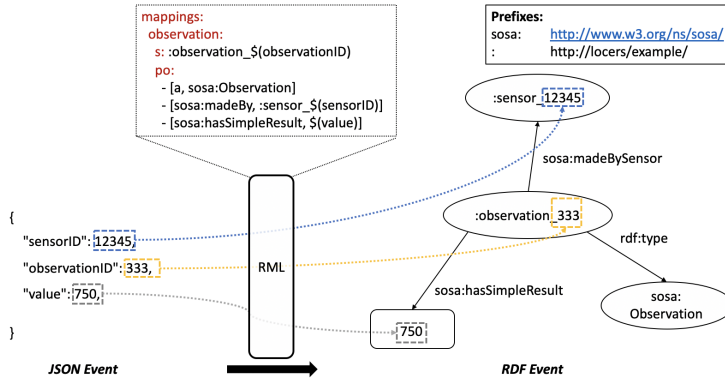
Fig. 5 Converting raw sensor data, e.g. JSON, to the ontology model using RML.

4 Defining the domain knowledge for the running example

We will now introduce the domain knowledge for the running example, defined through the *Concept* and *Role* definitions in DL and show some visual examples using depictions of RDF graphs.

Example 2 (Concept hierarchies) Concept inclusions allow to define hierarchies of concepts. In our running example, each location can have a variety of activities, based on the static defined functionality of the room. We can divide these activities into three categories:

- *Resting Breathing Activities*: that involve activities where people are mostly in rest and are not breathing excessively.
- *Standing Speaking Activities*: including activities where one or more persons are standing and speaking up, resulting in more particles to be exhaled.
- *Heavy Breathing Activities*: activities where people are breathing excessively and most particles are exhaled.

The concept definitions below, define a small part of the activity hierarchy, stating that *Reading* is a subclass of *RestingBreathingActivity*, etc.:

$$Reading \sqsubseteq RestingBreathingActivity$$
$$Whispering \sqsubseteq RestingBreathingActivity$$
$$Sitting \sqsubseteq RestingBreathingActivity$$
$$RestingBreathingActivity \sqsubseteq Activity$$
$$...$$

Figure 6 shows an larger extract of these concept inclusions in a hierarchy, including *StandingSpeakingActivities* and *HeavyBreathingActivities* with their subclasses. Note that this hierarhcy is an extension of the *Activities* that are linked to each *Location* as visualized in Figure 4. Thus each *Location* in our running example, can have a variety of *Activities* from this hierarchy. We can use more complex concept definitions to define the knowledge in a certain domain.
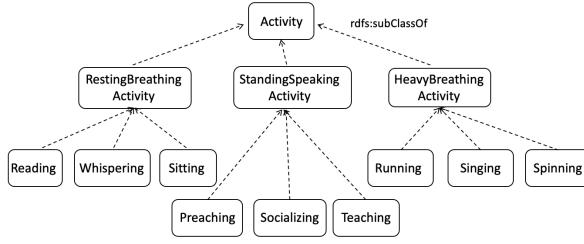
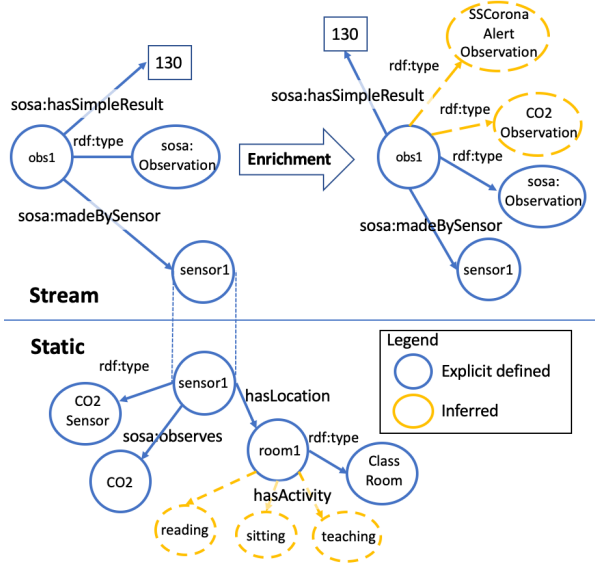**Fig. 6** Extract of the activity concept hierarchy. Each *Location* can have a variety of activities.

Fig. 7 Example of semantic event enrichment of CO2 observation based on the activity of the location it was captured in.

Example 3 (Domain Knowledge) In our running example, the severity of the CO2 values, in terms of potential COVID-19 infections, depend on the function of each location. First, we need to define the function of each room in terms of several activities that it can host. For example, a *ClassRoom* is a *Location* in which *Reading, Sitting and Teaching* can take place as specific *Activities* (as defined in the hierarchy in Example 2) . Note that *Activities* can belong to different parts of the hierarchy. The knowledge regarding the different locations can be modeled as follows:

$$ClassRoom \equiv Location$$
$$\sqcap \exists hasActivity.(Reading \sqcap Sitting \sqcap Teaching)$$
$$MusicClassRoom \equiv Location$$
$$\sqcap \exists hasActivity.(Reading \sqcap Teaching \sqcap Singing)$$

$$SpinningClass \equiv Location$$
$$\sqcap \exists hasActivity.(Teaching \sqcap Spinning)$$
$$Library \equiv Location$$
$$\sqcap \exists hasActivity.(Reading \sqcap Sitting \sqcap Whispering)$$
...

Note that the *equivalence* is used when defining the different rooms, this means that when a location is defined as a *ClassRoom*, the reasoner will infer its activities, even when they are not explicitly defined. Figure 7 shows the enrichment of a sensor observation event which is combined with static data detailing the metadata about the sensors and their locations. Through the event enrichment, the materialization process will infer all implicit facts about the individuals in the event and the static data. In the static data, *room1* is defined as a *ClassRoom*, through the use of reasoning, the activities are automatically inferred, even though they were not specifically defined. This is indicated by the yellow ovals. On the other hand, when the reasoner sees a location with the activities *Reading*, *Sitting* and *Teaching* it will know this is a *ClassRoom*. Note that in Figure 7, we have mapped the raw JSON data and the meta-data from the databases to the ontology model using RML so that the different sources of data can easily be integrated together.

The domain knowledge can also define when actions need to be taken:

Example 4 (Taking actions) In our running example, we need to automatically infer if the persons at a specific location should be alerted for high infection probabilities, based on the activities and functions of the room. This is possible through the following concept definitions, where a *CO2Observation* is defined as an *Observation* that was made by a sensor that observes the property *CO2*. An *HBCoronaAlertObservation* is a *CO2Observation* that was made by a sensor, which is positioned in a *Location* that contain *HeavyBreathing Activies* while the observed sensor value is above 50 ppm. Below we define the domain knowledge for all the activities. Note that the threshold for alerting depends on the activities that are taking place in each location. We used the following abbreviations: (HB=HeavyBreathing, SS=StandingSpeaking, RB=RestingBreathing)

$$CO2Observation \equiv Observation \sqcap \exists madeBy.\exists observes.CO2$$
$$HBCoronaAlertObservation \equiv CO2Observation$$
$$\sqcap \exists madeBySensor.\exists hasLocation.\exists hasActivity.HeavyBreathing$$
$$\sqcap hasSimpleResult > 50$$
$$SSCoronaAlertObservation \equiv CO2Observation$$
$$\sqcap \exists madeBySensor.\exists hasLocation.\exists hasActivity.StandingSpeaking$$
$$\sqcap hasSimpleResult > 100$$
$$RBCoronaAlertObservation \equiv CO2Observation$$
$$\sqcap \exists madeBySensor.\exists hasLocation.\forall hasActivity.RestingBreathing$$
$$\sqcap hasSimpleResult > 1000$$
$$CoronaAlertObservation \equiv HBCoronaAlertObservation$$
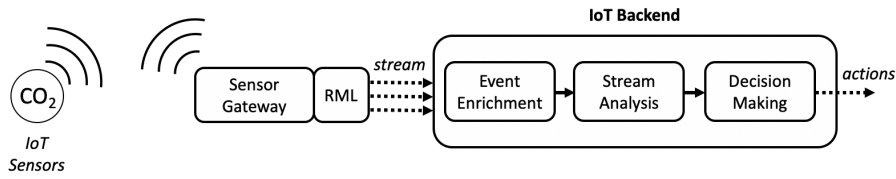$$\sqcup SSCoronaAlertObservation \sqcup RBCoronaAlertObservation$$

Fig. 8 Architecture of IoT pipeline. Sensors transmit their readings to the *Sensor Gateway*, which maps the raw data to RDF through RML, before sending the data to the backend for processing. Before the data can be analyzed, it needs to be enriched. Finally, decisions can be made and actions can be taken.

Note that the *RBCoronaAlertObservation* is more restrictive, as it requires that there are *only* ($\forall$) *RestingBreathing* activities taking place.

Figure 7 shows an example of an event enrichment of a CO2 observation that was captured in a Classroom. Even though many *RestingBreathing* activities take place in the class room[3], since the teacher is performing a *StandingSpeaking* activity (because *Teaching* is a subclass of *StandingSpeaking*), the observation will be enriched as a *SSCoronaAlertObservation*. Note that the event in the stream, i.e. the observation, only states the value and the sensor. The static data describes the observed property, i.e. CO2, and the location of the sensor, i.e. a classroom, without explicitly defining the activities in the room. By combining the observation with the static data, we can infer that the observation is a *CoronaAlertObservation* and more specifically a *SSCoronaAlertObservation*. This allows us to take additional actions and alert the people inside the room that there is a high probability for COVID-19 infection, or take automated actions such as opening doors or windows.

5 IoT Architecture

In this section, we detail the IoT architecture that is used to extract meaningful insights from IoT data and position *Event Enrichment* within this larger system. Figure 8 visualizes the used architecture, consisting of the following components:

- *IoT Sensors*: various IoT sensors, e.g. CO2 sensors, sense their environment and transmit their observations to the *Sensor Gateway*.
- *Sensor Gateway*: serves as an access point for the sensors to transmit their observations to and forwards these observations to the backend for further processing. We utilize the DYAMAND sensor gateway [33][4], as it provides many connectors to easily interact with a variety of sensor protocols and manufacturers. Furthermore, it allows to register plugins to apply additional transformations to the data. We apply such a plugin to convert the raw sensor data to the ontology model using RML [17].
- *RML:* converts the raw data to the ontology model. These RML mappings are typically defined when a new type of sensor is added to the platform. Note that this also means that the structure of the resulting event is fixed. As the shape of the raw data typically does not change, the resulting ontology graph

---

[3] Note that *Reading* and *Sitting* are subclasses of *RestingBreathingActivity* as defined in the activity hierarchy (see Figure 6)

[4] https://dyamand.tech/

will adhere to the same structure, only the values that are injected from the raw sensor JSON will be different.

– *Backend:* is responsible for the further processing of the captured IoT data. The *Event Enrichment* component combines the IoT events with the static knowledge, e.g. the sensor meta-data and the domain knowledge. It then materializes the events, i.e. performs reasoning to infer implicit facts regarding these events. Once the IoT events have been enriched, they are forwarded to the *Stream Analysis* components that investigates each *enriched event* and analyzes if certain events require further actions. The *Decision Making* component then decides which specific actions need to be taken.

The focus of this paper is specifically on the *Event Enrichment* component. The *Stream Analysis* and *Decision making* steps can be achieved through RDF Stream Processing [45], which allows to define declarative queries on top of RDF data streams to enable decision making and analysis.

6 Related Work

In this section, we will visit the related work that has touched on the topic of caching for reasoning or researched optimizations for reasoning over data streams.

OWL2 Reasoners such as HermiT [20] and Pellet [40] contain caching mechanisms but only for a single reasoner run [41]. If data is added, which is the case in a streaming scenario, no caching mechanisms can exploit the reoccurring nature of the data. This is because these reasoners were designed for static scenarios.

Glimm et al. [21] describe an approach that abstracts large ABoxes to smaller abstractions and shows that the reasoning results can be transferred from the abstraction to larger ABoxes. This approach is similar to ours, as we also transfer the results, between events, instead of abstractions and large ABoxes. However, we optimize for streaming data, while the described technique is for static data. The technique is particularly beneficial for extremely large static ABoxes, i.e. millions of triples, a scenario we currently do not consider in the streaming case.

TrOWL [44] is an OWL2 DL reasoner that performs approximations to enable stream reasoning over ontologies. This is supported by incrementally processing the addition and removal of facts while approximating the results. However, it is only an approximation and does not provide any optimization to reduce redundant reasoning steps over the facts in the data stream. Our caching approach is not an approximation and is correct and complete for the task of event enrichment.

Subset reasoning [10] is another approximation technique to materialize events in data streams that need to be combined with large static ABoxes. The technique extracts a subset of the static Abox in order to optimize the materialization process over the events in the stream. However, subset reasoning only gives large performance gains when events need to be combined with large static ABoxes. Furthermore, the technique does not provide any optimization to reduce redundant reasoning steps over the facts in the data stream.

RDFox [34] is the fastest incremental OWL2 RL reasoner currently available. It is built upon the principles of datalog and is optimized for incrementally reasoning upon new facts. However, it is not optimized for a streaming scenario, where events typically describe similar facts. Recently, research on datalog has investigated how frequently occurring workloads can be cached for future use [29]. OWL2

RL reasoners such as RDFox however do not yet contain such optimizations. Note that we are not able to include them, due to the closed-source nature of RDFox. Furthermore, we aim to support up to OWL2 DL and not limit to OWL2 RL.

RDF Stream Processing (RSP) is part of the Stream Reasoning initiative and focuses specifically on the processing of RDF streams [45]. A variety of RSP engines have been proposed by the community, e.g. CQELS [32], C-SPARQL [5] and RSP4J [45], however, these solutions focus more on query answering than on reasoning. As a result, if reasoning is supported, the expressivity is typically limited to RDFS, i.e. computation of hierarchies and transitive relations. IMARS [4] is an incremental materialization maintenance technique for stream reasoning under RDFS entailment. It reduces reasoning steps by investigating which reasoning steps incrementally need to be performed or can be skipped within a time window. IMARS is an incremental approach, but does not perform any form of caching. Furthermore, it operates on RDFS, which is far less expressive than OWL2 DL. StreamQR [12] is the most expressive RSP engine that uses Query Rewriting (QR) to inject the reasoning directly in the query evaluation process. This is done by rewriting a registered query into a much larger query consisting of many UNION definitions in order to inject the ontology TBox inside the query and eliminate the need to reason at query time. However, the expressiveness of the supported ontology is restricted to the $\mathcal{ELHIO}$ logic, which is a subset of the logic used within OWL2 DL. Note that rewriting queries over OWL2 DL ontologies in a sound and complete manner is not possible. When combining RSP with uncertainty management, caching has been introduced to speed up the reoccurring calls to the uncertainty functions [31]. Caching results of uncertainty functions is simpler than caching reasoning results, as the uncertainty functions with the same input parameters always returns the same results and thus can easily be stored.

In the realm of Answer Set Programming (ASP) reasoning, there have been some efforts to eliminate redundant reasoning steps in Stream Reasoners. Bazoobandi et al. [7] propose an optimization technique for LARS, i.e. a stream reasoning framework for ASP, that infers which facts will be impossible to derive from the streams in the future and blocks the reasoner from performing these computations. The technique thus limits the reasoning steps that should be performed on the stream. Compared to our technique, it will still recompute the same reasoning steps over reoccurring events, however, only deriving the facts that are meaningful. Furthermore, we focus on OWL2 instead of ASP.

Dodaro et al. [18] use reinforcement learning to identify which concept derivations are most useful for the stream and introduce an efficient caching methodology. The technique is specific for ASP solvers and is not always complete due to the learning component.

MASSIF [9] is a semantic message broker that allows consumers to subscribe to events using high-level ontological concepts that are evaluated using expressive OWL2 DL reasoning. To obtain decent performance, it provides an incomplete caching technique. Incomplete in the sense that it 1) cannot deal with subscriptions that are subclasses of each other, 2) can only cache the inference for a single individual in the event, and 3) cannot deal with data property or nominal reasoning. In this work, we present a caching technique that produces complete results for the enrichment of the whole event and prove its correctness.

It is clear that there is a large interest in increasing reasoning performance over dynamic data. To the best of our knowledge, we are the first to propose an
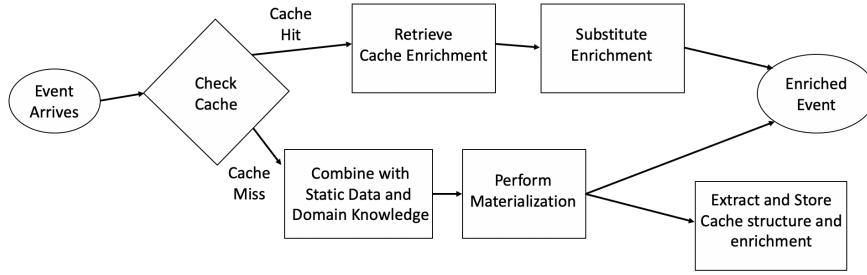
Fig. 9 Flowchart for event enrichment using the cache

attempt that exploits the characteristics of the stream to increase performance in a very natural and low-impact fashion through structural caching. Our proposition is reasoner independent and can be combined with any OWL2 reasoning system.

7 Caching for Event Enrichment

This section details the inner workings of our caching technique. Before going into the details, we explain the high-level idea. Figure 9 shows a visual representation of the flow of the cache. Our caching technique eliminates the need to reason upon reoccurring events, by identifying the structure of events that will lead to the same derivations through reasoning. This means that we move from reasoning to pattern matching. When a new event arrives, we check the cache. When there is no hit, we combine the event with the static data and the domain knowledge and then we materialize the event. We then extract the *Cache Structure* of the event, i.e. the structure that will be used for pattern matching when the next events arrives. Next, we store the inferred types for each individual in the event and store them as *Cached Enrichment*s. When a new event arrives, we check if there is a *Cache Structure* that exactly matches the event and we add the stored *Cached Enrichment*s. This allows to perform the materialization through pattern matching instead of reasoning. Note that it is important that there is an exact match between the new event and the *Cache Structure*, i.e. if the event has relations that are not in the *Cache Structure*, then there is no match. We explain the reason for this in detail in the remainder of this Section.

7.1 Creating Cache Structures

We first explain how events are converted to *Cache Structures* and how *Cache Enrichment* are extracted.

Definition 6 A *Cache Structure* is a special abstraction of an event with the property that if an event matches the *Cache Structure*, it can be materialized in the same way as the original enriched event from which the *Cache Structure* was extracted. The materialization itself is stored in the *Cache Enrichment* which contains all the inferred types that need to be added when an event matches the *Cache Structure*.

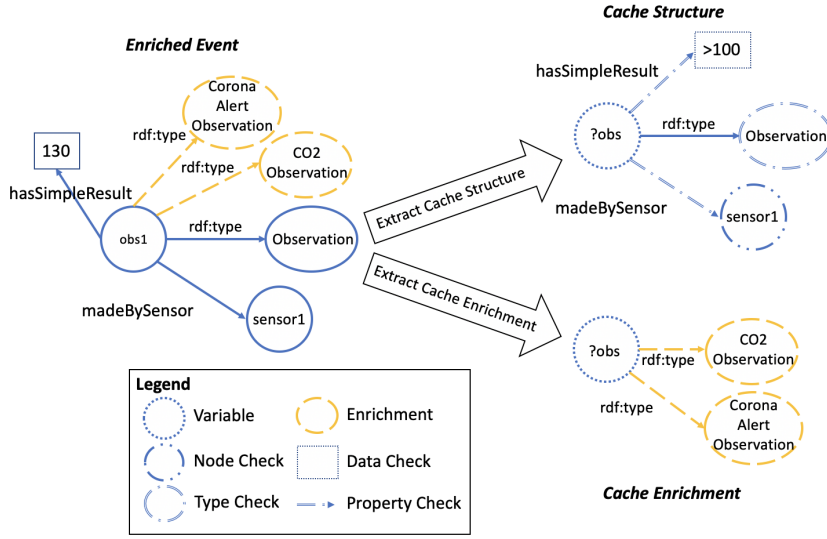The *Cache Structure* can impose a number of constraints:

Fig. 10 Cache structure extraction for event enrichment

- A *Node Check* is a constraint in the *Cache Structure* that requires the same individual to be present in the event as defined in the *Node Check*.
- A *Type Check* is a constraint in the *Cache Structure* that requires the same type assertion to be present in the event as defined in the *Type Check*.
- *Variables* need to be bound to a unique individual in the event. An individual can only be bound to a single *Variable*. *Variables* allow to match multiple events and thus transfer the enrichment from one event to another.
- A *Property Check* is a constraint that requires the same property to be defined over *Node Checks* or *Variables* that have already been successfully matched.
- A *Data Check* requires data values to be above or below a certain threshold.

The evaluation of the *Cache Structure* requires all of its constraints to be satisfied. Figure 10 shows an example of the extraction of a *Cache Structure* from an enriched event. We start from the enriched CO2 observation from Figure 7, consisting of the inferred types *CO2Observation* and *SSCoronaAlertObservation*, which could only be inferred by adding the sensor and location meta-data and the domain knowledge.

When extracting the *Cache Structure* we make a distinction between individuals from the events and individuals that are a reference to the static data. In Figure 7, the observation states that it is made by *sensor1*, while *sensor1* is described in the static background data. Thus *sensor1* is a reference to the static data. All **references to the static data** should be added as *Node Checks* in the *Cache Structure*. This means that in order to match the structure, the observation should have been made by *sensor1*. This is shown in the *Cache Structure* in Figure 10 with the variable dotted circles.

Example 5 (Static data) Suppose an additional CO2 sensor, *sensor2*, that is installed in the library:

$$CO2Sensor(sensor2), hasLocation(sensor2, room2), Library(room2)$$

When the same observation from Figure 7 would be made by *sensor2*, it would not cause an *SSCoronaAlertObservation*, as the library only allows *RestingBreathingActivities* and the threshold for alerts in such a location is much higher than the observed value (as specified in the domain knowledge detailed in Section 4).

Individuals from the event that have no references to the static data, should be substituted to *Variables* instead, allowing the structure to match various events.

Example 6 (Event data) Suppose a similar event:

$$Observation(obs2), madeBySensor(sensor1), hasSimpleResult(130)$$

The materialization of this event will result in the same inferred types as *obs1*.

The **object properties** described in the event should be added to the structure as *Property Check*s. This means that the same number and same type of properties should be present in a new event before the structure can be matched.

Example 7 (Object properties) Suppose that *Observation obs3* is similar to *obs2* but has an additional object property indicating its battery level:

$$hasBatteryLevel(obs3, low)$$

And the following axiom exists that infers low precision observations if their battery levels are low:

$$LowPrecisionObservation \equiv Observation \sqcap \exists hasBatteryLevel.Low$$

obs3 can be inferred as *LowPrecisionObservation*, which was not the case for *obs2*.

Data properties defined in the event can be ignored, if they are not part of a class definition that contains a data property restriction, e.g. such as *CoronaAlertObservations*, which state that the observed data values should be above a certain threshold. These restrictions are added as *Data Checks* to the *Cache Structure* if one of the concepts that contain a data property restriction is inferred in the materialization process. We use the same technique for nominal restrictions.

Example 8 (Data properties) The *SSCoronaAlertObservation* defines a threshold above 100. This restriction is added as a *Data Check* to the structure.

Type assertions defined in the event need to be added as *Type Check*s to the *Cache Structure*, for the same reason as object properties required exact matches.

Algorithm 1 gives a simplified version of the conversion algorithm in pseudocode. It shows the conversion of individuals either as *Variables* or *NodeChecks* and the property and type assertions as either *PropertyChecks* or *TypeChecks*, using the converted individuals. For simplicity reasons, we omitted the conversion of data properties and nominals.

Algorithm 1: *CacheStructure* extraction algorithm

```
// Extract the CacheStructure for event E
indStore = { } // stores indiviuals either
// as Variables or NodeChecks
// stores the cache structure
cacheStructure =new CacheStructure()
// Convert indiviuals to Variables or NodeChecks
for ind in Ind(E):
    if ind not in static data:
        indStore[ind] = Variable(ind)
        cacheStructure.addVariableCheck(Variable(ind))
    else:
        indStore[ind] = NodeCheck(ind)
        cacheStructure.addNodeCheck(NodeCheck(ind))
// Convert Object Properties to NodeChecks
for (subj, prop, obj) in objectProperties(e):
    cacheStructure.addPropCheck(indStore[subj], prop, indStore[obj])
// Convert Type assertions to NodeChecks
for (ind, a, type) in typeAssertions(e):
    cacheStructure.addTypeCheck(indStore[ind], type)
return cacheStructure
```

7.2 Storing the Materialization

All the inferred types for each individual in the original enriched event, from which the *Cache Structure* is extracted, are stored in the *Cache Enrichment* and linked to the extracted *Variables* or *Node Checks* for static reference individuals. When we add a new *Cache Structure* to the cache, we first extract the *Cache Structure* from the event and then combine the event with the static data to materialize the event. For each individual in the event, we store the inferred types retrieved through the materialization and link them to the extracted *Variable* or *Node Check* of the *Cache Structure*. The linking of the inferred types to the *Cache Structure* is called the *Cache Enrichment* and is depicted in Figure 10 at the bottom right. We note that *Cache Enrichment*s and *Cache Structure*s are stored together.

Example 9 (Cache Enrichment) In Figure 10, *obs1* is converted to the variable *?obs* in the extracted *Cache Structure*. Upon materialization, *obs1* gets the types *SSCoronaAlertObservation* and *CO2-Observation*. As *?obs* is extracted from the individual *obs1*, we store the inferred types *SSCoronaAlertObservation* and *CO2Observation* for the variable *?obs* in the *Cache Enrichment*. This allows other events, which match the extracted *Cache Structure*, to be materialized in the same fashion.

7.3 Checking the Cache

When checking the cache, it is important to have an exact match in order to claim a cache hit. This means that all the *Checks* should be fulfilled:

1. All references to the static data should be present.
2. The same type assertions should be defined.
3. The same property relations should be present.
4. Each variable should be bound to a unique individual.

5. Data property restrictions, if any, on literals should be evaluated.
6. Nominal restrictions, if any, should be evaluated.

If all the checks can be fulfilled, there is a cache hit. In this case, the stored extracted materialization for each variable can be assigned to the individuals that are bound to the variables in question.

Example 10 (Cache hit) When *Observation obs2* matches the *Cache Structure*, *obs2* will be bound to the *Variable ?obs*. In the *Cache Enrichment*, *?obs* is assigned to the inferred types *CO2Observation* and *CoronaAlertObservation*. Thus, *obs2* gets assigned the inferred types *CO2Observation* and *CoronaAlertObservation*.

This allows us to move from reasoning to pattern matching, i.e. checking the *Cache Structures*. In the evaluation, we will show that the latter is a lot more efficient.

7.4 Theoretical Foundations

This section details the theoretical foundations of our caching mechanism. It proves that the cache is correct and complete for the task of event enrichment.

As the cache is looking for similarities in structures of the events, we first define a function that translates ABoxes to new ABoxes with the same structure:

Definition 7 The **translation function** ε translates an ABox $\mathcal{A}$ to a new Abox $\mathcal{A}'$ such that all names $n \in N_{\mathcal{A}}$ (with $N_{\mathcal{A}}$ the individual names in $\mathcal{A}$) have a name translation in $\mathcal{A}'$ such that i) there is a one-on-one translation for each n to $\varepsilon(n)$, ii) for all concept assertions $C_i(a) \in \mathcal{A}$ the same concept assertion holds in $\mathcal{A}'$ such that $C_i(\varepsilon(a)) \in \mathcal{A}'$, iii) for all role assertions $R \in \mathcal{A}$ i.e. $R_i(n_j, n_q) \in \mathcal{A}$ the same relation R_i holds in $\mathcal{A}'$ such that $R_i(\varepsilon(n_j), \varepsilon(n_q)) \in \mathcal{A}'$.

This means that when an event E_i is added to the cache, its *Cache Structure* will match for any ABox $\varepsilon(E_i)$. We now define how the inferred types of E_i can be assigned to $\varepsilon(E_i)$ as in the *Cache Enrichment*.

Definition 8 A **translated equivalence function** $\equiv_\varepsilon$ states that if $C(a)$ is entailed in $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ and there exist a translation ε to $\mathcal{A}'$, then $C(\varepsilon(a))$ is entailed in $\mathcal{K}' = (\mathcal{T}, \mathcal{A}')$, i.e. $K \models C(a) \equiv_\varepsilon K' \models C(\varepsilon(a))$.

Proof The translation function ε can be seen as a *homomorphism* h with mappings $\forall a \in ind(\mathcal{A}) : \bigcup \{a \to \varepsilon(a)\}$. Let $\mathcal{A}$ and $\mathcal{A}' = \epsilon(\mathcal{A})$ be ABoxes and $h : ind(\mathcal{A}) \to ind(\mathcal{A}')$ a homomorphism from $\mathcal{A}$ to $\mathcal{A}'$. Then, by the definition of homomorphisms, for every TBox $\mathcal{T}$ and every axiom α, $\mathcal{A} \cup \mathcal{T} \models \alpha$ implies $\mathcal{A}' \cup \mathcal{T} \models h(\alpha)$. Therefore $K \models C(a)$ implies $K' \models C(h(a))$ with $h(a) = \varepsilon(a)$.

To make a distinction between events in the data streams and static background data we define a special translation function:

Definition 9 The **background translation function** ε_B is a translation function ε that fixes the translation for the individuals defined in the static background ABox $\mathcal{B}$. This can be seen as a translation function that maps the individuals defined in $\mathcal{B}$ onto themselves.

This means that for the ABox $\mathcal{A} = \mathcal{A}_{event} \cup \mathcal{B}$ with $\mathcal{B}$ the static background data, the translation function will only translate the individuals defined in the event data $\mathcal{A}_{event}$. As ε_B still can be seen as a homomorphism that maps the individuals from B onto themselves, $\equiv_{\varepsilon_B}$ still holds. We will now show that when a translation function exists between two events, their event enrichment is the same under $\equiv_{\varepsilon_{\mathcal{B}}}$:

Theorem 1 *If a translation function between two events E_i and E_j exists, then their event enrichment is the same under $\equiv_{\varepsilon_{\mathcal{B}}}$*

Proof We need to prove that:
$Enrichment(E_i, \mathcal{T}, \mathcal{A}) \equiv_{\varepsilon_{\mathcal{B}}} Enrichment(E_j, \mathcal{T}, \mathcal{A}) \Leftrightarrow E_i = \varepsilon_{\mathcal{B}}(E_j)$
We first prove $\Rightarrow$:
$Enrichment(E_i, \mathcal{T}, \mathcal{A}) \equiv_{\varepsilon_{\mathcal{B}}} Enrichment(E_j, \mathcal{T}, \mathcal{A}) \Rightarrow E_i \equiv \varepsilon_{\mathcal{B}}(E_j)$

$$
\begin{aligned}
& Enrichment(E_i, \mathcal{T}, \mathcal{A}) \equiv_{\varepsilon_{\mathcal{B}}} Enrichment(E_j, \mathcal{T}, \mathcal{A}) \\
(defEnrich.) \Rightarrow & E_i \cup \{C(e_j) | (\mathcal{T}, \mathcal{A} \cup E_i) \models C(e_j), \forall e_j \in E_i\} \\
\equiv_{\varepsilon_{\mathcal{B}}} & E_j \cup \{C(e_j) | (\mathcal{T}, \mathcal{A} \cup E_j \models C(e_j), \forall e_j \in E_j\} \\
(def \equiv_{\varepsilon_{\mathcal{B}}}) \Rightarrow & E_i \cup \{C(e_j) | (\mathcal{T}, \mathcal{A} \cup E_i) \models C(e_j), \forall e_j \in E_i\} \\
\equiv & \varepsilon_{\mathcal{B}}(E_j \cup \{C(e_j) | (\mathcal{T}, \mathcal{A} \cup E_j \models C(e_j), \forall e_j \in E_j\}) \\
(def \varepsilon_{\mathcal{B}}) \Rightarrow & E_i \cup \{C(e_j) | (\mathcal{T}, \mathcal{A} \cup E_i) \models C(e_j), \forall e_j \in E_i\} \\
\equiv & \varepsilon_{\mathcal{B}}(E_j) \cup \varepsilon_{\mathcal{B}}(\{C(e_j) | (\mathcal{T}, \mathcal{A} \cup E_j \models C(e_j), \forall e_j \in E_j\}) \\
(def \equiv_{\varepsilon_{\mathcal{B}}}) \Rightarrow & E_i \equiv \varepsilon_{\mathcal{B}}(E_j)
\end{aligned}
$$

We now prove $\Leftarrow$:
$E_i \equiv \varepsilon_{\mathcal{B}}(E_j) \Rightarrow Enrichment(E_i, \mathcal{T}, \mathcal{A}) \equiv_{\varepsilon_{\mathcal{B}}} Enrichment(E_j, \mathcal{T}, \mathcal{A})$

$$
\begin{aligned}
& E_i \equiv \varepsilon_{\mathcal{B}}(E_j) \\
(def \cup) \Rightarrow & E_i \cup \{C(e_j) | (\mathcal{T}, \mathcal{A} \cup E_i \models C(e_j), \forall e_j \in E_i\} \\
\equiv & \varepsilon_{\mathcal{B}}(E_j) \cup \{C(e_j) | (\mathcal{T}, \mathcal{A} \cup E_i \models C(e_j), \forall e_j \in E_i\} \\
(def \equiv_{\varepsilon_{\mathcal{B}}}) \Rightarrow & E_i \cup \{C(e_j) | (\mathcal{T}, \mathcal{A} \cup E_i \models C(e_j), \forall e_j \in E_i\} \\
\equiv_{\varepsilon_{\mathcal{B}}} & \varepsilon_{\mathcal{B}}(E_j) \cup \varepsilon_{\mathcal{B}}(\{C(e_j) | (\mathcal{T}, \mathcal{A} \cup E_j \models C(e_j), \forall e_j \in E_j\}) \\
(def \varepsilon_{\mathcal{B}}) \Rightarrow & E_i \cup \{C(e_j) | (\mathcal{T}, \mathcal{A} \cup E_i \models C(e_j), \forall e_j \in E_i\} \\
\equiv_{\varepsilon_{\mathcal{B}}} & \varepsilon_{\mathcal{B}}(E_j \cup \{C(e_j) | (\mathcal{T}, \mathcal{A} \cup E_j \models C(e_j), \forall e_j \in E_j\}) \\
(defEnrich.) \Rightarrow & Enrichment(E_i, \mathcal{T}, \mathcal{A}) \equiv_{\varepsilon_{\mathcal{B}}} Enrichment(E_j, \mathcal{T}, \mathcal{A})
\end{aligned}
$$

The cache can be seen as a structure that verifies if a translation function between a new event and the stored event exists. For optimization purposes, we store the structure of the translation, i.e. the *Cache Structure*, that verifies the existence of a translation function.

7.5 Complexity Study

When performing a lookup in the cache, we need to evaluate the stored *Cache Structures* and check if one of these structures matches the incoming event. The last

step requires the whole event to be inspected. Thus for a cache of size n consisting of events of size m, the complexity for a cache look-up would be $O(n*m)$. However note that m, the size of the event, is typically very small ($m << n$), and can often be disregarded, resulting in a complexity of $O(n)$. Note that the size of the event reflects the number of individuals it contains.

Adding to the cache requires to materialize the event and create a *Cache Structure* from the event. The last step takes $O(m)$ for events of size m. The materialization step is more expensive and depends on the used logic. When using OWL2 DL, the complexity can be up to NEXPTIME, whereas the subprofiles (RL, QL, EL) remain in PTIME. Thus the complexity of adding to the cache would be $O(m + matTime)$ with $matTime$ the time to materialize the event which depends on the used logic. As m is typically much smaller than $matTime$ ($m << matTime$), we can simplify to $O(matTime)$. Note that adding to the cache might seem expensive, however, keep in mind that without the cache every event would be processed in $O(matTime)$. The whole purpose of the cache is to eliminate this step.

7.6 Cache Replacement Policies

To avoid that the cache grows to unmaintainable sizes, cache replacement policies can fix the size of the cache and define how the elements in the cache should be replaced when the size limit has been reached. We support the four typical types of replacement policies: First in first out (FIFO), Last in first out (LIFO), Least recently used (LRU), and Most recently used (MRU). The choice of the optimal replacement policy depends on the use case at hand. The LRU policy is the most popular cache replacement policy [23] and thus the standard configured replacement policy in our cache.

7.7 Relation with Query Rewriting

Query Rewriting is the process of rewriting the reasoning inside the query, such that no reasoning is required during query evaluation. This results in a much larger query, containing many UNIONs in order to include the logic defined in the TBox. Note that the larger the query and the more UNIONs it contains, the longer it takes to evaluate these queries. Furthermore, Query Rewriting for expressive logics as used within OWL2 DL is not possible. Our structural cache can be seen as a lazy query rewriting process. Compared to traditional rewriting techniques, our cache cannot perform the rewriting at query registration time, since a complete rewrite is not possible, however, it adds rewritten queries at run time to its cache. This results in an expressive rewritten query that can grow in size at run time and contains only the actually used concepts, instead of all possible rewritten concepts. The downside is that as the rewriting process cannot happen at query registration time, it needs to happen at run time, in a lazy fashion, i.e. only when reasoning steps need to be evaluated that are not yet added to the cache. On the upside, the cache will only contain a kind of rewritten query for the reasoning steps that are actually are being used and a much higher expressiveness can be obtained.

7.8 Implementation Details

We build our structural caching mechanism in a reasoner agnostic fashion by building upon the OWL-API [25] interfaces. This implies that our caching approach can be used around any reasoner implementing the OWL-API reasoner interface in order to speed up event enrichment. In the standard configuration, the HermiT reasoner is used to perform the materialization of the events combined with the static data. The storage and evaluation of the *Cache Structures* has been custom built[5], to assure the *exact matches* with the events (as explained in Section 7.3).

8 Evaluation

This section evaluates the performance of our cache and compares it with existing OWL2 reasoning techniques. We compare the scalability in terms of increasing static data and increasing cache size. We have used the OWL2Streams [8] benchmark for the evaluation of dynamic OWL2 reasoners, consisting of three scenarios:

1. A *Smart Building* case, on which the running example is based. CO_2 sensors are producing observations in a variety of building locations. Each location has a different set of activities. Based on the static and domain knowledge explained in Section 2, the CO_2 events need to be enriched as *AlertObservations*, i.e. when the CO_2 values are exceeding the allowed CO_2 threshold based on the activities occurring at the location of measuring. The static data describes properties about the locations and the sensors contained within each location. The data is synthetically generated based on the data distributions found in real smart buildings, producing streams up to 100.000 events with each an average size of three RDF triples.

2. A *University Management* scenario, which is an extension of the OWL2Bench benchmark [39] where the stream consists of students registering to a certain university. This is not an IoT use case, but as the stream consists of events that are similar in shape and size, it can be used for evaluation of our approach as well. Based on the courses that students are following, their hobbies and the type of university, each of the students can be enriched as a specific type. For example, a student with many hobbies, a leisure student, a self-aware student, etc. Each type is described in the domain knowledge. Note that the static data consist of the information of the university, its personnel and information of the courses. The data for this scenario is also generated, allowing to configure the size of the university. The streams produce up to 10.000 events, each with an average size of 16 RDF triples.

3. A *Smart City* scenario. This is an extension of the CityBench benchmark [2], containing more elaborate domain knowledge. In this smart city use case, various traffic observations throughout the city are captured, e.g. the congestion level, the average vehicle speed, parking occupation, etc. The version of City-Bench contained in OWL2Stream describes more expressive types for the observations that can be used for event enrichment, from simple type inference for observations based on their observe the property, e.g. *CongestionObservation* observe the property *CongestionLevel*, to observations that lead to alerts when

---

[5] The source code can be found on `https://github.com/IBCNServices/LOCERS`.
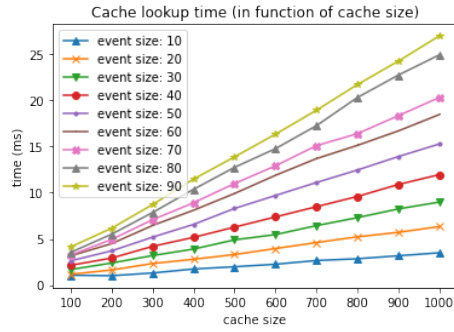
Fig. 11 Evaluation of the time to check the cache as a function of the cache size, with growing sizes of events.

high traffic has been observed near certain types of locations. Note that inferring the traffic levels and which locations require these kinds of alerts is also defined in the domain knowledge and is based on the types of roads, surroundings, and location properties. The static data describes information regarding the properties of the different roads and properties about the locations that are near the sensor observations. The data for this scenario is a replay of the data captured in the smart city of Aarhus, consisting of streams up to 15.000 events, each with an average size of 6 RDF triples. This scenario can be scaled by varying the number of sensors that produce data throughout the city.

Each of the scenarios consists of a data stream, a static knowledge base describing additional information regarding the events in the stream, and an expressive OWL2 DL ontology. All evaluations were conducted on the same hardware, i.e., Intel(R) Xeon(R) CPU E3-1220 with 16GB of ram running Ubuntu 18.04 LTS.

We first evaluate the scalability of the cache itself in terms of computation time and memory usage when the cache and the events increase in size. Next, we will compare the performance of the proposed cache with existing techniques for both materialization of the events, i.e. event enrichment, and query tasks. The last evaluation measures the performance of the different cache replacement policies.

8.1 Cache Scalability

To evaluate the scalability of the cache itself, we used the Smart Building scenario. In this evaluation we check how the cache reacts to:

1. Increasing cache size: number of *Cache Structures* stored.
2. Increasing event size: number of individuals in the events in the stream. Larger events will result in larger *Cache Structures*, and thus a larger number of *Variables* that needs to be evaluated.

In order to check how the cache performs with increasing cache size, we generated sensor streams originating from a large number of locations. In this case: 100, 200,..., till 1000 unique locations. As the locations are unique and have links with the static data, observations originating from a unique location results in a distinct *Cache Structure*. Thus with increasing number of unique locations, we
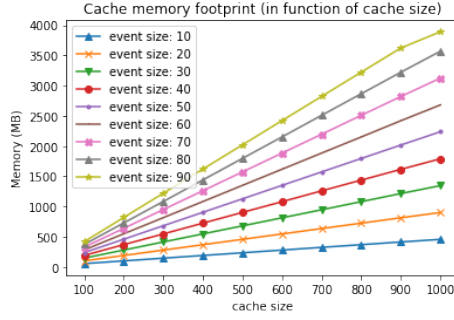
Fig. 12 Evaluation of the memory consumption of the cache as a function of the cache size, with growing sizes of events.

can grow the size of the cache. We also evaluate what happens if the sizes of the *Cache Structures* increase. This is artificially done by combining combinations of observations originating from unique locations, resulting in larger events. Thus, with increasing size of the events, the number of variables increases in the *Cache Structures*. It is important to note that this is only to really stress-test the cache. In realistic settings, event sizes rarely exceed size 10.

8.1.1 Time Performance

Figure 11 shows the influence of both increasing cache size (the number of stored *Cache Structures*) and size of the events. We depict the cache lookup time when the size of the cache increases while feeding various sizes of events. Note that larger events result in larger *Cache Structures*. We see that for small events (event size 10) the lookup times stay below 5ms, even for extremely large cache sizes. Even below 1ms for realistic cache sizes up to size 500. When increasing the size of the events, the lookup times increase as well. This was expected as larger *Cache Structures* need to be evaluated. In realistic settings, event size rarely exceed size 10, as events are kept small on purpose [47]. Note that the reported times represent the average cache look-up time over the whole scenario provided by OWL2Streams. As we are investigating the scalability of the cache itself, i.e. in growing number of variables and sizes of events, only cache hits are taken into account. The time for cache misses depends on the used reasoner that performs the materialization. We have used the HermiT reasoner as the underlying reasoner, which takes up to 8 seconds for events with size 100 for this specific use case.

8.1.2 Memory Consumption

Our structural cache is a typical example of the space-time tradeoff. In order to decrease reasoning time, we store *Cache Structures* and thus sacrifice memory. Figure 12 visualizes the memory consumption as a function of the size of the cache while storing different sizes of events. The memory increases in a rather linear fashion when the cache size increases or the size of the stored *Cache Structures* increases. Note that while the cache size might become large, the size of events

typically stays rather small. In our experiments, the size of the events in the streams did not exceed size 10, thus resulting in very limited memory impact.

8.2 Comparison with Existing Reasoners for Event Enrichment tasks

The second evaluation evaluates the scalability in terms of increasing static data to compare the performance of the cache with the following reasoning approaches:

– OWL2 DL reasoners Hermit [38] and Pellet [40]: these are the baselines in terms of performance, as they are complete in terms of the expressivity of the used ontologies. JFact and Konclude were initially included in the evaluation, however, they either did not reach a conclusion within one hour or crashed.
– OWL2 EL reasoner ELK [30] and OWL2 RL reasoner RDFox [34]: these reasoners are added as a reference only. They are a lot faster, as they do not infer all facts due to the lower supported expressivity. This is indicated in the completeness columns of the various result tables. We wanted to investigate if our cache could even have added value for these OWL2 reasoners with lower expressiveness. Note that both ELK and RDFox allow incremental reasoning, i.e. when new data arrives, they can incrementally update their materialization, without the need to start the reasoning process from scratch.
– Approximation approaches TrOWL [44] and Subset Reasoner [10]: these reasoners support OWL2 DL expressivity, however, make some approximations to speed up the reasoning over data steams, possibly resulting in incompleteness. A form of incremental maintenance is provided, but as an approximation.
– A rule engine approach that uses the part of the ontology that can be expressed as rules. We used the RETE-based rule engine that is part of Jena[6] as it can process RDF data and allows querying using SPARQL. This approach is also added as a reference only and can perform incremental reasoning.

We evaluate the performance on the different scenarios described above to perform Event Enrichment on the events in each stream. This means that the events are combined with the static data and the domain knowledge and all possible axioms with respect to the individuals in the events are inferred. For each approach, we compute the completeness, i.e. the average percentage of correctly inferred axioms over all events in the stream. For each scenario, we grow the static data, as the reasoning performance typically decreases with growing datasets and thus stresses their performance. Note that the reported times are the average time to perform event enrichment over all the events in the whole scenario.

8.2.1 Stream of Students

We start with the extension of OWL2Bench, where the stream consists of students registering to a certain university. The students follow certain courses and have various hobbies and interests. Various students can follow the same curriculum or have the same interests. Even though these are different students, they are classified in the same way. This is the type of students that are matched by the cache. Instead of growing the number of universities, we grow the number of departments

---

[6] jena.apache.org

Table 1 Evaluation of performing event enrichment over the student streams extracted from OWL2Bench. The reasoning time is shown as a function of the number of departments in the static data. Furthermore, the completeness of each approach is shown. All reported times are in milliseconds. (TO = Time Out, compl. = completeness)

engine	compl.	#departments				
		1	5	10	15	20
Pellet [40]	**100%**	1905.52	TO	TO	TO	TO
HermiT [20]	**100%**	1823.93	6166.73	15403.37	15730.57	35012.41
Subset [10]	**100%**	1160.12	1776.92	708.94	413.14	2498.76
TrOWL [44]	76%	28.44	162.31	507.76	1052.77	1864.25
RDFox [34]	72%	10.42	8.07	7.19	9.48	7.34
ELK [30]	67%	3.89	4.19	4.66	5.91	5.76
Jena	76%	109.82	403.72	851.07	1413.72	6343.59
Cache	**100%**	**0.79**	**0.74**	**0.75**	**0.63**	**0.98**

Table 2 Evaluation of materializing traffic observations of CityBench. The reasoning time is shown as a function of the number of intersections in the city. Furthermore, the completeness of each approach is shown. Pellet was not included as it was producing errors while handling the data property reasoning. All reported times are in milliseconds. (compl. = completeness)

engine	compl.	#road intersections					
		1	100	200	400	600	800
HermiT [20]	**100%**	112.68	749.06	1807.75	4724.34	9419.08	14197.38
Subset [10]	**100%**	686.19	673.41	673.73	667.28	672.52	672.91
TrOWL [44]	52%	17.37	33.47	48.29	80.44	135.08	196.72
RDFox [34]	85%	6.30	5.91	6.03	5.70	5.58	7.58
ELK [30]	74%	34.00	60.24	89.67	148.33	230.15	332.35
Jena	81%	42.53	146.92	300.09	622.25	707.28	1398.99
Cache	**100%**	**0.62**	**0.72**	**0.67**	**0.84**	**0.68**	**0.66**

within a single university. The reasoning time for expressive reasoners to reason over more than one university is far beyond the real-time requirements [39], and thus not considered. Table 1 shows the results for each of the reasoning techniques to enrich events describing students. To make a fair comparison, we ask the reasoners to only infer the types of the individuals contained in the events and not to materialize the whole static data. We see that the fully-fledged reasoners become very slow very quickly with increasing static background. Pellet timed out after 5 departments. Our caching technique is the fastest in event materialization, even faster than the approximation approaches or the less expressive reasoners such as RDFox and ELK or the RETE-based approach using Jena. Note that our cache produces results that obtain 100% completeness for the task of event enrichment. We note that these are the times when the events have already been added to the cache, as the cache internally uses the HermiT reasoner, upon cache misses the same performance as HermiT can be expected. While the overhead to add events to the cache is minimal as well, only a few milliseconds.

8.2.2 Traffic Stream

Compared to the student stream, the streams considered in the Smart City require reasoning on a data property level, which is a typical requirement in the IoT. The observations get classified differently based on the sensed values and their location. In the previous evaluation, no data property reasoning was considered. As the cache supports it, we evaluate it here as well. Table 2 shows the results for each of

Table 3 Evaluation of materializing traffic observations of CityBench for each reasoner equipped with our cache. We show the speed-up time as a result of using the cache for each reasoner over the whole scenario.

engines with cache	AVG Speed up	#road intersections					
		1	100	200	400	600	800
HermiT [20]	1773.70	2197.80	814.75	1346.20	1882.59	2146.86	2254.02
Subset [10]	758.11	767.35	757.39	757.64	752.57	756.69	757.00
TrOWL [44]	126.94	27.72	52.86	75.56	123.39	200.48	281.66
RDFox [34]	9.94	10.12	9.50	9.69	9.16	8.97	12.17
ELK [30]	211.22	53.68	93.56	136.77	218.43	323.33	441.55
Jena	562.52	66.79	216.53	405.64	716.34	783.55	1186.24

the reasoners for increasing city sizes, i.e. number of intersections where the traffic sensors are positioned. We can see that the cache is able to enrich the events in less than one millisecond, and it is clearly the fastest approach, even faster than the less expressive reasoners. While being very scalable, the cache produces results with the same completeness as the fully-fledged OWL2 DL reasoners. Note that these are the reported times for cache hits, cache misses have similar time consumption as the HermiT reasoner, as HermiT is used under the hood by our approach.

Table 3 details the speed up times of equipping each evaluated engine with our cache on the Smart City scenario. As our cache is reasoner agnostic, it can easily be combined with other reasoners, allowing these reasoners to benefit from the speed up from the cache. We see major improvements in reasoning speed up times for HermiT, i.e. thousands of times faster, while still hundreds of times faster for less expressive reasoners in the EL profile, such as ELK, and up to 10 times faster for the OWL2 RL reasoner RDFox. The speed up depends on the hit rate of the cache, in this scenario the hit rate was up to 99.96%. These high hit rates can be expected in IoT scenarios where the structure of the events is fixed due to the format of the sensor output.

Figure 13 shows the scenario run time for the first 100 events and compares the reasoning times between HermiT and our cache. There occur three cache misses (resembled by the three peaks), which result in the same reasoning times as HermiT. The other scenario steps are cache hits, resulting in great performance gains for our cache.
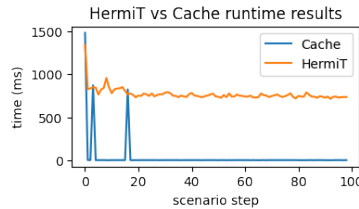

Fig. 13 Scenario run time comparison between HermiT and the Cache.

8.3 Comparison with Existing Reasoners for Query tasks

We have evaluated the cache for materialization tasks above, now we compare the performance of the cache with query driven approaches. Note that query driven approaches do not necessarily need to fully materialize the dataset before answering a certain query, but perform the reasoning necessarily to answer the query at query time. We compare these query driven techniques with our cache. Our cache performs event enrichment as usual with the additional step of query evaluation on top of the performed materialization. Thus, the reported times for our cache

Table 4 Evaluation of query task on the students streams extracted from OWL2Bench. The reasoning and query time is shown as a function of the number of departments in the static data. Furthermore, the completeness of each approach is shown. All reported times are in milliseconds.(QR = Query Rewriting, TO = Time Out, compl. = completeness)

engine	compl.	#departments				
		1	5	10	15	20
Pellet [40]	**100%**	776.46	TO	TO	TO	TO
HermiT [20]	**100%**	1225.31	17017.50	61481.21	132448.14	228852.50
TrOWL [44]	29.5%	28.78	186.94	532.61	1092.18	1841.70
QR [12]	48.1%	40.63	87.87	127.38	178.20	217.43
ELK [30]	29.5%	**4.06**	10.00	11.71	11.63	12.86
Cache	**100%**	5.79	**6.69**	**6.29**	**6.22**	**6.47**

are both event enrichment and query evaluation times. Note that for our cache, no reasoning is required at query evaluation time, due to the materialization process.

The OWL2Bench benchmark is accompanied by a set of SPARQL queries. We have selected all the type queries[7] as they allow to easily translated to DL-queries and answered by the reasoners that do not support SPARQL out of the box. For the query evaluation task, we compare against the following reasoners:

- OWL2 DL reasoners Hermit [38] and Pellet [40] as baselines and evaluate DL-queries.
- The lesser expressive OWL2 EL reasoner ELK [30] evaluating DL-queries.
- The approximation approach TrOWL [44] evaluating DL-queries.
- A Query Rewriting approach [12] that evaluates large rewritten SPARQL queries that contain the TBox logic instead of reasoning.

Table 4 shows the results of the evaluation of the type queries as a function of growing static data. Table 4 also shows the completeness, which is computed as the mean of the percentage of correct answers for each query. The cache outperforms the other approaches in terms of completeness and reasoning/query answering time. Even the Query Rewriting is slower than the cache due to the very large queries that need to be evaluated. Note that some of the queries are very large and contain up to 324 UNIONs. The times shown in Table 4 for our cache are a little higher than in Table 1 as both the event enrichment and the querying of the whole static data needs to be performed. For our cache approach, a conversion is necessary to allow the evaluation of SPARQL queries[8], hence the small increase in time. It is important to note that the caching approach assumes a fully materialized version of the static data, which is expensive to compute. Luckily, the static data does not change very often and can be computed as a preprocessing step.

8.4 Cache Replacement Strategies

Next, we evaluate the different cache replacement strategies for the Smart City scenario. Figure 14 visualizes the cache hit rates as a function of cache size and replacement policy for the Smart City scenario consisting of traffic streams resulting from 20 unique locations, producing a total of 60 distinct events. The MRU and

---

[7] Queries of the form: Select ?s WHERE{?s a :someType}

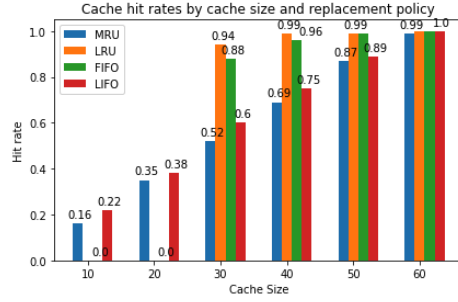[8] We used Jena for the evaluation of the queries.

Fig. 14 Evaluation of different cache replacement policies. Hit rates for each policy as a function of the cache size.

LIFO policies work best for smaller cache sizes, while LRU and FIFO work best for large cache sizes. Once the cache sizes exceed the size of 20, LRU and FIFO perform really well. This is due to the fact that there are 20 unique locations that send their observations around the same time, resulting in 20 unique events, making it impossible for LRU or FIFO policies to achieve a cache hit. The computation overhead for each of the policies is very limited and only around 0.003 ms.

9 Discussion

In this section, we discuss how our caching approach tackles the set objectives from Section 1, provide some additional insights into the evaluation results, and discuss the limitations of our caching approach.

9.1 Objectives Discussion

Looking back at the Objectives set in Section 1, we can now discuss how our caching approach tackles the various objectives:

1. *Expressive reasoning:* our cache can be used with any OWL2 reasoner implementing the OWL API interfaces and provides huge performance gains for OWL2 DL reasoners and even for lesser expressive OWL2 EL and RL reasoners. We have shown that our caching approach is complete and correct for the task of event enrichment when reasoning over expressive OWL2 DL ontologies.
2. *Eliminate reoccurring reasoning steps*: our caching approach can store the reasoning results by exploiting the fact that events in IoT streams typically have the same structure and size. Through a smart look-up mechanism that takes the static data into account, it can eliminate reasoning steps when a similar event has been processed in the past. In this case, the reasoning results can be transferred to the new event, moving from reasoning to pattern matching.
3. *Scalable*: as seen in the evaluation, the cache itself scales really well in the number of stored events and size of the events. Furthermore, as performing event enrichment through our caching approach is just a simple cache look-up, it can be easily distributed, allowing horizontal scaling as well. A synchronization step would be required when one of the distributed caches has a miss and a

new event is added to the cache. In this case, the other cache instances would require the new event as well.

4. *Integrate static knowledge:* as our caching approach can differentiate between static and streaming data, all connections with the static knowledge base are kept. This allows to correctly perform event enrichment over a variety of events, that have similar structure and size.

9.2 Evaluation Discussion

In Section 8, we have first evaluated the scalability of the cache in terms of the number of stored events and the size of the events. We have seen that for smaller size events, the cache scales very well, even with large amounts of events stored. For larger events, the cache tends to become slower as more events are stored. Fortunately, the size of the events in the IoT, and in stream processing in general, are typically kept small, making the cache ideal for these situations. We also see that as investigated in Section 7.5, the complexity of the cache is indeed linear in the size of the cache.

In comparison with existing reasoners, we have shown that our cache scales really well with increasing size of static background knowledge. This is because the other reasoners need to take larger amounts of data into account during reasoning when the size of the static background knowledge increases. Our cache circumvents this problem by turning to pattern matching in order to eliminate reoccurring reasoning steps. It is important to stress that this is only in the case when there is a cache hit, if not, the same reasoning times can be expected as for the traditional reasoners. As the HermiT reasoner is used under the hood to fill the cache in case of a cache miss, the same reasoning times as HermiT can be expected. However, as the events in an IoT setting are typically very similar in shape and size, the cache can provide very large performance gains, with very limited overhead. If one is willing to sacrifice some memory consumption to increase performance during event enrichment, then the cache is very beneficial.

We note that even for lesser expressive reasoners, such as RDFox and ELK that support the lesser expressive OWL2 RL and OWL2 EL profiles, our cache can still provide large performance gains. Our cache still outperforms these reasoners and is at least ten times faster. (And up to 300 times for large datasets using ELK.) This means that for streams with similar structure and size, we can employ expressive OWL2 DL reasoning and still be competitive in terms of event enrichment times compared to lesser expressive and more performant reasoners such as RDFox and ELK. This is very beneficial, as most of the IoT ontologies require OWL2 DL reasoning to be fully interpreted [10]. On the other hand, this also means that we can use our cache in combination with RDFox or ELK and still gain large performance gains. At this point, the OWL2 DL reasoner HermiT is used for the materialization of the events upon a cache miss. We can easily switch to another reasoner to perform this initial materialization.

9.3 Limitations

Our approach makes a number of assumptions, limiting its use to specific scenarios. We now go deeper into the limitations of the fact that events should have similar structure and size and that we focus specifically on the task of event enrichment.

9.3.1 Fixed Event Structure

The cache is optimized to exploit reoccurring reasoning steps over streams that contain events that are similar in shape and size. This is typical in an IoT setting where various sensors produce observations. Each observation is unique, in the sense that the observed value is unique, the sensor that made the observation may vary or the observed property. However, these observations are all still mostly the same structure and size. Outside the IoT this assumption may hold as well, e.g. as we have seen in the university management scenario in Section 8. However, in other Stream Processing scenarios such as Social Media analysis, this hypothesis typically does not hold. Social media posts can have a different number of likes, followers, comments, and so on. This variability in the number of properties would lead to a huge number of cache misses and thus minimize the benefits of the cache. At this point, the cache can thus not be used in scenarios with variable event structures. However, evaluating if the event structure is fixed is possible. When the original generated data is not in the RDF format, a mapping step is necessary to map it to the RDF model, e.g. through RML [17]. By analyzing the mapping file that indicates how the raw data needs to be mapped to semantic data, we can be sure that the structure of the resulting stream is fixed. If the stream is original an Ontology stream, and thus no mapping phase is available for analysis, still we can monitor the structure of the events and evaluate how much they vary before employing our cache solution. The latter can easily be done with our cache itself, by adding an alert on the number of stored events.

9.3.2 Event Enrichment

As indicated in Section 3, our approach is optimized for *Event Enrichment*, i.e. we only infer the types of the individuals that were contained within the event. Thus if the event invokes a change of types in the static data, that was not linked in the event, then these changes will not be captured. The latter is the problem of *Static Enrichment*, which we did not study here. Figure 15 shows a visual explanation between the different enrichments. In theory, the possible enrichments of the static data could be cached as well. The only requirement is that the reasoner provides some intuition which additional types have been added to the static data. HermiT does not allow to extract these insights, requiring to query the whole static knowledge base for possible changes. This is by no means a scalable solution as 1) the larger the static data, the more data needs to be queried, and 2) one needs to maintain a list with all the current types of all individuals in the static data to be able to verify if additional types have been added through the reasoning process. Reasoners that do provide this intuition allow to use the cache for both *Event* and *Static Enrichment*.
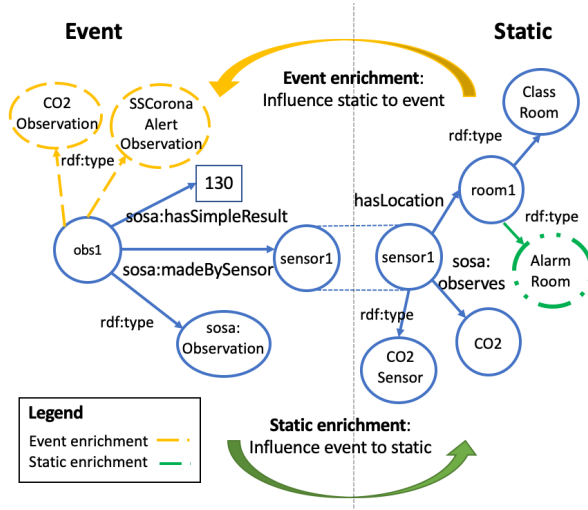
Fig. 15 Difference between event and static enrichment. Event enrichment: materialization based on the influence of the static data on the event. Static enrichment: materialization based on the influence of the event on the static data. The event is enriched as a CO2Observation, it needs the static knowledge stating that the sensor observes CO2. It is also a CoronaAlertObservation because it is a CO2Observation measuring a value above 100 in a ClassRoom. The static knowledge is enriched as an AlarmRoom, as one of the sensors in the room has captured an AlertObservation.

10 Conclusion & Future Work

In this paper, we presented a caching technique to solve the mismatch between expressive reasoning and real-time processing requirements in data-intensive domains. In order to significantly speed up reasoning time for the task of event enrichment, we exploit the fact that events in IoT data streams are often similar in structure and size. Our caching technique is ideal for the semantic enrichment of events, in combination with rich backgrounds and complex domain knowledge. In these cases, the proposed caching technique is up to thousands of times faster than fully-fledged OWL2 DL reasoners for event enrichment, while preserving the correct and completeness of reasoning. The caching technique is beneficial even for less expressive profiles, such as OWL2 RL and EL, where we obtain up to tenths and hundreds faster reasoning times.

In our future work, we will investigate the possibility to pre-fill the cache, based on the stream description such as provided by VoCaLS [46]. To further limit memory consumption, we will incorporate various encoding schemes. We will also investigate how to exploit our cache for non-IoT streams, i.e. streams consisting of events that have no similar structure or size. This is typical in Social Media analysis. We also aim to extend the technique to enable the enrichment of windows of events, while we currently only considered single events. In this paper we have limited the scope to the enrichment of the events only, disregarding the enrichment of the static data. We will further extend our technique to include both enrichment influences. This will allow to further bridge the gap between expressivity and efficiency of reasoning over high-velocity data streams.

References

1. Adams, T., Dullea, J., Clark, P., Sripada, S., Barrett, T.: Semantic integration of heterogeneous information sources using a knowledge-based system. In: Proc 5th Int Conf on CS and Informatics (CS&I'2000). Citeseer (2000)
2. Ali, M.I., Gao, F., Mileo, A.: Citybench: A configurable benchmark to evaluate RSP engines using smart city datasets. In: ISWC, pp. 374–389. Springer (2015)
3. Baader, F., Calvanese, D., McGuinness, D., Patel-Schneider, P., Nardi, D.: The description logic handbook: Theory, implementation and applications. Cambridge university press (2003)
4. Barbieri, D.F., Braga, D., Ceri, S., Della Valle, E., Grossniklaus, M.: Incremental reasoning on streams and rich background knowledge. In: Extended Semantic Web Conference, pp. 1–15. Springer (2010)
5. Barbieri, D.F., Braga, D., Ceri, S., VALLE, E.D., Grossniklaus, M.: C-sparql: a continuous query language for rdf data streams. International Journal of Semantic Computing **4**(01), 3–25 (2010)
6. Barnaghi, P., Wang, W., Henson, C., Taylor, K.: Semantics for the Internet of Things: Early Progress and Back to the Future. International Journal on Semantic Web and Information Systems (IJSWIS) **8**, 1–21 (2012)
7. Bazoobandi, H.R., Bal, H., van Harmelen, F., Urbani, J.: Handling impossible derivations during stream reasoning. In: ESWC, pp. 3–19. Springer (2020)
8. Bonte, P., Ongenae, F.: OWL2Streams a benchmark for expressive stream reasoning for dynamic owl2 reasoners (2020). URL `https://github.com/IBCNServices/OWL2Streams`
9. Bonte, P., Ongenae, F., De Backere, F., Schaballie, J., Arndt, D., Verstichel, S., Mannens, E., Van de Walle, R., De Turck, F.: The massif platform: a modular and semantic platform for the development of flexible iot services. Knowledge and Information Systems **51**(1), 89–126 (2017)
10. Bonte, P., Ongenae, F., De Turck, F.: Subset reasoning for event-based systems. IEEE Access **7**, 107533–107549 (2019)
11. Bonte, P., Tommasini, R., Della Valle, E., De Turck, F., Ongenae, F.: Streaming MASSIF: cascading reasoning for efficient processing of IoT data streams. Sensors **18**(11), 3832 (2018)
12. Calbimonte, J.P., Mora, J., Corcho, O.: Query rewriting in rdf stream processing. In: European Semantic Web Conference, pp. 486–502. Springer (2016)
13. Compton, M., Barnaghi, P., Bermudez, L., Garcia-Castro, R., Corcho, O., Cox, S., Graybeal, J., Hauswirth, M., Henson, C., Herzog, A., et al.: The ssn ontology of the w3c semantic sensor network incubator group. Journal of Web Semantics **17**, 25–32 (2012)
14. Della Valle, E., Ceri, S., Van Harmelen, F., Fensel, D.: It's a streaming world! reasoning upon rapidly changing information. IEEE Intelligent Systems **24**(6), 83–89 (2009)
15. Della Valle, E., Dell'Aglio, D., Margara, A.: Taming velocity and variety simultaneously in big data with stream reasoning: tutorial. In: DEBS, pp. 394–401 (2016)
16. Dell'Aglio, D., Della Valle, E., van Harmelen, F., Bernstein, A.: Stream reasoning: A survey and outlook. Data Science (Preprint), 1–24 (2017)
17. Dimou, A., Vander Sande, M., Colpaert, P., Verborgh, R., Mannens, E., Van de Walle, R.: Rml: a generic language for integrated rdf mappings of heterogeneous data. In: Ldow (2014)
18. Dodaro, C., Eiter, T., Ogris, P., Schekotihin, K.: Managing caching strategies for stream reasoning with reinforcement learning. Theory and Practice of Logic Programming **20**(5), 625–640 (2020)
19. Giustozzi, F., Saunier, J., Zanni-Merk, C.: Context modeling for industry 4.0: An ontology-based proposal. Procedia Computer Science **126**, 675–684 (2018)
20. Glimm, B., Horrocks, I., Motik, B., Stoilos, G., Wang, Z.: Hermit: an OWL 2 reasoner. Journal of Automated Reasoning **53**(3), 245–269 (2014)
21. Glimm, B., Kazakov, Y., Tran, T.K.: Ontology materialization by abstraction refinement in horn shoif. In: AAAI, pp. 1114–1120 (2017)

22. Gruber, T.R.: Toward principles for the design of ontologies used for knowledge sharing? International journal of human-computer studies **43**(5-6), 907–928 (1995)
23. Guo, F., Solihin, Y.: An analytical model for cache replacement policy performance. In: Proceedings of the joint international conference on Measurement and modeling of computer systems, pp. 228–239 (2006)
24. Heyvaert, P., De Meester, B., Dimou, A., Verborgh, R.: Declarative rules for linked data generation at your fingertips! In: European Semantic Web Conference, pp. 213–217. Springer (2018)
25. Horridge, M., Bechhofer, S.: The owl api: A java api for owl ontologies. Semantic web **2**(1), 11–21 (2011)
26. Horrocks, I., Kutz, O., Sattler, U.: The even more irresistible SROIQ. Kr **6**, 57–67 (2006)
27. Hustadt, U., Motik, B., Sattler, U.: Data complexity of reasoning in very expressive description logics. In: IJCAI, vol. 5, pp. 466–471 (2005)
28. Isah, H., Abughofa, T., Mahfuz, S., Ajerla, D., Zulkernine, F., Khan, S.: A survey of distributed data stream processing frameworks. IEEE Access **7**, 154300–154316 (2019)
29. Jordan, H., Subotić, P., Zhao, D., Scholz, B.: A specialized b-tree for concurrent datalog evaluation. In: Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming, pp. 327–339 (2019)
30. Kazakov, Y., Krötzsch, M., Simancik, F.: Elk reasoner: Architecture and evaluation. In: ORE (2012)
31. Keskisärkkä, R., Blomqvist, E., Hartig, O.: Optimizing rdf stream processing for uncertainty management. In: Further with Knowledge Graphs, pp. 118–132. IOS Press (2021)
32. Le-Phuoc, D., Dao-Tran, M., Xavier Parreira, J., Hauswirth, M.: A Native and Adaptive Approach for Unified Processing of Linked Streams and Linked Data, pp. 370–388. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
33. Nelis, J., Verschueren, T., Verslype, D., Develder, C.: Dyamand: dynamic, adaptive management of networks and devices. In: 37th Annual IEEE Conference on Local Computer Networks, pp. 192–195. IEEE (2012)
34. Nenov, Y., Piro, R., Motik, B., Horrocks, I., Wu, Z., Banerjee, J.: Rdfox: A highly-scalable rdf store. In: ISWC, pp. 3–20. Springer (2015)
35. Pan, J.Z.: Resource description framework. In: Handbook on ontologies, pp. 71–90. Springer (2009)
36. Peng, Z., Jimenez, J.L.: Exhaled CO2 as COVID-19 infection risk proxy for different indoor environments and activities. medRxiv (2020)
37. Petrolo, R., Loscri, V., Mitton, N.: Towards a smart city based on cloud of things, a survey on the smart city vision and paradigms. Transactions on Emerging Telecommunications Technologies **28**(1), e2931 (2017)
38. Shearer, R., Motik, B., Horrocks, I.: Hermit: A highly-efficient owl reasoner. In: OWLED, vol. 432, p. 91 (2008)
39. Singh, G., Bhatia, S., Mutharaju, R.: Owl2bench: A benchmark for owl 2 reasoners. In: International Semantic Web Conference, pp. 81–96. Springer (2020)
40. Sirin, E., Parsia, B., Grau, B.C., Kalyanpur, A., Katz, Y.: Pellet: A practical OWL-DL reasoner. Web Semantics: science, services and agents on the World Wide Web **5**(2), 51–53 (2007)
41. Steigmiller, A., Liebig, T., Glimm, B.: Extended caching, backjumping and merging for expressive description logics. In: IJCAR, pp. 514–529. Springer (2012)
42. Stuckenschmidt, H., Ceri, S., Della Valle, E., Van Harmelen, F.: Towards expressive stream reasoning. In: Dagstuhl Seminar Proceedings. Schloss Dagstuhl-Leibniz-Zentrum für Informatik (2010)
43. Teymourian, K., Paschke, A.: Semantic enrichment of event stream for semantic situation awareness. In: Semantic Web, pp. 185–212. Springer (2016)
44. Thomas, E., Pan, J.Z., Ren, Y.: TrOWL: Tractable OWL 2 reasoning infrastructure. In: Extended Semantic Web Conference, pp. 431–435. Springer (2010)
45. Tommasini, R., Bonte, P., Ongenae, F., Della Valle, E.: Rsp4j: An api for rdf stream processing. In: European Semantic Web Conference, pp. 565–581. Springer (2021)
46. Tommasini, R., Sedira, Y.A., Dell'Aglio, D., Balduini, M., Ali, M.I., Le Phuoc, D., Della Valle, E., Calbimonte, J.P.: Vocals: Vocabulary and catalog of linked streams. In: International Semantic Web Conference, pp. 256–272. Springer (2018)
47. Westermann, U., Jain, R.: Toward a common event model for multimedia applications. IEEE multimedia **14**(1), 19–29 (2007)