Efficient Temporal Reasoning on Streams of Events with DOTR

Alessandro Margara[1(✉)], Gianpaolo Cugola[1], Dario Collavini[1], and Daniele Dell'Aglio[2]

[1] DEIB, Politecnico di Milano, Milan, Italy
{alessandro.margara,gianpaolo.cugola,dario.collavini}@polimi.it
[2] IFI, University of Zurich, Zurich, Switzerland
dellaglio@ifi.uzh.ch

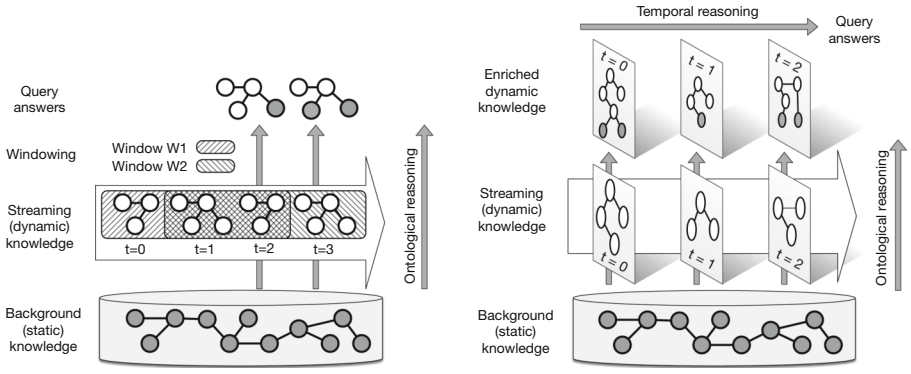
Abstract. Many ICT applications need to make sense of large volumes of streaming data to detect situations of interest and enable timely reactions. Stream Reasoning (SR) aims to combine the performance of stream/event processing and the reasoning expressiveness of knowledge representation systems by adopting Semantic Web standards to encode streaming elements. We argue that the mainstream SR model is not flexible enough to properly express the temporal relations common in many applications. We show that the model can miss relevant information and lead to inconsistent derivations. Moving from these premises, we introduce a novel SR model that provides expressive ontological and temporal reasoning by neatly decoupling their scope to avoid losses and inconsistencies. We implement the model in the DOTR system that defines ontological reasoning using Datalog rules and temporal reasoning using a Complex Event Processing language that builds on metric temporal logic. We demonstrate the expressiveness of our model through examples and benchmarks, and we show that DOTR outperforms state-of-the-art SR tools, processing data with millisecond latency.

1 Introduction

Many information systems need to make sense of large volumes of data as soon as they are produced to detect relevant situations and enable prompt reactions. Areas of application include smart cities, fraud detection systems, and social media analysis. These scenarios demand for processing abstractions and tools to "reason" on streaming data, both in ontological and temporal terms, while also coping with the volume, velocity, and variety of streaming data. More concretely, they require: (1) flexible data models to integrate heterogeneous data coming from multiple sources; (2) integration with background knowledge that describes the application domain; (3) expressive (temporal) reasoning on both streaming and background data; (4) high throughput and low latency.

Several systems have been developed in the last decade to address this problem, but none of them simultaneously tackles the requirements above [11,12]. Stream Processing (SP) systems [3] focus on continuous query answering: they

A. Gangemi et al. (Eds.): ESWC 2018, LNCS 10843, pp. 384–399, 2018.
https://doi.org/10.1007/978-3-319-93417-4_25



(a) Traditional approach to SR. (b) DOTR model.

Fig. 1. Comparison of the traditional model to SR and the proposed DOTR model.

use a recent portion of the streaming data to update the results of queries as new data becomes available. They mostly provide a temporal (relational) operations on structured (relational) data. Instead, Complex Event Processing (CEP) systems [14] are specialized in detecting the occurrence of temporal patterns in the stream of input elements. Both SP and CEP systems provide high throughput and low latency, but struggle at integrating heterogeneous data and at exploiting background knowledge on the application domain.

Recently, Stream Reasoning (SR) systems [12,19] addressed these limitations by adopting Semantic Web standards to represent information, thus enabling expressive reasoning on streaming data from heterogeneous sources and static background knowledge. Similar to SP systems, most SR systems continuously update the results of standing SPARQL queries over static background knowledge and dynamic streaming data. They use *window* operators to isolate the recent portion of data to be considered, such as `W1` and `W2` in Fig. 1(a). Ontological reasoning (when supported) assumes that *all* and *only* the information that is in the current window (plus the background knowledge) holds. SPARQL queries are then applied to both the original and the inferred knowledge, and the process is repeated any time the window content changes [5,8,17].

We claim that this model is not flexible enough to satisfy all the requirements identified above. Although some systems have extended SPARQL with temporal operators [2], the model does not support expressive temporal reasoning. On the other hand, using windows to determine the scope of reasoning and querying can result in information loss or inconsistent derivations (see Sect. 2). In summary, the problem of complementing expressive reasoning on rich ontological knowledge with temporal reasoning in a coherent yet efficient way remains open.

Moving from these premises, we propose a novel approach to SR called DOTR (Decoupled Ontological and Temporal Reasoning). It provides ontological reasoning on streaming and background knowledge, and efficient detection of temporal patterns (temporal reasoning), while keeping the two form of reasoning

sharply decoupled. In DOTR (Fig. 1(b)), the incoming stream elements represent events that occur at some point in time, encoded as time-annotated RDF graphs. Ontological reasoning takes place at each point in time separately, combining the events happening at that time with the background knowledge on the application domain. Temporal reasoning is applied separately: it considers enriched events—as determined in the ontological reasoning step—and searches for temporal patterns to derive the relevant consequences of what is happening. This decoupling allows combining established semantics, mechanisms, and tools in the domain of ontological reasoning with those available for temporal reasoning. In particular, we implement DOTR in a prototype system that provides ontological reasoning through Datalog rules and temporal reasoning through the TESLA event processing language, which grounds on metric temporal logic. We demonstrate through benchmarks and case studies the benefits of DOTR with respect to traditional SR approaches in terms of expressiveness and performance. We evaluate DOTR under different workloads and show that it can process input data with a latency of few milliseconds even in the presence of large knowledge bases and complex inference tasks.

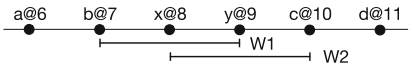
The paper is organized as follows. Section 2 presents background information on SR and motivates our work. Sections 3 and 4 present our model and its implementation, while Sect. 5 evaluates its performance. Section 6 discusses related work and Sect. 7 concludes the paper drawing future research directions.

2 Background and Motivations

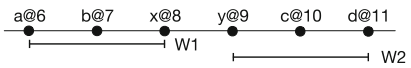
This section introduces the terminology and concepts that we use in the remainder of the paper and discusses the motivations underneath our work.

We denote a *stream* as a (possibly unbounded) sequence of time-annotated elements ordered according to temporal criteria. Each element brings a unit of information, such as a sensor observation or a stock exchange [11,12]. Individual elements can be represented in different formats: for instance, SP systems often adopt a relational model, whereas SR systems like DOTR promote data integration and expressive reasoning by using the RDF format for stream elements [12,19]. SP, CEP, and SR systems continuously evaluate standing *rules* or *queries* against stream elements. Rule evaluation can be either periodic or triggered by the incoming elements. The dominant approach to SR defines rules as SPARQL queries and adopts *window operators* to determine the scope for evaluating such queries [5,6,8,17]. Window operators create finite views over a stream, namely *windows*, that include the portion of data relevant for the current evaluation of rules. At each evaluation, they produce either the complete results of the processing or the differences—additions/deletions—with respect to the previous evaluation. Although several types of window operators have been defined, the most common are sliding windows, which have a fixed width in terms of time units or data elements and shift (slide) over time, always capturing the most recent part of the stream.

We argue that the processing model based on windows is not adequate to capture temporal relations and can result in undesired (i) duplicate results,



(a) Window of size 3 and slide 1. The pattern (`x@8`, `y@9`) is detected twice.



(b) Window of size 3 and slide 3. The pattern (`x@8`, `y@9`) remains undetected.

Fig. 2. Processing model based on windows: examples of limitations.

(ii) loss of information, or (iii) inconsistency in reasoning. Figure 2 exemplifies the first two problems. It represents each stream element with a label `l` and its time of occurrence `t` as `l@t`. Imagine we want to detect whenever an element labeled `x` is followed by an element labeled `y` after no more than three time units. Figure 2(a) shows that in the presence of sliding windows, the same occurrence might be detected multiple times. For instance, the sequence `x@8` and `y@9` (denoted `x@8, y@9`)) is detected both in window `W1` and in window `W2`, requiring additional downstream logic in the case the developer wants to prune duplicate reports of the same occurrence. To avoid this problem, developers can adopt tumbling windows, which partition the input stream in non-overlapping chunks. Unfortunately, this solution might result in missing some occurrences of the pattern of interest. For instance in Fig. 2(b), the sequence (`x@8`, `y@9`) is not detected since only `x@8` is part of `W1` and only `y@9` is part of `W2`. Any intermediate solution based on sliding windows that slide by more than one element at each time would exhibit both problems [23].

Moreover, considering all the facts in the active window as simultaneously true may result in inconsistent derivations [18]. Let us consider a surveillance system that monitors the position of visitors in a building, where sensors deliver a new notification every time a visitor enters a room. In a window of, say, five minutes, a visitor might move through multiple rooms. Thus considering all the notifications in the window as still true would lead to the erroneous conclusion that the visitor is simultaneously in multiple rooms.

Some SR systems partially address the above issues by extending the query language with operators to express temporal relations among the elements in the window [2,23]. However, they do not provide a formal framework to integrate temporal operators and reasoning capabilities, lack a concrete implementation, or do not provide the level of performance required in streaming scenarios.

3 The DOTR Model

We propose a novel DOTR SR model that avoids the issues discussed in Sect. 2 by: (1) performing ontological reasoning at each point in time independently to learn about the state of the application domain at that point in time; (2) correlating information from different point in time in a separate temporal reasoning process.

Figure 1(b) depicts a conceptual view of DOTR. The *background knowledge* contains knowledge about the application domain that holds at any point in

time. We assume background knowledge to be encoded in RDF. *Streaming* elements represent dynamic knowledge that only holds at a specific point in time: they indicate the occurrence of events of interest and are represented as time-annotated RDF graphs. At each time t, the state of the environment is represented by the *enriched dynamic knowledge*, comprising the content of any streaming element annotated with time t, any background knowledge about the application domain, and any information that can be derived through ontological reasoning (vertical axis in Fig. 1(b)), expressed using Datalog rules. Temporal reasoning (horizontal axis in Fig. 1(b)) is orthogonal to ontological reasoning and is encoded as patterns that correlate facts that are true at different points in time. In this paper we express patterns using the TESLA CEP language, which grounds on metric temporal logic [9].

The remainder of this section presents the DOTR model in details, starting with an overview of the original TESLA model, and discussing how we integrate Web structured data, background knowledge, and ontological reasoning.

The TESLA Model. In TESLA, each stream element represents an event (notification), having a type, a timestamp, and a set of attribute-value pairs. The type of an event defines the number, name, and type of attributes for that event. TESLA assumes events to occur instantaneously at some point in time and encodes their time of occurrence in the timestamp. For instance, the event `Temperature@10(val = 18.5, room = 'R')` has type `Temperature`, timestamp `10` and two attributes: `val` with float value `18.5`, and `room` with string value `R`.

TESLA models temporal reasoning through rules that define situations of interest—*composite events*—from patterns of events observed in the input stream. As an example, the following `Rule T` defines a composite event of type `Fire` from the observation of `Smoke` and high `Temperature` occurring in the same room within five minutes from each other:

```
Rule T
define    Fire(room: string)
from      Smoke() and last Temperature(val>60, room=Smoke.room) within 5 min. from Smoke
with      Fire.room = Smoke.room
consuming Temperature
```

Patterns of events start from a reference (final) event (`Smoke` in `Rule T`) and specify time ranges in which other events are allowed to occur starting from this reference event. For instance, `Rule T` requires `Temperature` to be within five minutes *before* `Smoke`. Composite events keep the same timestamp as the final event in the pattern: any `Fire` event produced by `Rule T` will have the same timestamp as the `Smoke` event that triggered its production.

Events are selected by *filtering* on their attributes. For instance, `Rule T` requires attribute `val` in `Temperature` to be greater than `60`. In addition, attributes of different events can be bound together, as in the case of attribute `room` that must be the same in `Smoke` and `Temperature` events to trigger `Rule T`. The `last` selection modifier indicates that only the last occurrence of an event of type `Temperature` with `val>60` and `room=Smoke.room` must be considered. Other selection modifiers include `first` and `each`. The latter triggers a *different* composite event for each `Temperature` event within five minutes before `Smoke`.

TESLA also supports *negations* and *aggregates*. Both of them operate on the set of events that appear in a given scope, defined with respect to the time of occurrence of other events in the rule. Negations declare events that must *not* occur in the scope, while aggregates compute an aggregation function (such as sum or average) considering the values of an attribute for all the events that occur in the scope. For example, a rule could constrain the *average* `val` of `Temperature` events observed in the 5 min before `Smoke` (aggregate) and could require a `Rain` event *not* to occur in the same scope (negation).

Finally, the `with` clause[1] of TESLA rules defines the value of attributes in the composite event, while the `consuming` clause marks events that are consumed by the rule and cannot be used in subsequent evaluations of the same rule.

The DOTR Data and Rule Model. DOTR abandons the attribute-value format of TESLA and encodes events as time-annotated RDF graphs. For instance, the following graph represents a 25 °C reading from a temperature sensor in location `:loc_1` at time 10:

```
{ :reading_1 rdf:type :Temp. :reading_1 :has_val ‘‘25’’.
  :reading_1 :is_from_sensor :sensor_1. :sensor_1 :is_at_location :loc_1. } @ 10
```

DOTR couples the temporal knowledge in input streams with atemporal background knowledge encoded as an ontology consisting of an RDF graph and a set of Datalog rules. DOTR rules take time-annotated RDF graphs in input and produce time-annotated RDF graphs as output. They model situations of interest by combining a set of SPARQL queries with TESLA temporal operators. SPARQL queries capture what is happening at each time instant, while TESLA operators combine these facts in temporal patterns. This schema is exemplified by `Rule R`, which rewrites `Rule T` from the TESLA model to the DOTR model:

```
Rule R
define    Fire  = [dotr_id1 rdf:type :Fire. dotr_id1 :at_room ?room ]
from      Smoke = [SELECT ?read1 ?room1 WHERE
           { ?read1 rdf:type :Smoke. ?read1 :is_from_sensor ?sens1.
             ?sens1 :is_in_room ?room1 }
          ] and last HighTemp = [SELECT ?read2 ?room2 WHERE
           { ?read2 rdf:type :Temp. ?read2 :is_from_sensor ?sens2.
             ?sens2 :is_in_room ?room2. ?read2 :has_val ?v.
             FILTER (?v > 60 && ?room2 = Smoke.?room1). }
          ] within 5 min from Smoke
with      Fire.?room = Smoke.?room1
consuming HighTemp
```

The `define` clause specifies the RDF graph produced as output. It may include variables, like `?room` that will be bound by the `with` clause, and unique resource identifiers like `dotr_id1` that are automatically generated every time a new output graph is produced.

The `from` clause specifies the temporal pattern that triggers the rule, using the TESLA syntax and semantics. The role of SPARQL in the `from` clause is to extract the relevant information from the enriched knowledge that holds at each

[1] We renamed the original TESLA `where` clause in `with` to avoid ambiguities with the `WHERE` clause used in SPARQL.

time `t`. In particular, at each time `t`, SPARQL queries embedded in rules (such as queries `Smoke` and `HighTemp`) get re-evaluated to extract flows of (timestamped) *facts of interest*, which TESLA operators combine in patterns.

The (optional) `consuming` clause retains the TESLA semantics and lists the events unavailable for subsequent triggering of the same rule.

As a concrete example of rule evaluation, consider again `Rule R` and assume the availability of background knowledge that associates the locations of sensors to rooms: `:locA :is_in_room :roomA. :locB :is_in_room :roomA`. An inference rule (Datalog) specifies the relation between locations and rooms: `:is_in_room(?A,?B) :- :is_at_location(?A,?C), :is_in_room(?C,?B).` Upon receiving the time-annotated RDF graph:

```
{ :r1 rdf:type :Temp. :r1 :has_val 70.
  :r1 :is_from_sensor :s1. :s1 :is_at_location :locA. } @ 10
```

DOTR combines it with the background knowledge to derive: `:s1 :is_in_room :roomA`. Thus the `HighTemp` SPARQL query produces one result with `?read2=:r1` and `?room2=roomA`. Similarly, when receiving the time-annotated RDF graph:

```
{ :r2 rdf:type :Smoke. :r2 :is_from_sensor :s2.
  :s2 :is_at_location :locB. } @ 12
```

DOTR derives: `:s2 :is_in_room :roomA.` Thus the `Smoke` SPARQL query produces one result with `?read1=:r2` and `?room1=:roomA`. These two facts, derived at time `10` and `12`, satisfy all the constraints in `Rule R`. Thus, DOTR identifies a composite event and generates a timestamped RDF graph as specified by the `define` clause: { *dotr:id1234* `rdf:type :Fire.` *dotr:id1234* `:at_room :roomA` } `@ 12`, where *dotr:id1234* is a randomly generated unique resource identifier[2].

This example shows how the DOTR rule model sharply separates the role of SPARQL and ontological reasoning from the temporal domain. This approach overcomes the limitations of window-based approaches by enriching individual events independently from their temporal relations. This separation of concerns also simplifies the processing of rules at run-time, enabling the adoption of existing and efficient tools for reasoning and event processing.

The Semantics of DOTR. DOTR inherits the semantics of temporal reasoning from TESLA, which uses the TRIO metric temporal logic to define situations of interest from patterns that predicate on the content and time of input events. The semantics of TESLA encode the occurrence of an event e as a logic predicate that is true when e occurs, and define patterns as a set of logic formulas [9].

DOTR extends TESLA by (i) extracting events from input time-annotated RDF graphs as solution mappings of SPARQL queries evaluated under ontological reasoning regime; (ii) producing output time-annotated RDF graphs. Since ontological reasoning is orthogonal with respect to temporal reasoning, the semantics of TESLA can be extended to capture DOTR by (i) associating each solution mapping produced at time t from a SPARQL query to a logic predicate that is true at time t; (ii) lifting the situations of interest identified by TESLA as RDF graphs with their associated time of occurrence.

[2] Alternatively, blank nodes can be used.

4 Implementation

We implemented DOTR in a prototype system[3] that adopts a modular approach to exploit state-of-the-art tools for ontological and temporal reasoning. This provides high performance and simplifies software evolution if better components become available. The prototype uses RDFox [21] to store, query, and reason on background knowledge, and T-Rex [10] for temporal reasoning. Figure 3 shows how the DOTR system operates.

At rule deployment time, a rule parser analyzes the input DOTR rules. For each DOTR rule R, the parser (1) extracts the SPARQL queries embedded in R and submits them to RDFox, (2) translates R into a TESLA rule T by substituting each SPARQL query with the mapping it defines, and submits T to T-Rex, (3) extracts the set of *graph definitions* from the `define` clause of R.

At runtime, upon receiving an RDF graph G with timestamp t, DOTR: (1) enriches G with the background knowledge and derives all the information that holds at t (ontological reasoning); (2) executes the SPARQL queries embedded in rules to extract the facts of interest that hold at t; (3) converts these facts into TESLA events and sends them to T-Rex (temporal reasoning); (4) uses the output generator to convert the composite events produced by T-Rex into time-annotated RDF graphs. Next, we discuss these steps in detail.

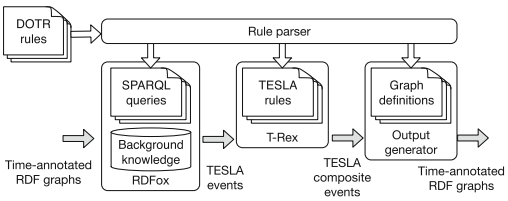
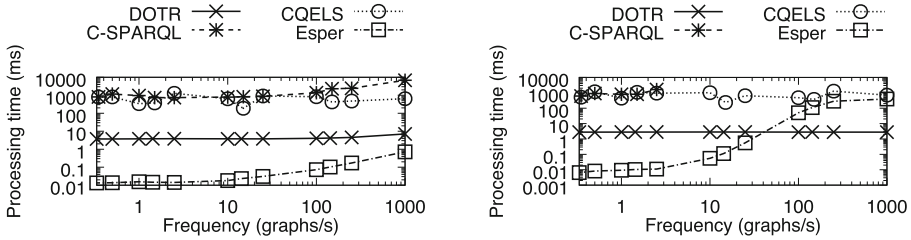


Fig. 3. Architecture of the DOTR system.

Knowledge Inference. The background knowledge is preloaded into RDFox at system initialization time. When a graph G with timestamp t is received, DOTR computes the whole knowledge that holds at t by: (i) removing from RDFox any information Δ^- coming from every graph G' annotated with time $t' < t$; (ii) adding to RDFox all the RDF triples Δ^+ that are in G; (iii) performing ontological reasoning with the available inference rules to remove old information that could be derived *only* in the presence of Δ^- and to add new information that can be derived from Δ^+. This is done in a single inference process by exploiting the incremental materialization of RDFox [20]. After this reasoning step, DOTR determines the facts of interest that are valid at the current time by submitting to RDFox the SPARQL queries extracted from the deployed rules. For each query RDFox generates zero, one, or more facts of interests.

Event Processing. Each fact of interest f gives rise to a new TESLA event. The event type is the label of the query that extracted f and the event attributes are the variables selected by the query. For instance, the example reported at the end of Sect. 3 gives rise to two events: `Smoke@10(read1=':r1',room1=':roomA')` and `HighTemp@12(read2=':r2',room2='roomA')`. T-Rex processes these events trying

[3] DOTR is available at https://github.com/margara/DOTR.



(a) Query Q1 (b) Query Q2

Fig. 4. Performance of DOTR, C-SPARQL, CQELS, and Esper in the CityBench suite.

to detect the temporal patterns expressed within the deployed rules. When a rule is triggered, T-Rex generates one or more composite TESLA events.

Output Translation. The generated events have the type specified in the `define` clause of the firing rule and one attribute for each variable that appears there. For instance, consider again `Rule R` in Sect. 3. Each composite event produced by T-Rex has type `Fire` and a single attribute `room`, with the same value as the attribute `room1` in the `Smoke` event that triggered the rule. DOTR converts composite events into one or more RDF triples following the graph definition specified in the `define` clause of the fired rules. In our example, composite event `Fire@12(room='roomA')` would be converted to: { *dotr:id1234* `rdf:type :Fire.` *dotr:id1234* `:at_room roomA.`} `@ 12`.

5 Evaluation

In this section we measure the performance of DOTR focusing on latency, which is a key requirement for SR, and compare it with state-of-the-art SP, CEP, and SR tools using real data from the CityBench benchmark [1]. We also discuss the differences among these systems in terms of expressiveness and semantics. Furthermore, we use synthetic workloads to study which parameters affect the behavior of DOTR the most.

We perform all the experiments on a Intel Core i7 4850HQ machine with 16 GB of DDR3 RAM, running macOS 10.13.0. We use the processing time per input element as a performance indicator. All the systems under analysis process individual elements from the input stream sequentially and do not overlap the computation of different elements. As a consequence, the inverse of the processing time also represents a good estimate of the maximum rate of input elements that each system supports, that is, its maximum input throughput. In each experiment we submit 30k time-annotated graphs and we compute the average processing time per input element, which includes the time to update the content of the RDFox store, perform reasoning, querying, event processing, and produce output graphs. The input graphs are stored in files on a RAM disk with RDF turtle format. The background knowledge is also stored in RAM by RDFox.

Benchmark. We compare DOTR with the C-SPARQL [5] and CQELS [17] SR systems and with the Esper SP/CEP system, Java version 6.1.0. We rely on the CityBench suite [1], which includes sensor data from the city of Aarhus. We use the vehicle traffic dataset, containing the congestion level between two points over a duration of time, and the weather dataset, with observations on temperature, humidity, and wind. Given the similarities between the benchmark queries, we present the results for queries Q1 and Q2 only. In the case of C-SPARQL and CQELS, we rely on the implementation of the queries provided with the CityBench suite. In the case of Esper, we implement a translator that converts the input RDF elements into Esper events (Java objects). The conversion takes place before the experiments and does not contribute to the time we measure.

We report queries Q1 and Q2 as presented in the original paper on CityBench: (Q1) What is the traffic congestion level on each road of my planned journey? (Q2) What is the traffic congestion level and weather condition on each road of my planned journey? In practice, the implementation of Q1 considers two streams, each containing the congestion of a specific road, and returns all possible pairs of congestion level values, one per stream, that occur in a window of 3 s. The implementation of Q2 considers a stream of congestion readings and a stream of weather readings, and reports congestion level, temperature, humidity, and wind speed notifications that occur in a window of 3 s.

Since DOTR does not distinguish between input elements coming from different streams, we combine RDF triples from all the relevant streams and generate a time-annotated graph for each time instant. Each resulting graph contains 10 triples to represent congestion readings and 5 triples to represent weather readings. In Esper, we encode each sensor reading as a separate event object. In the case of DOTR, C-SPARQL, and CQELS we consider a background knowledge of over 100k triples with the type of property measured by each sensor: congestion or weather reading. In Esper, we encode the property type as an event attribute, thus avoiding the use of a separate background knowledge.

Different engines present different execution models that lead to different results [1,12]. CQELS, Esper, and DOTR operate in a pure reactive way, by producing new results each time they receive an input element. C-SPARQL only supports time-based windows, which produce results periodically, when the window closes. Moreover, C-SPARQL and CQELS rely on the windowing model presented in Sect. 2 and compute *all* the query results that derive from the data in the active window, whereas Esper and DOTR produce only the *new* results at each evaluation, thus automatically removing duplicates.

We implement query Q1 using the `each within` operator that instructs DOTR to always consider *all* the congestion level information contained in the window. This mimics the behavior of C-SPARQL and CQELS, although it avoids producing duplicate results as discussed above. In Esper we select all congestion level events by exploiting the `every` operator offered by the engine. For query Q2, we use the `last within` operator that reports, for each incoming element, only the *last* available information about congestion and weather. This better satisfies the semantics of the query by notifying users about the most recent traffic and weather information when some change occurs. Esper, C-SPARQL, and CQELS

do not provide operators to select only the most recent information. This well exemplifies the flexibility of the temporal reasoning offered by DOTR. As we demonstrate later, this flexibility comes with the benefit of a reduced processing time, since temporal reasoning only needs to analyze the latest information.

The processing time of a query depends on the number of events it considers, which depends on the timestamp of events. Accordingly, we measure processing time for different frequencies of event arrival by artificially manipulating the timestamp of input graphs. This is the same approach followed by CityBench.

As Fig. 4 shows, for both query Q1 and Q2 DOTR takes than 4 ms to process each input graph when the frequency of arrival remains below 100 graphs per second. When further increasing the input rate, the latency for Q1 grows up to 7 ms, since the number of congestion elements to consider increases. Instead, the latency of Q2 remains stable below 3 ms, since Q2 only considers the last available notifications.

The latency of C-SPARQL and CQELS is in the order of hundreds or even thousands of milliseconds in the same scenarios, and C-SPARQL often crashes without providing results with a high input rate. The order of magnitude and the trends of these results are in line with those measured in previous studies [1]. The results of C-SPARQL and CQELS are motivated by the nature of the problem, which grows quadratically with the number of elements in each window. C-SPARQL suffers more when the number of elements increases since it recomputes all the results when the window changes, while CQELS indexes and re-uses results from previous window evaluations.

Esper is the winner in Q1 tests, with a processing time always below 0.8 ms. However, Esper does not to use background knowledge, since we encoded all the information within the input events. Also, input events are pre-encoded as plain Java objects in memory, which removes the times of (de)serialization and parsing from our measures. In practice, the time we measure for Esper represents the pure cost of event processing (or temporal reasoning). At high input rates, this processing time is about one tenth of the average processing time of DOTR, in line with the analysis we present in the remainder of this section. CQELS and C-SPARQL exhibit higher processing times for query Q2, since Q2 extracts more input elements and produces more results than Q1. The advantages of the DOTR model become evident: by considering only the last available congestion and weather data, DOTR reduces the amount of elements to process and the amount of results produced. Esper also suffers query Q2, due to the large number of results to produce at each query evaluation. When the input rate grows its processing time grows well above that of DOTR, reaching the level of CQELS.

We may summarize the considerations above by concluding that: (1) under comparable conditions (query Q1), DOTR outperforms C-SPARQL and CQELS by almost two orders of magnitude; (2) thanks to its expressive temporal reasoning, DOTR can better select the results to produce, providing the best performance in query Q2; (3) the use of RDF format, background knowledge, and ontological reasoning increase the expressiveness but come at a cost with respect to traditional event processing, as exemplified by the comparison with Esper.

Sensitivity to Parameters. We study the sensitivity to workload characteristics through synthetic benchmarking, starting from a default scenario and changing one parameter at a time.

Default Scenario. Our default scenario considers ten rules, all having the structure below and differing only for the value *X* used in the `FILTER` clause.

```
define  CE  = [dotr_id1 :has_val ?num.]
from    A   = [SELECT ?sensorA ?valA ?roomA WHERE
               { ?sensorA :read_valA ?valA. ?sensorA :is_in_room ?roomA.
               FILTER(?valA>X) } ] and last
        B   = [SELECT ?sensorB ?valB ?roomB WHERE
               { ?sensorB :read_valB ?valB. ?sensorB :is_in_room ?roomB.
               FILTER(?valB>X && ?roomB=A.?roomA) } ] within 30s from A
with    CE.?num = A.?valA
```

The input time-annotated graphs contain two types of RDF triples: `:sensor`*K* `:read_valA` *val* and `:sensor`*K* `:read_valB` *val*, where *K* is uniformly distributed in `1..10` and *val* is a random integer that always satisfies SPARQL filters in rules. Triples of the first type can trigger rules by completing a valid sequence pattern and occur with 20% probability. The timestamps of any two consecutive input graphs differ by one second, simulating an input rate of one graph per second. The background knowledge includes information on the position of sensors, necessary to trigger the rules, in the form: `:sensor`*S* `:equips :object`*O*`. :object`*O*`:placed_in :room`*X*`. :room`*R* `:number` *R*, where the number of rooms `R` is uniformly distributed in `1...5` and there are two objects per room. Datalog rule `:is_in_room(?S, ?R) :- :equips(?S, ?O), :placed_in(?O, ?R).` determines the room each sensor is in. We increase the size and complexity of the background knowledge by adding 10 rules similar to the one above, which contribute in producing triples (albeit not relevant for existing rules). We also add 10k triples to the background knowledge, with additional attributes for each sensor.

In our default scenario each input graph contains a single RDF triple. Table 1 shows the average processing time per input element, together with the absolute and relative cost of each processing phase. DOTR produces results with an average latency of 12.09 ms, spending almost 80% of the time in the processing steps concerned with ontological reasoning. Event processing (temporal reasoning) accounts for about 16% of the overall processing time. Updating the knowledge base by adding new information and removing old one accounts for only 1.65% of the total time. Translating input data from RDF to TESLA native events and back takes about 3% of the total time.

Table 1. Default scenario: processing time.

	Time (ms)	Time (%)
Knowledge update	0.20 ms	1.65%
Ontological reasoning	1.96 ms	16.21%
Query evaluation	7.56 ms	62.53%
Input translation	0.19 ms	1.57%
Event processing	2.01 ms	16.63%
Output translation	0.17 ms	1.41%
Total	**12.09 ms**	**100.00%**

Cost of Reasoning. We study the cost of reasoning by increasing the number of Datalog rules in the knowledge store (Fig. 5). The cost of reasoning (and materializing information) is initially small, and the processing time remains almost unchanged from 1 to 1k Datalog rules. After this threshold, the

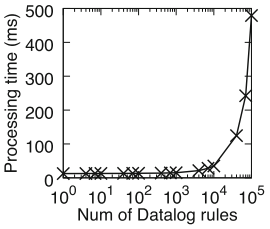


Fig. 5. Datalog rules

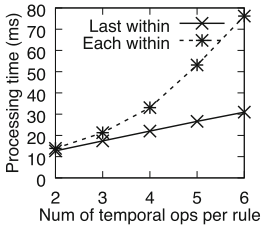


Fig. 6. Temporal ops.

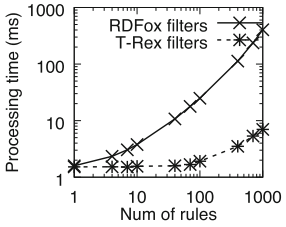


Fig. 7. Rules

impact of reasoning becomes more prominent, and the processing time increases by a factor of 30 from 1k to 100k Datalog rules. Additional experiments, not reported here for space reasons, measure the same level of performance when changing the number of RDF triples in the background store from 1k to 1M.

In general, the cost of reasoning depends not only on the number of rules, but also on their complexity. Given the breadth of the topic and the availability of specialized articles, we do not present a detailed analysis of reasoning. The interested reader can refer to the materialization algorithm of RDFox [20].

Cost of Event Processing. We study the cost of temporal reasoning by adding temporal operators to rules, thus increasing the length of the event patterns to detect (Fig. 6). We consider both patterns with `last within` and with `each within` operators. In the first case, the triggering of a rule generates a single TESLA event, while in the second case it generates as many TESLA events as the number of event bindings that satisfy the pattern. With two to three operators per rule, the two cases are comparable, since the processing time is dominated by reasoning and SPARQL querying. With longer sequences, the `each within` rules cost more, in line with the original results of T-Rex.

Number of Rules. Figure 7 (continuous line) shows the processing time when increasing the number of deployed rules. More rules means more queries to the RDF store, and more results to produce. The processing time remains below 400 ms even in the case of 1k rules, which involves evaluating 2k SPARQL queries for each input graph (one for each of the two events appearing in each rule). To further optimize DOTR, we move from the observation that query processing represents a large fraction of the entire processing time, and we modify DOTR to move data filtering from SPARQL (RDFox) to T-Rex. In this way, SPARQL queries that differ only for their `FILTER` part can be merged together, removing and delegating the filtering step to T-Rex. Our workload represents an optimal setting for the modified system, since all the DOTR rules extract the same two types of events applying different filters (the value `X` used in the `FILTER` clause). The dashed line in Fig. 7 shows the processing when we enable this optimization. Remarkably, it remains almost constant and below 2 ms with up to 100 rules, and increases to 7 ms only when reaching 1k rules.

6 Related Work

The last decade has seen an increasing interest in techniques and tools to process streaming data. SP systems define abstractions to continuously query streams of (typically relational) data [3], while CEP systems aim to detect situations of interest from streams of low-level event [11,14].

SP and CEP systems are not suited to process Web structured data, to integrate background knowledge, and to perform ontological reasoning [2]. Stream Reasoning (SR) systems address these limitations by adapting stream processing to the RDF data model [12,19].

As discussed in Sect. 2, most SR systems follow the CQL approach and extend SPARQL with windows to support continuous queries over streaming data. C-SPARQL [5], CQELS [17], $\text{SPARQL}_{\text{stream}}$ [8], and Laser [6] all follow this direction. DL-Lite$_{\mathcal{A}}$ (S, F) [13] extends DL-Lite$_{\mathcal{A}}$ to perform spatial and window-based temporal reasoning over streams. Some proposals investigate the use of Answer Set Programming (ASP) for SR. LARS [7] features model-based semantics by building on ASP enriched with window operators. StreamRule [22] combines a window-based engine, such as CQELS, with an ASP reasoner. INSTANS [23] avoids windows and demonstrates that standard SPARQL queries can express several common forms of ontological and temporal reasoning. Although conceptually interesting, the approach lacks high-level abstractions to express the reasoning tasks. Moreover, it currently does not provide a level of performance comparable with state-of-the-art SP/CEP systems. Perhaps the approach most closely related to ours is EP-SPARQL [2], which extends SPARQL with temporal operators derived from the CEP domain. However, it does not investigate in depth the relation between detection of temporal patterns and semantic inference, as we do in our model.

Some approaches extend RDF with a temporal dimension [15,24]. They differ from DOTR since they do not target on the fly detection of temporal patterns, and they integrate rather than decoupling ontological and temporal reasoning.

Finally, DOTR exploits the recent advances in incremental reasoning: they include the exploitation of time annotations [4], counting algorithms and parallel processing [25], and optimizations for small incremental changes [16].

7 Conclusions

This paper presented a novel model for SR that decouples ontological and temporal reasoning. It grounds ontological reasoning on RDF graphs and Datalog rules, and temporal reasoning on a CEP language that provides an expressive yet computationally efficient subset of a metric temporal logic. We implemented the model by building on state-of-the-art tools for event processing and knowledge storage, query, and inference. The resulting system outperforms existing SR systems, showing that the added expressiveness in terms of temporal reasoning can be beneficial for processing, too. As future work, we will study the usability and expressiveness of the model in greater details, also considering primitives to alter the content of the background knowledge over time.

References

1. Ali, M.I., Gao, F., Mileo, A.: CityBench: a configurable benchmark to evaluate RSP engines using smart city datasets. In: Arenas, M., et al. (eds.) ISWC 2015. LNCS, vol. 9367, pp. 374–389. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25010-6_25
2. Anicic, D., Fodor, P., Rudolph, S., Stojanovic, N.: EP-SPARQL: a unified language for event processing and stream reasoning. In: International Conference on World Wide Web, WWW (2011)
3. Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Models and issues in data stream systems. In: Symposium on Principles of Database Systems, PODS (2002)
4. Barbieri, D.F., Braga, D., Ceri, S., Della Valle, E., Grossniklaus, M.: Incremental reasoning on streams and rich background knowledge. In: Aroyo, L., Antoniou, G., Hyvönen, E., ten Teije, A., Stuckenschmidt, H., Cabral, L., Tudorache, T. (eds.) ESWC 2010. LNCS, vol. 6088, pp. 1–15. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13486-9_1
5. Barbieri, D.F., Braga, D., Ceri, S., Valle, E.D., Grossniklaus, M.: Querying RDF streams with C-SPARQL. SIGMOD Rec. **39**(1), 20–26 (2010)
6. Bazoobandi, H.R., Beck, H., Urbani, J.: Expressive stream reasoning with laser. In: d'Amato, C., Fernandez, M., Tamma, V., Lecue, F., Cudré-Mauroux, P., Sequeda, J., Lange, C., Heflin, J. (eds.) ISWC 2017. LNCS, vol. 10587, pp. 87–103. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68288-4_6
7. Beck, H., Dao-Tran, M., Eiter, T., Fink, M.: LARS: a logic-based framework for analyzing reasoning over streams. In: Conference on AI, AAAI (2015)
8. Calbimonte, J.P., Jeung, H., Corcho, O., Aberer, K.: Enabling query technologies for the semantic sensor web. Int. J. Sem. Web Inf. Sys. **8**(1), 43–63 (2012)
9. Cugola, G., Margara, A.: TESLA: a formally defined event specification language. In: International Conference on Distributed Event-Based Systems, DEBS (2010)
10. Cugola, G., Margara, A.: Complex event processing with T-REX. J. Sys. Softw. **85**(8), 1709–1728 (2012)
11. Cugola, G., Margara, A.: Processing flows of information: from data stream to complex event processing. ACM Comput. Surv. **44**(3), 15 (2012)
12. Dell'Aglio, D., Della Valle, E., van Harmelen, F., Bernstein, A.: Stream reasoning: a survey and outlook. Data Sci. **1**, 59–83 (2017)
13. Eiter, T., Parreira, J.X., Schneider, P.: Spatial ontology-mediated query answering over mobility streams. In: Blomqvist, E., Maynard, D., Gangemi, A., Hoekstra, R., Hitzler, P., Hartig, O. (eds.) ESWC 2017. LNCS, vol. 10249, pp. 219–237. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-58068-5_14
14. Etzion, O., Niblett, P.: Event Processing in Action. Manning, Shelter Island (2010)
15. Gutierrez, C., Hurtado, C.A., Vaisman, A.: Introducing time into RDF. Trans. Knowl. Data Eng. **19**(2) (2007)
16. Kazakov, Y., Klinov, P.: Incremental reasoning in OWL EL without bookkeeping. In: Alani, H., et al. (eds.) ISWC 2013. LNCS, vol. 8218, pp. 232–247. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-41335-3_15
17. 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. In: Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N., Blomqvist, E. (eds.) ISWC 2011. LNCS, vol. 7031, pp. 370–388. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25073-6_24

18. Margara, A., Dell'Aglio, D., Bernstein, A.: Break the windows: explicit state management for stream processing systems. In: International Conference on Extending Database Technology, EDBT (2017)
19. Margara, A., Urbani, J., van Harmelen, F., Bal, H.: Streaming the web: reasoning over dynamic data. Web Semant. **25**(1), 24–44 (2014)
20. Motik, B., Nenov, Y., Piro, R., Horrocks, I.: Combining rewriting and incremental materialisation maintenance for datalog programs with equality. In: Conference on AI, IJCAI (2015)
21. Nenov, Y., Piro, R., Motik, B., Horrocks, I., Wu, Z., Banerjee, J.: RDFox: a highly-scalable RDF store. In: Arenas, M., et al. (eds.) ISWC 2015. LNCS, vol. 9367, pp. 3–20. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25010-6_1
22. Pham, T.L., Mileo, A., Ali, M.I.: Towards scalable non-monotonic stream reasoning via input dependency analysis. In: International Conference on Data Engineering, ICDE (2017)
23. Rinne, M., Nuutila, E.: Constructing event processing systems of layered and heterogeneous events with SPARQL. J. Data Semant. **6**(2), 57–69 (2017)
24. Tappolet, J., Bernstein, A.: Applied temporal RDF: efficient temporal querying of RDF data with SPARQL. In: Aroyo, L., et al. (eds.) ESWC 2009. LNCS, vol. 5554, pp. 308–322. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02121-3_25
25. Urbani, J., Margara, A., Jacobs, C., van Harmelen, F., Bal, H.: DynamiTE: parallel materialization of dynamic RDF data. In: Alani, H., et al. (eds.) ISWC 2013. LNCS, vol. 8218, pp. 657–672. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-41335-3_41