



**University of
Zurich**^{UZH}

**Zurich Open Repository and
Archive**

University of Zurich
Main Library
Strickhofstrasse 39
CH-8057 Zurich
www.zora.uzh.ch

Year: 2018

Efficient Temporal Reasoning on Streams of Events with DOTR

Margara, Alessandro ; Cugola, Gianpaolo ; Collavini, Dario ; Dell' Aglio, Daniele

DOI: https://doi.org/10.1007/978-3-319-93417-4_25

Posted at the Zurich Open Repository and Archive, University of Zurich

ZORA URL: <https://doi.org/10.5167/uzh-152207>

Conference or Workshop Item

Accepted Version

Originally published at:

Margara, Alessandro; Cugola, Gianpaolo; Collavini, Dario; Dell' Aglio, Daniele (2018). Efficient Temporal Reasoning on Streams of Events with DOTR. In: The Semantic Web - 15th International Conference, ESWC 2018, Heraklion, Crete, Greece, June 3-7, 2018, Proceedings, Heraklion, Crete, Greece, 3 June 2018 - 7 June 2018, 384-399.

DOI: https://doi.org/10.1007/978-3-319-93417-4_25

Efficient Temporal Reasoning on Streams of Events with DOTR

Alessandro Margara¹, Gianpaolo Cugola¹,
Dario Collavini¹, and Daniele Dell’Aglio²

¹ DEIB, Politecnico di Milano

[alessandro.margara|gianpaolo.cugola]@polimi.it

² IFI, University of Zurich

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. The Stream Reasoning (SR) domain aims to combine the performance of stream/event processing and the reasoning expressiveness of knowledge representation systems by adopting Semantic Web standards to represent streaming elements. In this paper, 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 information losses and inconsistencies. We implement the model in the DOTR system that defines ontological reasoning using Datalog rules and temporal reasoning using the TESLA Complex Event Processing language, which builds on metric temporal logic. We demonstrate the expressiveness of our model through various 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

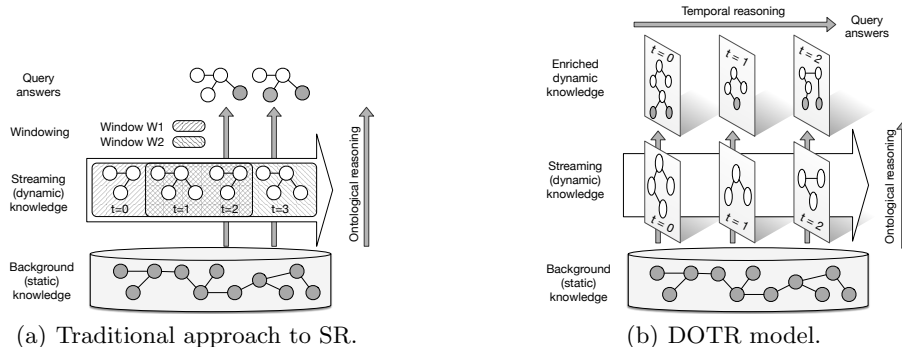


Fig. 1. Comparison of the traditional model to SR and the proposed DOTR model.

use a recent portion of the streaming data to update the results of queries as new data becomes available. They mostly provide atemporal (relational) operations on structured (relational) data. In contrast to this, Complex Event Processing (CEP) systems [13] 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 [18, 12] 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, 16].

We claim that this model is not flexible enough to satisfy all the requirements identified above. On the one hand, 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 missing some information or in deriving inconsistent results during the inference process (see Sec. 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 combines ontological reasoning on streaming and background knowledge with efficient detection of temporal patterns (temporal reasoning), while keeping the two form of reasoning sharply decoupled. In DOTR (Fig. 1(b)), elements of the incoming data stream represents events that occur at some point in time, encoded as time-annotated

RDF graphs. Ontological reasoning takes place at each point in time separately. It combines the events happening at that time with background knowledge about the application domain to derive new knowledge about what is happening. In a separate step, temporal reasoning is applied. It considers enriched events —as determined in the integration and ontological reasoning step above— 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. Sec. 2 presents background information on SR and motivates our work. Sec. 3 and Sec. 4 present our model and its implementation, while Sec. 5 evaluates its performance. Sec. 6 discusses related work and Sec. 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 [18, 12]. 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, 16, 8, 6]. Window operators create finite views over a stream, namely *windows*, that include the portion of data relevant for the current evaluation of rules. They produce, at each evaluation, either the complete results of the processing, or the differences —additions and/or 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, (ii) loss

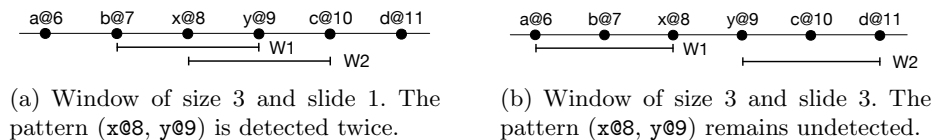


Fig. 2. Processing model based on windows: examples of limitations.

of information, or (iii) inconsistency during the reasoning process. Fig. 2 exemplifies the first two problems. Let us denote stream elements with a label and their time of occurrence: we represent an element with label l occurring at t as $l@t$. Imagine that we want to detect whenever an element labeled x is followed by an element labeled y after no more than three time units. Fig. 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 pattern defined by $x@8$ and $y@9$ cannot be 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 [21].

Moreover, considering all the facts in the active window as simultaneously true may result in logical inconsistencies during query evaluation [17]. Consider for instance 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 explicit time operators that capture the temporal relationships among the elements into the window [2, 21]. However, they do not provide a formal framework to seamlessly 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 (Decoupled Ontological and Temporal Reasoning) model for SR that avoids the inconsistencies illustrated in Sec. 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. Fig. 1(b) depicts a conceptual view of this model. 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. We assume streaming elements to be represented as time-annotated RDF graphs with information about the occurrence of events of interest at that point in time. 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 the ontological reasoning that takes place at each point in time. It 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). Each event has 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 in TESLA 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 (the *anchor point*). For instance, `Rule T` requires `Temperature` to be within five minutes *before* `Smoke`. Composite events keep the same timestamp as the anchor point, such that any `Fire` event produced by `Rule T` will be given the timestamp of 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`. Although not represented in `Rule T`, TESLA also supports *negations*, which predicate on events that must *not* occur within a given temporal scope, and *aggregates*, which are values computed from multiple events. For instance, a TESLA rule can predicate over the average temperature measured in a given time span.

The `with` clause³ 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. The DOTR data model abandons the attribute-value format used by TESLA to encode events as time-annotated RDF graphs. For instance, the following graph represents a 25°C reading, coming at time 10, from a temperature sensor at location `:loc_1`:

```
{ :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 this temporal knowledge, coming from input streams, with temporal background knowledge, represented as an ontology. It includes facts that always hold, expressed as a (possibly very big) RDF graph, together with 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 knowledge⁴ that holds at each

³ We renamed the original TESLA `where` clause in `with` to avoid ambiguities with the `WHERE` clause used in SPARQL.

⁴ As mentioned at the beginning of section, the enriched knowledge that holds at `t` includes: (i) any event received as a time-annotated graph with timestamp `t`, (ii) the

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 identifier⁵.

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. At the same time, the separation of concerns simplifies the processing of rules at run-time, enabling the adoption of existing and efficient tools for reasoning and event processing.

4 Implementation

This section discusses the implementation of DOTR in a prototype system⁶ 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 [20] to store and query background knowledge and to perform ontological inference, and T-Rex [10] for temporal reasoning. Fig. 3 shows how the DOTR system operates. At rule deployment time, a rule parser analyzes the input DOTR rules and

background knowledge, and (iii) any knowledge that can be inferred from the first two sources through reasoning.

⁵ Alternatively, blank nodes can be used

⁶ DOTR is available at <https://github.com/margara/DOTR>.

extracts standard SPARQL queries, TESLA rules, and a set of *graph definitions*. 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.

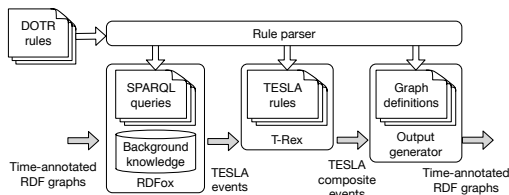


Fig. 3. Architecture of the DOTR system.

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 [19]. 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 whose type comes from the label of the query that extracted f and having one attribute for each variable selected by the query. For instance, the example reported at the end of Sec. 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 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 by the `define` clause of the firing rule and one attribute for each variable that appears there. For instance, consider again Rule R in Sec. 3. Each composite event produced by T-Rex has type `Fire` and a single attribute `room`, whose value comes from the value of attribute `room1` of the `Smoke` event that triggered the rule. DOTR converts those 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.`

Knowledge inference. The background knowledge is pre-loaded 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

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 [16] 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 seconds. 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 seconds. 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

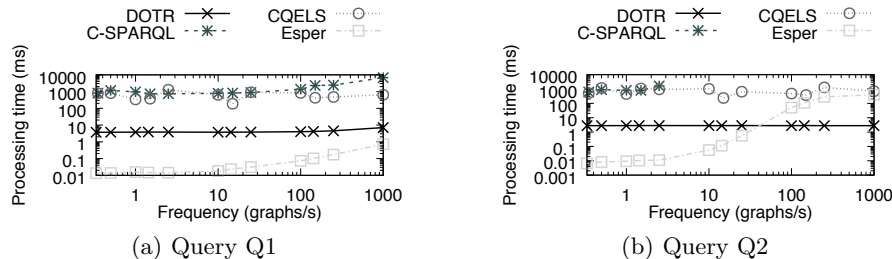


Fig. 4. Performance of DOTR, C-SPARQL, CQELS, and Esper in the CityBench suite.

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 Sec. 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 for each query depends on the number of events it considers, which in turn depends on the timestamp of events. Accordingly, we measure processing time for different frequencies of event arrival by artificially manipulating the timestamp of the input graphs. This is the same approach followed by CityBench. As Fig. 4 shows, for both query Q1 and Q2 DOTR takes than 4ms 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 7ms, since the number of congestion elements to consider increases. Instead, the latency of Q2 remains stable below 3ms, 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 0.8ms. However, Esper does not 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 amount of results that need to be produced 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: `:sensorS :equips :objectO. :objectO :placed_in :roomX. :roomR :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.

	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 %

Table 1. Default scenario: processing time.

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.

Size of the knowledge base. Fig. 5 shows how the performance of DOTR changes with the number of RDF triples in the background knowledge (new triples do *not* change the number of results). DOTR efficiently scales, preserving the same level of performance when moving from 1k to 1M RDF triples.

Cost of reasoning. We study the cost of reasoning by increasing the number of Datalog rules in the knowledge store (Fig. 6). 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 impact of reasoning becomes more prominent, and the processing time increases by a factor of 30 from 1k to 100k Datalog rules. 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 [19].

Cost of event processing. To study the cost of temporal reasoning, we change rules by adding additional temporal operators, thus increasing the length of the event patterns to detect (Fig. 7). We consider both the case of patterns using `last within` operators and the case of patterns using `each within` operators. In the former case, each triggering of a rule generates a single TESLA event,

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 12.09ms, spending almost 80% of the time in the processing

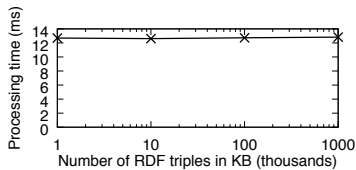


Fig. 5. Size of the knowledge base.

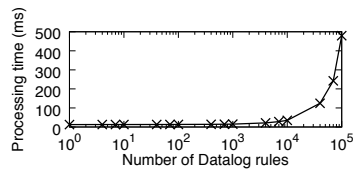


Fig. 6. Number of Datalog rules.

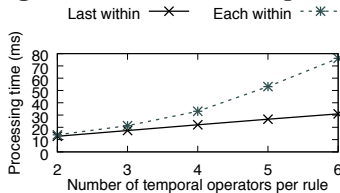


Fig. 7. Number of temporal op. per rule.

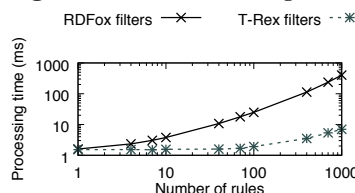


Fig. 8. Number of rules.

while in the latter case, each triggering 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 significantly, in line with the original results of T-Rex.

Number of rules. Fig. 8 shows the performance of DOTR when increasing the number of deployed rules. Adding new rules means performing more queries to the RDF store and also triggering more TESLA rules that generate more results. The continuous line in Fig. 8 shows how the processing time increases with the number of rules. The processing time remains below 400ms even in the extreme case of 1k rules, which involves evaluating 2k SPARQL queries (one for each of the two events appearing in each rule) for each input graph. To further improve these numbers, we move from the observation that query processing represents a large fraction of the entire processing time of DOTR, and we modify DOTR to move the data filtering process from SPARQL (RDFox) to T-Rex. In this way, SPARQL queries that differ only for their `FILTER` part can be merged together, removing the filtering step, which is delegated to T-Rex. Our workload represents an optimal setting for the modified system, since all the DOTR rules we consider extract the same two types of events applying different filters (the value X used in the `FILTER` clause). The dashed line in Fig. 8 shows the performance of DOTR when we enable this optimization. Remarkably, the processing time remains almost constant and below 2ms with up to 100 rules, and increases to 7ms 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 [13]. The interested reader can refer to a survey paper that compares these systems [11]. Neither SP nor CEP systems are 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 [18, 12]. As discussed in Sec. 2, most SR systems follow the CQL approach and extend SPARQL with windows to support continuous queries over streaming data. C-SPARQL [5], CQELS [16], SPARQL_{stream} [8], and Laser [6] all follow this direction. INSTANS [21] 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. LARS [7] is SR framework that, differently from other systems that present operational semantics, features model-based semantics by building on Answer Set Programming (ASP) enriched with window operators. 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. Related to our work are also the approaches that extend RDF with a temporal dimension [14, 22]. They differ from our proposal since they do not target the detection of temporal patterns on the fly, and they integrate rather than decoupling ontological and temporal reasoning. Finally, our work exploits the recent advances in the domain of incremental reasoning to enable expressive reasoning on streaming data. Proposals in the area include the exploitation of time annotations [4], counting algorithms and parallel processing [23], and optimizations for small incremental changes [15].

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 syntax and usability of our language and the 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: Proc. of the Int. Sem. Web Conf.

- ISWC (2015)
2. Anicic, D., Fodor, P., Rudolph, S., Stojanovic, N.: EP-SPARQL: A Unified Language for Event Processing and Stream Reasoning. In: Proc. of the Int. Conf. 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: Proc. of the Symp. on Principles of Database Sys. PODS (2002)
 4. Barbieri, D.F., Braga, D., Ceri, S., Della Valle, E., Grossniklaus, M.: Incremental Reasoning on Streams and Rich Background Knowledge. In: Proc. of the Int. Conf. on The Sem. Web. ESWC (2010)
 5. Barbieri, D.F., Braga, D., Ceri, S., Valle, E.D., Grossniklaus, M.: Querying RDF Streams with C-SPARQL. SIGMOD Records 39(1) (2010)
 6. Bazoobandi, H.R., Beck, H., Urbani, J.: Expressive Stream Reasoning with Laser. In: Proc. of the Int. Conf. on the Sem. Web. ISWC (2017)
 7. Beck, H., Dao-Tran, M., Eiter, T., Fink, M.: LARS: A Logic-based Framework for Analyzing Reasoning over Streams. In: Proc. of the Conf. on AI. AAAI (2015)
 8. Calbimonte, J.P., Jeung, H., Corcho, O., Aberer, K.: Enabling Query Technologies for the Semantic Sensor Web. Int. J. on Sem. Web and Inf. Sys. 8(1) (2012)
 9. Cugola, G., Margara, A.: TESLA: A Formally Defined Event Specification Language. In: Proc. of the Int. Conf. on Distrib. Event-Based Sys. DEBS (2010)
 10. Cugola, G., Margara, A.: Complex Event Processing with T-REX. J. of Sys. and Software 85(8) (2012)
 11. Cugola, G., Margara, A.: Processing Flows of Information: From Data Stream to Complex Event Processing. ACM Comput. Surveys 44(3) (2012)
 12. Dell’Aglia, D., Della Valle, E., van Harmelen, F., Bernstein, A.: Stream Reasoning: a Survey and Outlook. Data Science (2017)
 13. Etzion, O., Niblett, P.: Event Processing in Action. Manning (2010)
 14. Gutierrez, C., Hurtado, C.A., Vaisman, A.: Introducing Time into RDF. Trans. on Knowledge and Data Engineering 19(2) (2007)
 15. Kazakov, Y., Klinov, P.: Incremental Reasoning in OWL EL Without Bookkeeping. In: Proc. of the Int. Sem. Web Conf. ISWC (2013)
 16. Le-Phuoc, D., Dao-Tran, M., Parreira, J.X., Hauswirth, M.: A Native and Adaptive Approach for Unified Processing of Linked Streams and Linked Data. In: Proc. of the Int. Conf. on the Sem. Web. ISWC (2011)
 17. Margara, A., Dell’Aglia, D., Bernstein, A.: Break the Windows: Explicit State Management for Stream Processing Systems. In: Proc. of the Int. Conf. on Extending Database Technology. EDBT (2017)
 18. Margara, A., Urbani, J., van Harmelen, F., Bal, H.: Streaming the Web: Reasoning over Dynamic Data. Web Semantics 25(1) (2014)
 19. Motik, B., Nenov, Y., Piro, R., Horrocks, I.: Combining Rewriting and Incremental Materialisation Maintenance for Datalog Programs with Equality. In: Proc. of the Conf. on AI. IJCAI (2015)
 20. Nenov, Y., Piro, R., Motik, B., Horrocks, I., Wu, Z., Banerjee, J.: RDFox: A Highly-Scalable RDF Store. In: Proc. of the Int. Conf. on the Sem. Web. ISWC (2015)
 21. Rinne, M., Nuutila, E.: Constructing Event Processing Systems of Layered and Heterogeneous Events with SPARQL. J. on Data Semantics 6(2) (2017)
 22. Tappolet, J., Bernstein, A.: Applied Temporal RDF: Efficient Temporal Querying of RDF Data with SPARQL. In: Proc. of the Europ. Sem. Web Conf. ESWC (2009)
 23. Urbani, J., Margara, A., Jacobs, C., Van Harmelen, F., Bal, H.: DynamiTE: Parallel Materialization of Dynamic RDF Data. In: Proc. of the Int. Sem. Web Conf. ISWC (2013)