Query Transformation for Processing Streams in Decision-making Agents

Simon Schiff[*], **Mena Leemhuis**[†], **Özgür L. Özçep**[†], **Ralf Möller**[*]
Institute of Information Systems (IFIS)
University of Lübeck
Lübeck, Germany
{schiff,leemhuis,oezcep,moeller}@ifis.uni-luebeck.de

Abstract

An agent in pursuit of a task repeatedly perceives its environment through sensors, updates its state based on observations, and then decides which action to take, given the current state of the environment. Observations have in common that they are made at a given time point and thus referred to as temporal data. Usually, such temporal data is provided as stream data if the agent continuously receives the data, or it is provided as historic data if the stream data is stored in, for instance, a database the agent has access to. DBMSs are especially designed to process static data (i.e. non-temporal data) given a declarative query language such as SQL. However, if the aim is to exploit temporal data as required in time series analysis, SQL has its limits because it does not provide useful abstractions such as a window operator. Hence high-level declarative stream query languages, equipped with time-based window operators were designed. A challenge of those abstractions is the additional overhead of the algorithms that automatically transform high-level queries into low-level queries executable over DBMSs. If not handled properly those transformation algorithms may result in low-level queries with processing times too long for agents to make decisions. We describe a robust and optimized transformation algorithm for a high-level declarative stream query language and show that it leads to low-level queries with feasible processing times on real-world data.

Introduction

Observations can be of any type, such as, but not limited to, raw or event based data, and they have in common that they are made at a given time point. Usually, such observations are stored as historical data in a DBMS to be analyzed later or directly processed as stream data at a data stream management system (DSMS) at which a registered query is continuously executed on incoming streams of data. Historical data can be very large and be associated with tens or even hundreds of other, possibly non-temporal, datasets stored in a DBMS. For instance, a medical database contains thousands of historical datasets, containing observations at intervals of a few milliseconds, associated with static data about the observed patients. Manually writing queries to be executed by a DBMS on such large and diverse datasets can be cumbersome and might lead to very long processing times, too long for an agent to make decisions, if not all optimizations possible are exploited for reducing the time it takes to process the data. Such optimizations include knowledge about the exact schema of the database containing the to be processed data, all features a query language provides, and of how a query is translated into an execution plan by a database. Among others, some features of standardized DBMS implementations are mostly unknown, such as procedural code that can be combined with SQL, running directly on database side.

Access to complex, diverse, and distributed, usually non-temporal data stored in a DBMS can be facilitated by Ontology-Based Data Access (OBDA). High-level queries are answered by transforming them into queries on data source level. A query is transformed by first rewriting it, where knowledge from an ontology about a specific domain is incorporated into the query, and then unfolding it with respect to a set of mappings. The ontology is a knowledge base and can be maintained by an expert of a specific domain such as an engineer. Mappings map ontology predicates into a query on data source level such as SQL. Our system builds on Ontop (Calvanese et al. 2017) for relevant OBDA tasks—in particular for transforming SPARQL queries to SQL queries on data-source level.

This work deals with a robust transformation algorithm for STARQL (Streaming and Temporal ontology Access with a Reasoning-based Query Language), a high-level declarative stream query language (Özçep, Möller, and Neuenstadt 2014). Via ontology and time-based window operators, STARQL adds additional abstraction layers that supports users in expressing their information needs. A challenging aspect of the abstraction layer is the added overhead in transforming it to data-source level queries.

So, one would like to ensure that the transformation of a STARQL query—and in general, of queries of any other high-level stream query language following the OBDA paradigm—leads to an efficiently processable query on data-source level. In particular, feasible query answering on data-sources is of utmost importance for decision-making agents

[*]This research was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC 2176 'Understanding Written Artefacts: Material, Interaction and Transmission in Manuscript Cultures', project no. 390893796.

[†]This research was funded by the Federal Ministry of Education and Research of Germany (BMBF) within the project Smadi under grant number 13XP5124A/B.
Copyright © 2023 by the authors. All rights reserved.

acting in an environment where quick decisions need to be made based on historical data. Execution times for transformed STARQL queries should be no longer than as if the information need of the user is expressed as a data-source level query by an IT-expert.

In this paper we present three different optimized algorithms for transforming a STARQL query into a set of queries executable at a DBMS. These new algorithms do not only minimize the overhead of the abstraction layer introduced by the transformation with respect to an ontology, but additionally they reduce the processing times of resulting queries by implementing efficient incremental time-based window operators on data-source level matching the semantics of STARQL window operators. As we show in the evaluation, our optimizations reduce processing times all together from hours to a few minutes.

The basic transformation algorithm we start with (Özçep, Möller, and Neuenstadt 2014; Neuenstadt 2017) was tested on streaming and historical data from industrial partners in the domain of turbine monitoring and wellbore-related data (Giese et al. 2015). The first refined algorithm optimizes the original transformation algorithm by reducing the number of required joins and the size of intermediate results. However, even with this refined algorithm, processing times and the size of intermediate results are still high. Hence, in a second refinement, we enhance the query generation by incorporating optimizations relying on the scripting language Procedural Language/PostgreSQL structured query language (PL/pgSQL). The size of intermediate results does no longer depend on the size of the to be processed data, and all joins required for implementing a time-based window operator are eliminated. The adaptation of PL/pgSQL implementations, to be executed on other DBMS, is rather minimal, as PL/pgSQL is similar to the SQL/PSM standardized language that can be executed on other DBMS implementations.

A last refinement is specially meant to deal with big data by parallelizing the data-source level queries resulting out of the transformation. The possible degree of parallelization is almost unlimited, without great loss of performance, except that with a very high degree of parallelization, reading data and writing results to disk can become a bottleneck.

Using synthetic data and real-world data from various use cases, we can show that our refinements result in processing times that grow linearly rather than polynomially with input size, as is the case with the basic transformation algorithm, which is a big step toward transforming high-level queries into efficient queries on data source-level.

STARQL Query Language

STARQL is a high-level declarative stream query language (Özçep, Möller, and Neuenstadt 2014) getting streams of assertions as input and outputting such streams.

It is a high-level declarative language, as streams are processed with respect to background knowledge in form of an ontology. In description logic (DL), an ontology consists in general of a terminological box (tbox) $\mathcal{T}$, containing axioms representing given background knowledge, and an assertional box (abox) $\mathcal{A}$ containing assertions, thus facts, about the actual world. When considering streams, we make a distinction between static and temporal aboxes. Whereas the static abox $\mathcal{A}_{static}$ contains time independent facts, the temporal abox contains (part of) time-dependent streams. Thus, the ontology $\mathcal{O}$ is defined as $\mathcal{O} = \{\mathcal{T}, \mathcal{A}_{static}\}$.

An example ontology is the semantic sensor network (SSN) ontology presented in (SSN 2008). SSN specifies sensors and observations on them having some measured value. Each sensor is a system that runs on a platform that is part of a deployment. A sensor produces a stream of abox assertions of the form $hasSimpleResult(o_1, 93)\langle 0s\rangle$, asserting that 93 is observed as o_1 at $0s$. Here is an example of an ontology $\mathcal{O}$ (in DL notation) that uses SSN vocabulary and consists of the tbox $\mathcal{T}$

$$\begin{aligned}\{&Observation \sqsubseteq \exists madeBySensor.Sensor,\\ &Sensor \sqsubseteq System,\\ &System \sqsubseteq \exists isHostedBy.Platform,\\ &Platform \sqsubseteq \exists inDeployment.Deployment\}\end{aligned} \tag{1}$$

and a static abox $\mathcal{A}_{static}$

$$\begin{aligned}\{&Sensor(TC260), Deployment(Generator),\\ &Platform(Turbine)\}\end{aligned} \tag{2}$$

The tbox stores intensional information that allows to complete the abox. For example, it directly follows from the tbox expressed in (1) that $TC260$ is a $System$ in addition to being a $Sensor$.

STARQL has a framework character, as different query languages can be embedded into the language, and a STARQL query can refer to a wide range of ontology DL languages. Here, STARQL embeds SPARQL (SPARQL 2008) and refers to DL-Lite ontologies (Calvanese et al. 2007) in OWL 2 QL format (OWL 2004), the standard profile tailored towards classical OBDA.

Typically, historical data consists of tuples with an associated schema. However, a STARQL query has virtual abox assertions as input. Here, by "virtual" we mean that abox assertions from the user point of view are actually data points with an associated timestamp and are not materialized but only presented in an SQL-view-like manner as logical (timestamped) facts to the user. Mappings map ontology predicates such as abox assertions into a query on data source level. To bridge the gap between historical data and virtual abox assertions, the STARQL query Q is answered by, first rewriting the query into Q_{rew} with respect to an ontology, then unfolding the query into query Q_{unf} with respect to a set of mappings. During rewriting, knowledge stored in the ontology is incorporated into the query. The mappings actually bridge the gap between the ontology and database world. This approach is called OBDA (Poggi et al. 2008) and is visualized in Figure 1.

A query $Q(x, y) = System(x) \wedge madeBySensor(o, x) \wedge hasSimpleResult(o, y)\langle t\rangle$ returns sensors together with values they have observed at time points t. Query Q has a sequence of abox assertions of the form of $hasSimpleResult(TC260, 93)\langle 0s\rangle$ as input. Ontology $\mathcal{O}$ contains the tbox $\mathcal{T}$ from (1) and the abox $\mathcal{A}_{static}$ from (2). A database, containing two relations, has the schema as

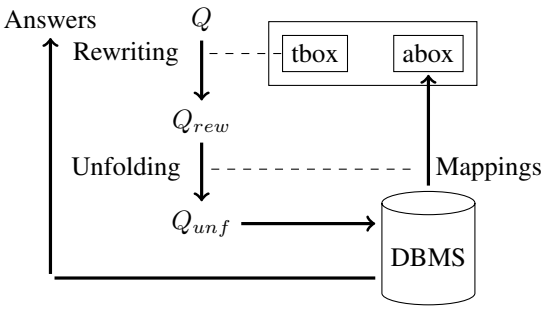

Figure 1: Ontology-Based Data Access

```
stream(obs, timestamp, sensor, value)
sensor(id, assemblypart, name, type)
assemblypart(id, assembly, name)
assembly(id, name)
stream.sensor → sensor.id
sensor.assemblypart → assemblypart.id
assemblypart.assembly → assembly.id
```

Table 1: Schema of a PostgreSQL database

listed in Table 1. Five mappings bridge the database world to the ontology world:

m_1: $Sensor(x) \leftarrow$ **SELECT** name **FROM** sensor

m_2: $madeBySensor(o, x) \leftarrow$ **SELECT** stream.obs **AS** o, sensor.name **AS** x **FROM** stream, sensor **WHERE** stream.sensor = sensor.id

m_3: $hasSimpleResult(o, x) \leftarrow$ **SELECT** stream.obs **AS** o, stream.value **AS** x **FROM** stream

m_4: $isHostedBy(x, y) \leftarrow$ **SELECT** sensor.name **AS** x, assemblypart.name **AS** y **FROM** sensor, assemblypart **WHERE** sensor.assemblypart = assemblypart.id

m_5: $inDeployment(x, y) \leftarrow$ **SELECT** assemblypart.name **AS** x, assembly.name **AS** y **FROM** assemblypart, assembly **WHERE** assemblypart.assembly = assembly.id

The heads of the mappings are in DL notation, while the tails are in a query language of the back-end database (here SQL). Query Q contains the atom $System(x)$, but none of the mappings $m_{1 \leq i \leq 5}$, contain $System(x)$ in their head. However, the ontology $\mathcal{O}$ contains a tbox, where $Sensor \sqsubseteq System$ states that every sensor is a system. Query Q can be rewritten into $Q_{rew}(x, y) = Sensor(x) \wedge madeBySensor(o, x) \wedge hasSimpleResult(o, y)\langle t \rangle$ with respect to the ontology $\mathcal{O}$. Using mappings m_1, m_2, and m_3, query Q_{rew} is unfolded into a query Q_{unf} of a back-end (here SQL), as listed in Table 2. A logical $\wedge$ is transformed into a **NATURAL JOIN**, and a logical $\vee$, if necessary, into a **UNION**. Query Q can be executed over abox assertions from an ontology perspective and query Q_{unf} can be executed over historic data from a relational database perspective.

STARQL queries have (possibly infinite, synchronous) streams of abox assertions as input and output. Each abox of an input stream has an associated timestamp t from a flow of time $(T, \leq)$, where $\leq$ is a linear order. STARQL provides a time based window operator, which groups the abox assertions of each input stream into streams of finite sets. Syntactically, a window is declared for an input stream S as S [NOW - wr, NOW] -> sl, where wr denotes the range, sl the slide, and NOW the current local time t_{str} of a window. A global time t_{pulse} can be declared by the PULSE declaration as START = st, END = end, FREQUENCE = fr for synchronising input streams. Only abox assertions of the input stream, with an associated timestamp $t : st \leq t \leq end$, are processed. The global time t_{pulse} evolves over time:

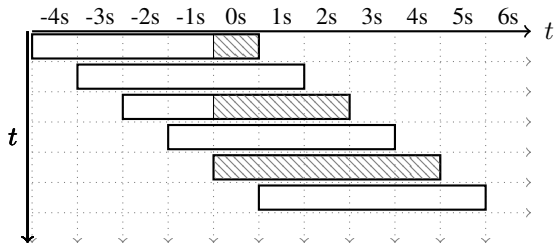

Figure 2: Window with $wr = 4s$, $sl = 1s$, and pulse with $st = 0$s, $end = 5s$, $fr = 2s$

$$t_{pulse} = st \rightarrow st + fr \rightarrow st + 2fr \rightarrow \ldots \quad (3)$$

Each local time t_{str} of each window evolves over time and is maximal with respect to the current global time t_{pulse}:

$$t_{str} = t_{pulse} - (t_{pulse} \bmod sl) \quad (4)$$

The global time t_{pulse} evolves and each window slides m times until m is maximal in such a way that $t_{str} + m \cdot sl \leq t_{pulse}$ holds. Each window with local time t_{str} contains

$$\{a\langle t \rangle \mid t_{str} - wr \leq t \leq t_{str}\} \quad (5)$$

where $a\langle t \rangle$ denotes the abox assertion with associated timestamp t. A given STARQL query should return the same result for each input, as defined, regardless of whether the input is a stream or a historic dataset. Therefore, $t_{str} \leq t_{pulse}$ is only allowed while processing historic data and streams in near real time (Kharlamov et al. 2019). If $t_{str} > t_{pulse}$ holds, then a window could contain abox assertions with an associated timestamp $t > t_{pulse}$, which is impossible while processing streams in near real-time.

Each abox at each window is grouped into mini-bags depending on a sequencing method SEQUENCE BY $seqMethod$, such as the standard sequencing StdSeq. StdSeq groups each abox assertion $a\langle t \rangle$ with the same timestamp t into the same mini-bag. An example is depicted in Figure 2, at where only those abox assertions are processed that fall under the dashed areas of the windows.

The *having* and *where* part of a STARQL query is transformed into SQL as described, at where the concept $System$ is rewritten in our example query into $Sensor$ and finally the query is unfolded into SQL. The transformed SQL query needs to be applied on each grouped set of virtual abox assertions, depending on the pulse and window declaration of

Q_{rew}	Q_{unf}
$Q_{unf}(x,y) =$	**SELECT** x, y **FROM**
$Sensor(x)$	(**SELECT** name **AS** x **FROM** sensor) m1
$\wedge$	**NATURAL JOIN**
$madeBySensor(o,x)$	(**SELECT** stream.obs **AS** o, sensor.name **AS** x **FROM** stream, sensor
	WHERE stream.sensor = sensor.id) m2
$\wedge$	**NATURAL JOIN**
$hasSimpleResult(o,y)$	(**SELECT** stream.obs **AS** o, stream.value **AS** y **FROM** stream) m3;

Table 2: Unfolding of query Q_{rew} into Q_{unf}

a STARQL query. However, historic data stored in a PostgreSQL database is not grouped already into finite sets depending on a window operator part of a STARQL query and PostgreSQL does not provide a time based window operator, matching the semantics of our STARQL window operator. In the next section we describe how we implement an efficient STARQL window operator that can be executed on a PostgreSQL database. Our implementation can be adapted easily to be executable on other database implementations as well, such as Apache Spark.

Efficient STARQL Window Operators

The first implementation of transforming STARQL queries into SQL (Özçep, Möller, and Neuenstadt 2014; Neuenstadt 2017), is based on the creation of different relations depending on the pulse, window, and sequencing strategy. These relations are joined with historic data, referred by a STARQL query, to group the abox assertions of historic data stored in a PostgreSQL database into finite sets. This approach only makes use of SQL which leads to compatibility with other DBMS implementations, providing a standardized SQL interface. As a drawback, many joins are involved and disk space is needed for intermediate results. Every abox assertion of historic data needs to be grouped into one or more sets depending on a window declaration of a STARQL query. Instead of creating a new relation for each grouped set, all grouped sets are stored in one single relation. Therefore, a window id is needed to distinguish between every grouped set. Additionally, an abox id column is created for intra abox sequencing at each window. More specifically, a relation `groups(wid, abox, left, right)` is created via SQL depending only on the range and slide of a window, and frequency of the pulse as declared by a STARQL query. Column `wid` is a numerical unique identifier for each window. Relation `groups` is joined on relation `stream` with `left <= timestamp` **AND** `timestamp <= right`. Column `abox` is a unique numerical identifier assigned to each timestamp in each window. The unfolded *where* and *having* part of a STARQL query is executed on relation `groups`. Finally, the result is joined with relation `pulse(timestamp, wid)` for associating each window with the timestamp declared by the frequency of a STARQL query.

A new idea presented in this paper is to optimize queries that use standard sequencing `StdSeq`. Instead of using a window id `wid` to distinguish grouped sets stored inside a single relation, we use for each window the current time t_{pulse}, depending on the pulse declaration `PULSE` as `START` = st, `END` = end, `FREQUENCE` = fr, because it is needed anyway in the output. An additional column for abox ids is omitted, as the existing timestamp of an abox assertion is sufficient for intra abox sequencing, using standard sequencing `StdSeq`, where each abox assertion with the same timestamp is grouped into the same mini-box. Therefore, instead of creating an extra column for the abox ids, the existing column of timestamps associated with the abox assertions is used. The number of intermediate relations is reduced to a single one, containing the grouped sets of abox assertions, to reduce disk space needed for intermediate results and to reduce the number of required joins. This new idea leads to considerably shorter processing times (as can be seen in the next section).

Even though the new idea leads to shorter processing times, disk space is needed for intermediate results. For omitting intermediate results at all and for reducing the processing times further, we forgo to transform STARQL queries into plain SQL. Instead, queries are transformed into a combination of SQL and procedural code based on PL/pgSQL.

The rewritten and unfolded *where* and *having* part of a STARQL query is executed on each window, using PL/pgSQL, as listed in Algorithm 1. Here, Q_{unf} is the rewritten and unfolded query Q, st/end the start/end at where the stream is being processed, fr the frequency, wr the range of the windows, and sl the slide of the windows. No window id, abox id, or an intermediate relation is required. Processing times are reduced and are linear instead of polynomial as it is the case for the two previous approaches.

Algorithm 1 Execute SQL

```
1: procedure EXECUTESQL(Q_unf, st, end, fr, wr, sl)
2:     t_str ← st
3:     while t_str ≤ end do
4:         we ← t_str − (t_str mod sl)
5:         ws ← we − wr
6:         PROCESSWINDOW(Q_unf, stream[ws : we])
7:         t_str ← t_str + fr
```

For further reducing the time required it takes to process each window, the procedure listed in Algorithm 1 is executed in parallel. That allows for answering STARQL queries horizontally scaled across all cores available on a

node or even across several nodes, if the data is stored in an appropriately distributed manner.

Finally, we transform STARQL queries to plain SQL that can be executed on Apache Spark (Zaharia et al. 2016). As well as PostgreSQL, Apache Spark lacks of a window operator matching the semantics of the STARQL window operator. The available window operators do not allow for executing arbitrary SQL queries that originate from the transformation of the *where* and *having* part of a STARQL query on each grouped set of historic data. Hence, we reuse our approach of transforming STARQL into a plain SQL query, at where no window and abox ids are generated, for using Apache Spark as a back-end.

Evaluation

We evaluate our approach based on two different historic datasets. One is synthetic and the other one stems from real world material sciences:

Synthetic SSN-data The data has the same schema as listed in Table 1, and queries are answered with respect to the tbox listed in (1) and virtual aboxes listed in (2).

Real-world data The data stems from use cases in materials sciences. Those use cases were developed in the project SmaDi[1] aiming at the digitalization of smart materials. The use cases involve different types of queries possible, ranging from parameter values at specific time points, e.g., steps in an experiment, to input or output values of models and it is also possible to query values calculated on-the-fly based on user defined functions which complicates the query answering additionally. The data resides in a PostgreSQL-database and is accessed via an ontology incorporating approximately 230 classes and 30 relations and 600 axioms in total.

The evaluation results are similar for the synthetic and the real-world data. For visualization, in the following, the synthetic data is used for the general evaluation, as the size of the data and the number of observations per time step can be chosen arbitrarily and thus our approaches can be tested under different conditions.

We evaluate all our presented approaches on synthetic data, at where 20 sensors made minute wise observations 300 days long and measure of how long it takes to process the data given the STARQL query listed in (6).

```
SELECT x
FROM stream [NOW - 20m, NOW] -> 1m
WHERE System(x) ∧ madeBySensor(o, x)
HAVING FORALL ti, tj, y1, y2                    (6)
IF hasSimpleResult(o, y1)⟨ti⟩
AND hasSimpleResult(o, y2)⟨tj⟩
AND ti < tj THEN y1 <= y2
```

For simplicity, the frequency fr is set equal to the slide sl of the window. The query returns all sensors x that have observed monotonic increasing values in the last 20 minutes at each time step in the data.

[1]https://www.materialdigital.de/project/10

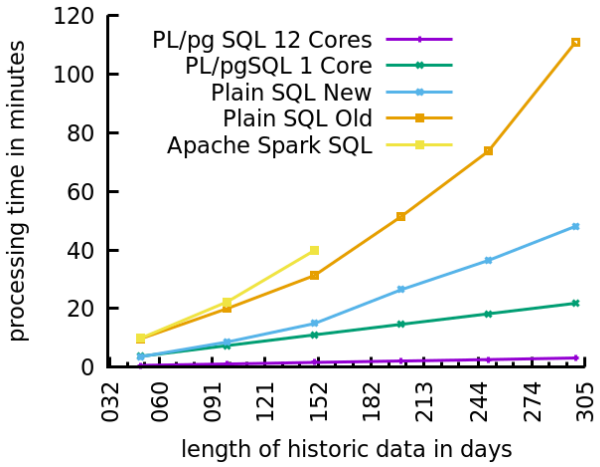

Figure 3: Processing times

The node we use in our experiments has a single AMD Ryzen 5 3600 6-Core Processor with up to 3.6 GHz, 64 GB-Ram, running Ubuntu 22.04, PostgreSQL 14.6, Apache Spark 3.3.1, Apache Hadoop 3.3.4, and Apache Hive 3.1.3. We use Ontop 5.0.1 (Calvanese et al. 2017) for rewriting and unfolding queries with respect to the ontology $\mathcal{O}$ listed in (1) and (2). Results are depicted in Figure 3. Surprisingly, all four approaches based on the PostgreSQL backend are faster than using Apache Spark. The processing of stream data that has a length of more than 152 days, using Apache Spark, was terminated after more than three hours. Processing times could be reduced by using more than one node, as Apache Spark is able to scale across many nodes. Although it is disadvantageous to require more computing power, more than one node can be used.

Our new approach of transforming STARQL queries into plain SQL supersedes the old approach at where window and abox ids are generated. However, both approaches grow polynomially in terms of the time it takes to process the data, and space required for storing intermediate results is enormous. Transforming STARQL queries into a combination of SQL and PL/pgSQL leads to processing times that grow linearly with increasing size of the data and space required for intermediate results is bound to the maximum size of data fitting within the range of a specific time window. Processing times are further reduced by executing the PL/pgSQL function in parallel. In case of processing 300 days of data, the old SQL approach needs 111 minutes and the combination of parallel SQL and PL/pgSQL only three minutes.

STARQL embeds SPARQL and refers to DL-Lite ontologies in OWL 2 QL format. In case of the STARQL query listed in (6), we use Ontop for rewriting and unfolding embedded SPARQL queries, such as the query listed in (7).

$$Q(x, o) = System(x) \wedge madeBySensor(o, x) \qquad (7)$$

If we extend the STARQL query listed in (6) to the one in (8), it yields the exact same results, as all sensors are hosted

by a platform, and a platform is always part of a deployment.

```
SELECT x
FROM stream [NOW - 20m, NOW] -> 1m
WHERE
```
$$System(x) \wedge madeBySensor(o,x) \wedge isHostedBy(s,p) \wedge inDeployment(p,d) \wedge Observation(o) \wedge Platform(p) \wedge Deployment(d) \quad (8)$$
```
HAVING FORALL ti, tj, y1, y2
IF hasSimpleResult(o, y1)⟨ti⟩
AND hasSimpleResult(o, y2)⟨tj⟩
AND ti < tj THEN y1 <= y2
```

In case of the extended SPARQL query listed in (8), it is not wrong to use Ontop for rewriting the SPARQL query listed in (9) to SQL.

$$Q(x,o,s,p,d) = System(x) \wedge madeBySensor(o,x) \wedge isHostedBy(s,p) \wedge inDeployment(p,d) \wedge Observation(o) \wedge Platform(p) \wedge Deployment(d) \quad (9)$$

However, variables s, p, and d are unbound in the STARQL query, and not further required in the result set of the SPARQL query. Ontop returns a SQL query at where relations `stream`, `sensor`, `assemblypart`, and `assembly` are joined. If we rewrite query $Q(x,o,s,p,d)$ to $Q'(x,o)$ the result is the same and Ontop returns a SQL query at where only the relation `stream` is referred in the query without any join at all. Answering $Q'(x,o)$ requires only x and o which are mapped to the relation `stream` and with the constraints in the database listed in Table 1, Ontop can conclude automatically that a sensor only exists iff it is part of an `assemblypart`, and that an `assemblypart` only exists if it is part of an `assembly`. Depending on the size of the historic data to be processed, and the window range and slide of the STARQL query, such optimization can make a huge difference. Processing one week of stream data at where 20 sensors made minute-wise observations with the STARQL query listed in (8) needs 53 minutes to be processed when not optimized. If the use of Ontop is optimized, then it takes only 16.55 seconds to process the query. This can also be observed for the real-world data set, where it takes 12.01s without such optimizations, and 11.07s with such optimizations. Though, the data set is smaller than the synthetic one and has a more complex structure, thus, the effect between optimized and not-optimized query is not as big as for the synthetic data, but even for this, an advantage can be seen which will increase when more data is used as input. Hence, we use Ontop for transforming SPARQL queries, embedded in a STARQL query, at where only bound variables are added to the `SELECT` part of a SPARQL query. That not only reduces the time required for processing historic data, additionally it reduces the time required for processing streams as well, as optimizations are applied for each window.

Related Work

Ever since the formal foundations of stream processing has been laid down in the database community with the query language CQL (Arasu, Babu, and Widom 2006), DSMSs with CQL languages have become a background engine and a blueprint for processing high-level streams w.r.t. background knowledge. For many recent stream engines (as ours based on STARQL) that background knowledge is an ontology represented in OWL 2 or a description logic: examples are Streaming-SPARQL (Bolles, Grawunder, and Jacobi 2008), SPARQLstream (Calbimonte et al. 2012), C-SPARQL (Barbieri et al. 2010), CQELS (Phuoc et al. 2011), RSP-QL (Dell'Aglio et al. 2015), EP-SPARQL (Anicic et al. 2011), TEF-SPARQL (Kietz et al. 2013), (Borgwardt, Lippmann, and Thost 2015). Other high-level stream engines rely on background knowledge that is rule-based. In particular regarding the latter, recent approaches consider representations formalisms that extend Datalog with temporal or window operators following early work on deductive temporal databases (Chomicki and Imieliński 1988; Abadi and Manna 1989; Toman, Chomicki, and Rogers 1994): examples are DatalogMTL (Brandt et al. 2017; Wałega et al. 2019; Walega, Kaminski, and Grau 2019; Ronca et al. 2022), Streamlog (Zaniolo 2012) and LARS/Laser (Bazoobandi, Beck, and Urbani 2017; Beck, Dao-Tran, and Eiter 2018). As high-level stream processing w.r.t. a background knowledge adds a further (for sure: useful) abstraction level to the overall system, it leads to additional challenges. In this paper we demonstrate how to cope with the overhead generated by such abstractions in a feasible way, namely by describing a generic transformation of the window mechanism and by exploiting parallelism. Further challenges regarding the produced overhead are space requirements dealt with under the term "memory-boundedness" (Arasu et al. 2004). Based on criteria of (Arasu et al. 2004), in (Schiff and Özçep 2020) bounded memory criteria for streams with application time are defined that rewritten STARQL queries have as input. Not only processing times are enormously reduced for a specific class of rewritten STARQL queries with an unbounded window, additionally only a bounded amount of memory is required during processing.

Conclusion and Future Work

A STARQL query is answered by transforming it into a query which can be executed at a back-end system at where data is stored/received. This approach is more appropriate than writing a query over diverse, rapidly changing and distributed back-end databases manually. As shown in this paper, rewriting and unfolding needs to be done with care, as otherwise processing times can be too long for an agent to make decisions. We were able to reduce processing times from hours to minutes or even seconds by optimizing various steps during the transformation of a STARQL query. The size of intermediate results depends on the window operator of a STARQL query and no longer on the data itself, and grouping data into finite sets based on the window operator no longer requires joins over the complete input data that can be very large.

References

Abadi, M., and Manna, Z. 1989. Temporal logic programming. *Journal of Symbolic Computation* 8(3):277–295.

Anicic, D.; Fodor, P.; Rudolph, S.; and Stojanovic, N. 2011. Ep-sparql: a unified language for event processing and stream reasoning. In *Proceedings of the 20th international conference on World wide web*, WWW '11, 635–644. New York, NY, USA: ACM.

Arasu, A.; Babu, S.; and Widom, J. 2006. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal* 15:121–142. 10.1007/s00778-004-0147-z.

Arasu, A.; Babcock, B.; Babu, S.; McAlister, J.; and Widom, J. 2004. Characterizing Memory Requirements for Queries over Continuous Data Streams. *ACM Transactions on Database Systems (TODS)* 29(1):162–194.

Barbieri, D. F.; Braga, D.; Ceri, S.; Valle, E. D.; and Grossniklaus, M. 2010. C-sparql: a continuous query language for rdf data streams. *Int. J. Semantic Computing* 4(1):3–25.

Bazoobandi, H. R.; Beck, H.; and Urbani, J. 2017. Expressive stream reasoning with laser. In d'Amato, C.; Fernandez, M.; Tamma, V.; Lecue, F.; Cudré-Mauroux, P.; Sequeda, J.; Lange, C.; and Heflin, J., eds., *The Semantic Web – ISWC 2017*, 87–103. Cham: Springer International Publishing.

Beck, H.; Dao-Tran, M.; and Eiter, T. 2018. Lars: A logic-based framework for analytic reasoning over streams. *Artificial Intelligence* 261:16–70.

Bolles, A.; Grawunder, M.; and Jacobi, J. 2008. Streaming sparql - extending sparql to process data streams. In Bechhofer, S.; Hauswirth, M.; Hoffmann, J.; and Koubarakis, M., eds., *The Semantic Web: Research and Applications*, volume 5021 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg. 448–462.

Borgwardt, S.; Lippmann, M.; and Thost, V. 2015. Temporalizing rewritable query languages over knowledge bases. *Journal of Web Semantics* 33:50–70. Ontology-based Data Access.

Brandt, S.; Kalayci, E. G.; Kontchakov, R.; Ryzhikov, V.; Xiao, G.; and Zakharyaschev, M. 2017. Ontology-based data access with a horn fragment of metric temporal logic. In Singh, S. P., and Markovitch, S., eds., *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA.*, 1070–1076. AAAI Press.

Calbimonte, J.-P.; Jeung, H.; Corcho, O.; and Aberer, K. 2012. Enabling query technologies for the semantic sensor web. *Int. J. Semant. Web Inf. Syst.* 8(1):43–63.

Calvanese, D.; De Giacomo, G.; Lembo, D.; Lenzerini, M.; and Rosati, R. 2007. Tractable Reasoning and Efficient Query Answering in Description Logics: The DL-Lite Family. *Journal of Automated reasoning* 39(3):385–429.

Calvanese, D.; Cogrel, B.; Komla-Ebri, S.; Kontchakov, R.; Lanti, D.; Rezk, M.; Rodriguez-Muro, M.; and Xiao, G. 2017. Ontop: Answering SPARQL queries over relational databases. *Semantic Web* 8(3):471–487.

Chomicki, J., and Imieliński, T. 1988. Temporal deductive databases and infinite objects. In *Proceedings of the Seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '88, 61–73. New York, NY, USA: Association for Computing Machinery.

Dell'Aglio, D.; Della Valle, E.; Calbimonte, J.; and Corcho, O. 2015. Rsp-ql semantics: A unifying query model to explain heterogeneity of rdf stream processing systems. *International Journal on Semantic Web and Information Systems (IJSWIS)* 10(4).

Giese, M.; Soylu, A.; Vega-Gorgojo, G.; Waaler, A.; Haase, P.; Jiménez-Ruiz, E.; Lanti, D.; Rezk, M.; Xiao, G.; Özçep, Ö. L.; and Rosati, R. 2015. Optique: Zooming in on big data. *IEEE Computer* 48(3):60–67.

Kharlamov, E.; Kotidis, Y.; Mailis, T.; Neuenstadt, C.; Nikolaou, C.; Özçep, Ö.; Svingos, C.; Zheleznyakov, D.; Ioannidis, Y.; Lamparter, S.; Möller, R.; and Waaler, A. 2019. An Ontology-mediated Analytics-aware Approach to Support Monitoring and Diagnostics of Static and Streaming Data. *Journal of Web Semantics*.

Kietz, J.-U.; Scharrenbach, T.; Fischer, L.; Nguyen, M. K.; and Bernstein, A. 2013. Tef-sparql: The ddis query-language for time annotated event and fact triple-streams. Technical Report IFI-2013.07, Department of Informatics.

Neuenstadt, C. 2017. *An Engine for Ontology Based Stream Processing*. Ph.D. Dissertation, University of Lübeck.

2004. OWL Web Ontology Language Overview. Online. Available at https://www.w3.org/TR/owl-features.

Özçep, Ö.; Möller, R.; and Neuenstadt, C. 2014. A Stream-Temporal Query Language for Ontology Based Data Access. In Lutz, C., and Thielscher, M., eds., *KI 2014*, volume 8736 of *LNCS*, 183–194. Springer International Publishing Switzerland.

Phuoc, D. L.; Dao-Tran, M.; Parreira, J. X.; and Hauswirth, M. 2011. 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. F.; and Blomqvist, E., eds., *The Semantic Web - ISWC 2011 - 10th International Semantic Web Conference, Bonn, Germany, October 23-27, 2011, Proceedings, Part I*, volume 7031 of *Lecture Notes in Computer Science*, 370–388. Springer.

Poggi, A.; Lembo, D.; Calvanese, D.; Giacomo, G. D.; Lenzerini, M.; and Rosati, R. 2008. Linking data to ontologies. *Journal of Data Semantics* 10:133–173.

Ronca, A.; Kaminski, M.; Cuenca Grau, B.; and Horrocks, I. 2022. The delay and window size problems in rule-based stream reasoning. *Artificial Intelligence* 306:103668.

Schiff, S., and Özçep, Ö. L. 2020. Bounded-memory criteria for streams with application time. In *The Thirty-Third International Flairs Conference*.

2008. SPARQL Query Language for RDF. Online. Available at https://www.w3.org/TR/rdf-sparql-query.

2008. Semantic Sensor Network. Online. Available at https://www.w3.org/TR/vocab-ssn/.

Toman, D.; Chomicki, J.; and Rogers, D. S. 1994. Datalog with integer periodicity constraints. In *Proceedings of the 1994 International Symposium on Logic Programming*, ILPS '94, 189–203. Cambridge, MA, USA: MIT Press.

Wałega, P. A.; Cuenca Grau, B.; Kaminski, M.; and Kostylev, E. V. 2019. Datalogmtl: Computational complexity and expressive power. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, 1886–1892. International Joint Conferences on Artificial Intelligence Organization.

Walega, P. A.; Kaminski, M.; and Grau, B. C. 2019. Reasoning over streaming data in metric temporal datalog. In *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence and Thirty-First Innovative Applications of Artificial Intelligence Conference and Ninth AAAI Symposium on Educational Advances in Artificial Intelligence*, AAAI'19/IAAI'19/EAAI'19. AAAI Press.

Zaharia, M.; Xin, R. S.; Wendell, P.; Das, T.; Armbrust, M.; Dave, A.; Meng, X.; Rosen, J.; Venkataraman, S.; Franklin, M. J.; et al. 2016. Apache SPARK: A Unified Engine for Big Data Processing. *Communications of the ACM* 59(11):56–65.

Zaniolo, C. 2012. Logical foundations of continuous query languages for data streams. In *Proceedings of the Second International Conference on Datalog in Academia and Industry*, Datalog 2.0'12, 177–189. Berlin, Heidelberg: Springer-Verlag.