Stream-temporal Querying with Ontologies

Ralf Möller, Christian Neuenstadt, and Özgür L. Özçep

Institute of Information Systems (Ifis)
University of Lübeck
Lübeck, Germany
{moeller,neuenstadt,oezcep}@ifis.uni-luebeck.de

Abstract. Recent years have seen theoretical and practical efforts on temporalizing and streamifying ontology-based data access (OBDA). This paper contributes to the practical efforts with a description/evaluation of a prototype implementation for the stream-temporal query language framework STARQL. STARQL serves the needs for industrially motivated scenarios, providing the same interface for querying historical data (reactive diagnostics) and for querying streamed data (continuous monitoring, predictive analytics). We show how to transform STARQL queries w.r.t. mappings into standard SQL queries, the difference between historical and continuous querying relying only in the use of a static window table vs. an incrementally updated window table. Experiments with a STARQL prototype engine using the PostgreSQL DBMS show the implementability and feasibility of our approach.

Keywords: stream reasoning, OBDA, monitoring, temporal reasoning

1 Introduction

This paper contributes results to recent efforts for adapting the paradigm of ontology-based data access (OBDA) to scenarios with streaming data [12,6,22] as well as temporal data [5,4]. It extends our previous work [20,21,19]—started in the context of the IP7 EU project Optique and resulting in the query language framework STARQL (Streaming and Temporal ontology Access with a Reasoning-based Query Language)—with a proof-of-concept implementation that is based on PostgreSQL. STARQL serves the needs for industrially motivated scenarios, providing the same interface for querying historical data—as needed for reactive diagnostics—and for querying streamed data—as needed for continuous monitoring and predictive analytics in real-time scenarios.

Considering streams, the main challenge is that data cannot be processed as a whole. The simple but fundamental idea is to apply a (small) sliding window which is updated as new elements from the stream arrive at the query-answering system (see, e.g., [3]). The idea of previous approaches adapting OBDA to streams [12,6,22] is that answering continuous ontology-level queries on streams can be reduced to answering ontology-level queries on dynamically updated finite window contents, which are treated as single ABoxes. This approach can

lead to unintended inconsistencies, as exemplified by industrial use cases such as that of Siemens, one of the industrial stakeholders in the Optique project. For example, multiple measurement values for a sensor collected at different time points lead to inconsistencies since the value association should be functional. In STARQL, the idea of processing streams with a window operator is pushed further by defining a finite *sequence of consistent ABoxes* for each window.

Considering temporal reasoning for reactive diagnostics as needed for the Optique use case provided by Siemens [14], we found that a window based approach leads to elegant solutions as well. A window is used to focus on some subset of temporal data in a temporal query. Now, over time, different foci are relevant for reactive diagnosis. Thus, foci changes can be modelled by a window virtually "sliding" over temporal data, albeit the case that sliding is not done in realtime. Thus, STARQL is defined such that it can be used equally for temporal and stream reasoning. The semantics of STARQL does not distinguish between the realtime and the historical querying scenario.

The ABox sequencing strategy for windows required to avoid inconsistencies, as argued above, makes rewriting and, more importantly, unfolding of STARQL queries a challenging task. In particular, one may ask whether it is possible to rewrite and unfold one STARQL query into a single backend query formulated in the query language provided by the backend systems. We show that (Postgre)SQL transformations are indeed possible and describe them in the paper for the special case of one stream with slide parameter identical to pulse parameter and a specific sequencing strategy (for the details see the following section).

2 The STARQL Framework

We recap the syntax and the semantics of STARQL with a simple example. For a complete formal treatment we refer the reader to [20,21]. We assume familiarity with the description logic DL-Lite [7].

For illustration purposes, our running example is a measurement scenario in which there is a (possibly virtual) stream S_{Msmt} of ABox assertions. A stream of ABox assertions is an infinite set of timestamped ABox assertions of the form $ax\langle t\rangle$. The timestamps stem from a flow of time $(T, \leq)$ where T may even be a dense set and where $\leq$ is a linear order. The initial part of S_{Msmt}, called $S_{Msmt}^{\leq 5s}$ here, contains timestamped ABox assertions giving the value of a temperature sensor s_0 at 6 time points starting with $0s$.

$$S_{Msmt}^{\leq 5s} = \{val(s_0, 90^\circ)\langle 0s\rangle, val(s_0, 93^\circ)\langle 1s\rangle, val(s_0, 94^\circ)\langle 2s\rangle \\ val(s_0, 92^\circ)\langle 3s\rangle, val(s_0, 93^\circ)\langle 4s\rangle, val(s_0, 95^\circ)\langle 5s\rangle\}$$

Assume further, that a static ABox contains knowledge on sensors telling, e.g., which sensor is of which type. In particular, let $BurnerTipTempSens(s_0)$ be in the static ABox. Moreover, let there be a pure DL-Lite TBox with additional information such as $BurnerTipTempSens \sqsubseteq TempSens$ saying that all burner tip temperature sensors are temperature sensors.

The Siemens engineer has the following information need: Starting with time point 00:00 on 1.1.2005, tell me every second those temperature sensors whose value grew monotonically in the last 2 seconds. A possible STARQL representation of the information is illustrated in Listing 1.

```
PREFIX : <http://www.optique-project.eu/siemens>
CREATE STREAM S_out AS
CONSTRUCT  GRAPH NOW { ?s rdf:type MonInc }
FROM STREAM S_Msmt [NOW-2s, NOW]->"1S"^^xsd:duration
     WITH START = "2005-01-01T01:00:00CET"^^xsd:dateTime,
          END = "2005-01-01T02:00:00CET"^^xsd:dateTime
     STATIC ABOX <http://www.optique-project.eu/siemens/ABoxstatic>,
     TBOX <http://www.optique-project.eu/siemens/TBox>
USING PULSE WITH
            START = "2005-01-01T00:00:00CET"^^xsd:dateTime,
            FREQUENCY = "1S"^^xsd:duration
WHERE { ?s rdf:type :TempSens }
SEQUENCE BY StdSeq AS seq
HAVING FORALL ?i < ?j IN seq,?x,?y:
 IF (GRAPH ?i { ?s :val ?x }  AND GRAPH ?j { ?s  :val ?y }) THEN ?x <= ?y
```

Listing 1: Example query in STARQL

After the create expressions for the stream and the output frequency the queries' main contents are captured by the `CONSTRUCT` expressions. The construct expression describes the output format of the stream, using the named-graph notation of SPARQL for fixing a basic graph pattern (BGP) and attaching a time expression, here `NOW`, for the evolving time. The actual result in the example (in DL notation) is a stream of ABox assertions of the form $MonInc(s_0)\langle t\rangle$.

$$S_{out}^{\leq 5s} = \{MonInc(s_0)\langle 0s\rangle, MonInc(s_0)\langle 1s\rangle, MonInc(s_0)\langle 2s\rangle, MonInc(s_0)\langle 5s\rangle\}$$

The WHERE clause binds variables w.r.t. the non-streaming sources (ABox, TBox) mentioned in the `FROM` clause by using (unions) of BGPs. We assume an underlying DL-Lite logic for the static ABox, the TBox and the BGP (considered as unions of conjunctive queries UCQs) which allows for concrete domain values, e.g., DL-Lite$_{\mathcal{A}}$ [7]. In this example, instantiations of the sensors $?s$ are fixed w.r.t. a static ABox and a TBox. The semantics of the UCQs embedded into the WHERE and the HAVING clause ist the certain answer semantics [7].

The heart of the STARQL queries is the window operator in combination with sequencing. The operator `[NOW-2s, NOW]->"1S"^^xsd:duration` used in Listing 1 describes a sliding window, which collects the timestamped ABox assertions in the last two seconds and slides 1s forward in time. Note that the `START` and `END` specifications over the stream: These make sense only for temporal queries over streamed historical data (see Sect. 3.1).

Every temporal ABox produced by the window operator is converted to a sequence of (pure) ABoxes. The sequence strategy determines how the timestamped assertions are sequenced into ABoxes. The sequencing method used in the example is *standard sequencing* (`StdSeq`): assertions with the same timestamp come into the same Abox. So, in the example the resulting sequence of ABoxes at time point 5s is trivial as there are no more than two ABox assertions with the same timestamp: $\{val(s_0, 92°)\}\langle 0\rangle, \{val(s_0, 93°)\}\langle 1\rangle, \{val(s_0, 95°)\}\langle 2\rangle\}$.

Now, at every time point, one has a sequence of ABoxes on which temporal (state-based) reasoning can be applied. This is realized in STARQL by a sorted first-order logic template in which state stamped UCQs conditions are embedded. We use here again the `GRAPH` notation from SPARQL. In our example the HAVING clause expresses a monotonicity condition stating that for all values $?x$ that are values of sensor $?s$ w.r.t the i^{th} ABox (subgraph) and for all values $?y$ that are values of the same sensor $?s$ w.r.t. the j^th ABox (subgraph), it must be the case that $?x$ is less than or equal to $?y$.

The grammar for the HAVING clause (not shown here) exploits a safety mechanism. Without it a `HAVING` clause such as $?y > 3$, with free concrete domain variable $?y$ over the reals, would be allowed: the set of bindings for $?y$ would be infinite. Even more, the safety conditions guarantees that the evaluation of the HAVING clause on the ABox sequence depends only on the active domain not the whole domain, i.e., HAVING clauses are domain independent (d.i.) (see [1] for a definition of domain independence). This in turn guarantees that the HAVING clause can be smoothly transformed into queries of d.i. languages such as SQL or CQL [3]. For the details we refer again to [21].

3 OBDA Transformation of STARQL

As the HAVING clause language is d.i. (see [21]), STARQL can be used as ontology query language in the OBDA paradigm: STARQL queries can be transformed into queries over data sources that are quipped with d.i. query languages.

As (backend) data source candidates we consider any DBMS providing a declarative language such as SQL. This is not a limitation in comparison with those approaches (in particular our own [21,19]) that allow relational data stream management systems (DSMS) as data sources. In fact, the STARQL prototype in the Optique platform uses a stream-extension of ADP [24] which provides a CQL-like [3] query language. For the transformation of this paper we by-pass the additional abstraction of DSMS by reconstructing the implementation of the relational window operators on top of incrementally updated window tables: The operators are the same as for ordinary RDBMS, but the tables are dynamic.

Because our transformation does not rely on a window operator on the backend side but constructs the window contents within a window table, two different implementations become possible: 1. Preprocesses the data in the backend in order to materialize the window table for the whole time interval fixed within the STARQL query. The abstract computation model for this implementation is combined rewriting: The given query is rewritten w.r.t. the TBox and the rewritten query is posed to a pre-processed ABox resulting from the original ABox by (partially) materializing TBox axioms (see [18,17,15]). 2. Generate the window contents on the fly—during query run time. The abstract computation model is that of classical OBDA, in which the query is rewritten w.r.t. the TBox, unfolded w.r.t. the mappings and issued to original data—without any preprocessing of the data source. Our experiments below show that the second approach outperforms the first one. But the former approach is useful as a caching means for

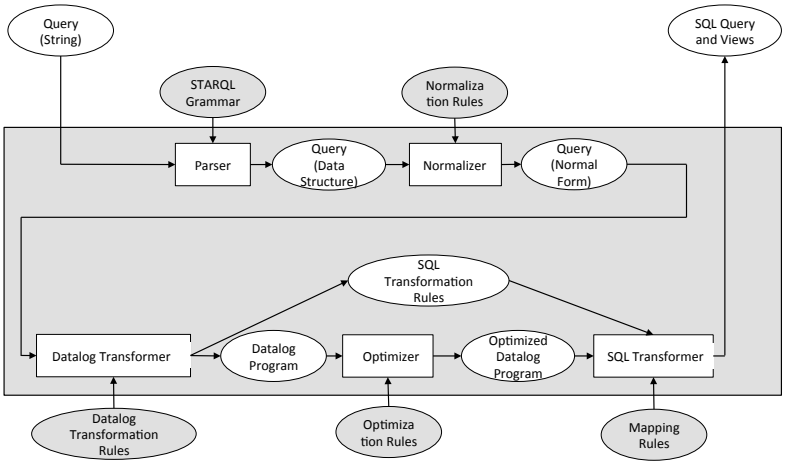

Fig. 1: Processing pipeline for transforming a STARQL query into an SQL query.

scenarios with multiple query processing where (many of) the queries use the same windows on the same streams.

As mentioned before, the aim of this paper is to give a proof-of-concept implementation for a stream-temporal engine. In particular, regarding the required OBDA transformations from the ontology layer to the backend sources, the engine is applicable to the following special case: There is only one stream, the slide is identical to the pulse, and the sequencing strategy is standard sequencing.

3.1 Temporal Reasoning by Mapping STARQL to SQL

For reactive diagnosis as investigated in Optique with the Siemens power plant use case, specific patterns, aka events, are to be found in previously recorded data. Data represents 1.) measurement values of sensors and 2.) turbines with assembly groups, mounted sensors and their properties. Reactive diagnosis deals with analyzing data in order to understand, e.g., reasons for engine shutdown.

If we consider again the information need discussed in Section 2, it becomes clear that in the Siemens use case, patterns to be detected are defined w.r.t. certain time windows in which relevant events take place (e.g., monotonically increasing in a window of 10 minutes, say). Thus, in this context, temporal queries are formulated using time windows in the same spirit as time windows are used for continuous queries in stream processing systems.

The same queries can be registered as a continuous query or a historical query. For the latter, it should obviously be possible to specify a period of interest, i.e., a starting point and an end point for finding answers.

We consider the example query of Listing 1 for demonstrating the transformation of historical queries. Figure 1 shows how historical STARQL queries are processed internally. The prototype is implemented in Prolog, and the rule sets

used as inputs in Fig. 1 are Prolog (DCG) rules. A query is parsed in order to produce parse tree data structures, which then are normalized. Normalization converts FORALL expressions in the HAVING clause into NOT EXISTS NOT expressions by pushing negation inside. The following listing shows the normalized HAVING part of our example query.

```
HAVING NOT EXISTS ?i < ?j IN seq ?x, ?y :
       ( GRAPH ?i { ?s val ?x } AND GRAPH ?j { ?s val ?y } AND ?x > ?y ) ;
```

The normalized query is translated into a datalog program (see Figure 1) with fresh predicate names being generated automatically. The WHERE expression is rewritten and unfolded due to the axioms in the TBox given in the query and the mapping rules, respectively. Here we assume that sensors are created by an SQL view defined below. In the body of the first rule, datalog code for the WHERE clause is inserted (rule q0, see Listing 2)). For every language element found in the parse tree (e.g., `NOT EXISTS...`), we generate a new datalog rule.

For every `{?x rdf:type C}` (`{?x R ?y}`)) mentioned in the WHERE or HAVING clause we insert `C(WID, X)` (`R(WID, X, Y)`) in the datalog program. `WID` is an implicit parameter representing a so-called window `ID`. Correspondingly, for every `GRAPH i {?x rdf:type C}` (`GRAPH i {?x R ?y}`) mentioned in the HAVING clause we insert `C(WID, X, i)` (`R(WID, X, Y, i)`) with i representing the specific ABox in each window sequence.

The datalog clauses `q1` to `q4` in Listing 2 are generated automatically from the HAVING clause of the query above.

```
q0(WID, S) :- sensor(WID, S).
q1(WID, S) :- q0(WID, S), not q2(WID, S).
q2(WID, S) :- seq(WID, I), seq(WID, J), q3(WID, I, J, S), I < J.
q3(WID, I, J, S) :- q4(WID, I, J, X, Y, S).
q4(WID, I, J, X, Y, S) :- val(WID, I, X, S), val(WID, J, Y, S), X > Y.
```

Listing 2: Generated datalog rules

The mapping specifications of the stream `s_msmt`, the relations `val` and `sensor` are predefined using datalog clauses and added to the datalog rules from a file. The additional mapping rules are shown in Listing 3.

```
val(WID, Index, Sensor, Value) :-
   window(WID, Index, Timestamp),
   measurement(Timestamp, Sensor, Value).
sensor(WID, Sensor) :- val(WID, _Index, Sensor, _Value).
seq(WID, Index) :- window(WID, Index, _Timestamp).
```

Listing 3: Mapping specifications using datalog clauses

During clause generation, SQL transformation rules are generated (see Figure 1). Transformation rules represent the name of the SQL relation, the number of arguments as well as the type of each parameter. SQL transformation are statically specified also for the relations val, window, and seq.

For the `CREATE STREAM` and `CONSTRUCT` expression in the query, a clause

```
s_out(T, S) :- q1(WID, S), window_range(WID, T).
```

is added to the datalog program.

The datalog program generated by the module Datalog Transformer (see Figure 1) is then optimized. Optimization eliminates wrapper clauses. An according optimization of Listing 2 is shown below.

```
q1(WID, S) :- sensor(WID, S), not q2(WID, S).
q3(WID, I, J, S) :- seq(WID, I), seq(WID, J), val(WID, I, X, S),
         val(WID, J, Y, S), X > Y, I < J.
```

Here, the datalog rules `q0` and `q1` have been reformulated to `q1` and `q2` to `q4` from Listing 2 have been reformulated by eliminating wrapper clauses to `q3`.

Moreover, body atoms are removed if bindings are already generated by other atoms. In our example we see that a `seq` clause is already a subclause of the `val` clause (Listing 3). Using semantic query optimization [8,9], both `seq` atoms can be eliminated in `q3`. As a consequence also the clause for `seq` is eliminated.

```
s_out(Right, S) :- q1(WID, S), window_range(WID, _Left, _Right).
q1(WID, S) :- sensor(WID, S), not q3(WID, S).
q3(WID, S) :- val(WID, I, X, S), val(WID, J, Y, S), I < J, X > Y.
sensor(WID, Sensor) :- val(WID, _Index, Sensor, _Value).
val(WID, Index, Sensor, Value) :-
   window(WID, Index, Timestamp),
   measurement(Timestamp, Sensor, Value).
```

Listing 4: Optimized datalog rules

The datalog program is non-recursive and safe. So it can be translated to SQL as shown in Listing 5. The column names for relations are given by the SQL Transformation Rules generated by the Datalog Transformer and by mapping rules given as input to the processing pipeline (see Figure 1).

The translation to SQL relies on tables `window` and `window_range` (Listing 6). These are based on the stream specification(s) in the query. Given start (`$start$`) and end (`$end$`) points for accessing temporal data as well as window size (`$window_size$`) and window slide (`$slide$`), a representation for all possible windows (with specific time points for the window range) together with all indexes for states that are built for the window by the specified sequencing method specified are computed. For standard sequencing, `window` and `window_range` are generated using PostgreSQL functions such as `generate_series`.

3.2 Transformations for Continuous STARQL Querying

The transformation above applies to temporal queries on historical data stored in a RDBMS. So, the window table generation as part of the transformation above is a one-step generation. In order to cope with streaming data, the transformation process has slightly to be adapted. The window table now is assumed to be incrementally updated by some function. Apart from that, the same transformation as for temporal queries can be applied to realize continuous querying with STARQL. In fact, the second implementation of the transformation that we tested does not materialize the whole window table, and so it can be directly adapted to DBs with dynamically updated entries. Similar ideas for continuos processing are used for TelegraphCQ [10], an DSMS built on top of PostgreSQL.

```
CREATE VIEW val AS
SELECT rel1.WID, rel2.SID, rel2.VALUE, rel1.INDEX
FROM window rel1, measurement rel2
WHERE rel2.timestamp = rel1.timestamp;

CREATE VIEW sensor AS
SELECT rel1.WID, rel1.SID
FROM val rel1;

CREATE VIEW q3 AS
SELECT rel1.WID, rel1.SID AS S
FROM HASVAL rel1, HASVAL rel2
WHERE rel2.WID = rel1.WID AND rel2.SID = rel1.SID AND
    rel1.INDEX < rel2.INDEX AND
    rel1.VALUE > rel2.VALUE;

CREATE VIEW q1 AS
SELECT rel1.WID, rel1.SID AS S
FROM sensor rel1
WHERE NOT EXISTS(SELECT *
                 FROM q3 rel2
                 WHERE rel2.WID = rel1.WID AND rel2.S = rel1.SID );

CREATE VIEW s_out AS
SELECT rel2.right, rel1.S AS SID
FROM q1 rel1, window_range rel2
WHERE rel1.WID = rel2.WID;
```

Listing 5: SQL transformation result

```
CREATE TABLE window_range AS
SELECT row_number() OVER (ORDER BY x.timestamp) - 1 AS wid,
       x.timestamp as left
       x.timestamp + $window_size$ as right
FROM (SELECT generate_series($start$, $end$, $slide$) AS timestamp) x ;

CREATE VIEW wid AS
SELECT DISTINCT r.wid, mp.timestamp
FROM measurement mp, window_range r
WHERE mp.timestamp BETWEEN r.left AND r.right ;

CREATE TABLE window AS
SELECT wid, rank() OVER (PARTITION BY wid ORDER BY timestamp ASC) as ind,
    timestamp
FROM wid ;
```

Listing 6: Window (range) tables

4 Evaluation

The system is evaluated along two example STARQL queries for reactive diagnosis in the Siemens use case. They representatives for queries expected to demand processing times that are quadratic (Query1) and linear (Query2). The engine transforms the queries w.r.t. predefined mappings into PostgreSQL queries.

The datasets that we use and describe in the following are part of the Siemens use case [23] in Optique. The original data processed/produced by Siemens appliances are sensor measurements, event data, operation logs, and other data stored in tables. These data are confidential, so Siemens provided a small public

Dataset	Total Measured Values	Timespan	Sensors	Measurements per Day/Sensor	Total Data Size
Ds1	82080	3 Days	19	1440	5 MB
Ds2	210,000	1506 Days	3	46.5	10 MB
Ds3	515,845,000	1824 Days	204	1386	23,000 MB

Table 1: The three used data sets

dataset and two larger anonymized datasets for use inside Optique. The public dataset (approximately 100 MB) has a simplified structure. For the evaluation we used three datasets: Ds1, Ds2 and Ds3. Ds1 contains public data and has a size of approximately 5MB. Ds2 contains anonymous data and has a size of approximately 10 MB, and Ds3 contains anonymous data with size 23 GB.

The (simplified) schema of the normalized tables is as follows:

```
CREATE TABLE measurement (timestamp,sensor,value);
CREATE TABLE sensor (id,assemblypart,name)
```

Measurements are represented with a table `measurement` and consist of a timestamp, a reference to the sensor, and a value. For our evaluation we are using one dataset of about 82,000 entries with a timestamp ranging over 3 days (Ds1), another dataset with about 210,000 entries with a timestamp ranging over 5 years (Ds2), and a dataset (Ds3) with more than 500 million entries over 5 years. All datasets contain data referring to a number of sensors.

The datasets differ in the total number of recorded values and also in the number of measured values per timeframe. In Ds1 and Ds3 a value is measured every minute. In Ds2 a value is measured only every 30 minute in average. So we expect the calculation of a single time window for Ds1 and Ds3 to be more difficult compared to the calculation for Ds2.

Query1 (Listing 7) builds, within each window, a sequence of all sensor values in the last 24 hours and checks whether one sensor increased monotonically. This query is expected to run quadratically slower as window size increases, due to the comparisons of all value pairs (?x, ?y) for all pairs of states (i,j).

Query2 (Listing 8) outputs, every minute, sensors that show a value higher than 90. This query is expected to be faster because of its simple window content with at most one timestamp. Both queries can be transformed to PostgreSQL. We show only the transformation for Query1 in Listing 9.

The tests were run in a VM on a system with an i7 2.8 GHz CPU and 16 GB of ram. Mean values of several test runs with cold cache are shown in the following tables. For our tests we used two approaches corresponding to combined rewriting and classical rewriting, respectively (cf. Sect. 3), and for each of these Query1 and Query2. In the first approach we pre-calculated all time windows in one large table, where every window has a window id, evaluated both queries once and a second time with additional window index structures added to the table.

```
CREATE STREAM S_out1 AS
SELECT { ?sensor rdf:type :RecentMonInc }<NOW>
FROM burner_regulator [ NOW - 24 hours, NOW ]-> 24 hours
SEQUENCE BY StdSeq AS seq
HAVING FORALL i, j IN seq, ?x,?y
IF {?sensor :hasVal ?x}<i> AND { :Regulator :hasVal ?y }<j> AND i < j
THEN ?x <= ?y ELSE TRUE
```

Listing 7: STARQL query Query1

```
CREATE STREAM S_out2 AS
SELECT { ?sens rdf:type :tooHigh }<NOW>
FROM burner_3  [ NOW , NOW ]-> 1 minute
SEQUENCE BY StdSeq AS seq
HAVING FORALL i IN seq, ?x  IF  { ?sens :hasVal ?x }<i> THEN ?x > 90
```

Listing 8: STARQL query Query2

```
CREATE OR REPLACE VIEW window_range AS
SELECT row_number() OVER (ORDER BY x.timestamp) - 1 AS wid,
       x.timestamp as left, x.timestamp + '1 hour'::interval as right
FROM  (SELECT generate_series(MIN(mp.timestamp),
             MAX(mp.timestamp), '1 hour'::interval)  AS timestamp FROM
                  measurement mp) x;

CREATE OR REPLACE VIEW wid AS
SELECT distinct wid, timestamp
FROM measurement mp, window_range w
WHERE mp.timestamp >= w.left and mp.timestamp < w.right;

CREATE VIEW win AS
SELECT wid, rank() OVER (PARTITION BY wid ORDER BY timestamp ASC) as ind,
       timestamp FROM wid;

CREATE VIEW val AS
SELECT DISTINCT rel1.WID , rel2.SID , rel2.VALUE , rel1.ind
FROM win rel1 , measurement rel2
WHERE rel2.timestamp = rel1.timestamp
ORDER BY wid, ind;

CREATE VIEW sensors AS SELECT rel1.WID , rel1.SID FROM val rel1;

CREAT VIEW q3 AS
SELECT rel1.WID , rel1.SID AS S
FROM val rel1 , val rel2
WHERE rel2.WID = rel1.WID AND rel2.SID = rel1.SID AND
    rel1.ind < rel2.ind AND
    rel1.VALUE > rel2.VALUE;

CREATE VIEW q1 AS
SELECT rel1.WID , rel1.SID AS S
FROM sensors rel1
WHERE NOT EXISTS(SELECT *
                 FROM q3 rel2
                 WHERE rel2.WID = rel1.WID AND rel2.S = rel1.SID);

CREATE VIEW s_out AS
SELECT rel2.right, rel1.S AS SID
FROM q1 rel1, window_range rel2
WHERE rel1.WID = rel2.WID;

SELECT DISTINCT s.right as timestamp, s.SID as Subject, 'rdf:type' as
     Predicate,    ':recentMonInc' from s_out s;
```

Listing 9: Query1 in PostgreSQL

	Query1						Query2		
	1 hour window			1 day window			1 minute window		
	precalc	no index	indexed	precalc	no index	indexed	precalc	No Index	Indexed
Ds1	1.5s	2s	1s	2s	30s	26s	2s	2s	2s
Ds2	5s	4s	2.8s	5s	17s	8s	20s	8s	7s
Ds3	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

Table 2: Times for precalculated window tables

	Query1		Query2
	1 hour window	1 day window	1 minute window
Ds1	2s	45s	12s
Ds2	40s	26s	1500s
Ds3	3942s	92637s	7123s

Table 3: Query times for dynamic stream generated windows

Results for the first approach are shown in Table 2. For every query you see a column with times in seconds for the non-indexed and indexed table. The non-indexed times consist of generating the table and evaluating the query. In the indexed version we added a B-tree index. The precalculation column shows the additional time for generating the window table, which has to be added in every case to the next two evaluation columns for non-indexed and indexed data.

There are different trade offs for both queries. For Query1 the total evaluation time increases as the window size increases, the precalculation time stays the same. The evaluation time for Ds1 increases faster as it has more measured values per window, compared to Ds2, which only has about one measured value per 30 minutes. Comparing the time for indexed and non-indexed data, one sees that with progressing time more single windows are used per query. On Ds1 we use only three windows and can decrease time from 30 to 26 seconds. On Ds2 we have about 1800 windows and can decrease the time from 17 to 8 seconds, which is more than 50 percent. Query2 produces a lot of small windows, so the evaluation time for each window is very short. The precalculation time for Ds2 increases a lot as we have about 2 million potential single windows in the window table. With a timeframe of only three days, the precalculation time for Ds1 stays small. The index has nearly no influence for Query2 as each window has only one tuple entry and all windows are evaluated once.

Ds3 could not be evaluated with the precalculation step, as it requires more than 50 GB additional memory for the window table. Therefore, we implemented a second approach by additional pl/pgsql functions. The idea was to generate each window dynamically, evaluate it, and delete the memory afterwards.

Results for the second approach are shown in Table 3. As each window is generated dynamically, there is no precalculation. On the other hand, no indexes can be added to a materialized table. The main disadvantage is that windows without values can not be filtered out in advance. As there are potentially 2 million windows for Query2 on Ds2, the system tries also to generate the empty

windows, which increases the time a lot. Nevertheless, the problem of additional required space is solved and also the complete 20GB of Ds3 can be evaluated.

5 Related Work

Much of the relevant work on stream processing has been done in the context of DSMS [3,10,13,16].

First steps towards streamified OBDA are stream extensions of SPARQL with a window operator having a multi-bag semantics where timestamps are dropped [12,6,22]. This does not interfere with the potential inconsistency of functional constraints on sensor values mentioned in the introduction, as these approaches handle timestamps by reification, for example, talking about measurements. Reification may lead to bulky representations of simple facts and hinders expressing simple functionality constraints (as mentioned above) in OWL.

The temporal OBDA approach of [5] uses an LTL based language with embedded CQs and not a sorted FOL language. For engineering applications with information needs as in the monotonicity example LTL is not sufficient, as it does not provide exists quantifier on top of the embedded CQs.

Though not directly related to OBDA, other relevant work stems from the field of *complex event processing*. For example, EP-SPARQL/ETALIS [2] uses also a sequencing constructor; and T-REX with the event specification language TESLA [11] uses an FOL language for identifying patterns.

6 Conclusion and Outlook

The main objective in designing a query language that is intended to be used in industry is to find the right balance between expressibility and feasibility. OBDA goes for weak expressibility and high feasibility by choosing rewriting and unfolding as methodology for query answering. But even in OBDA, feasibility is not a feature one gets for free; rather it has to be achieved with optimizations. So, for engines that are implemented according to the OBDA paradigm one has to show that such transformations are theoretically possible and feasible.

In this paper we argued that STARQL provides an adequate solution for streamified and temporalized OBDA scenarios as that of Siemens because: 1. It offers a semantics that allows a unified treatment of querying temporal and streaming data. 2. It combines high expressivity with safeness to guarantee a smooth transformation into standard domain independent backend queries such as SQL. 3. It can be implemented in an engine that implements the transformations s.t. acceptable query execution times are achievable, if run with the optimization mentioned here.

Future work contains, amongst others, the following: 1. extensive (scalability) tests with known benchmarks for stream processing, 2. generalization of the transformation to multiple streams where the slides parameters are not equal to the pulse parameter and where non-standard-sequencing strategies are used, and 3. extensive comparison with other approaches, in particular CEP approaches.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
2. Anicic, D., Rudolph, S., Fodor, P., Stojanovic, N.: Stream reasoning and complex event processing in ETALIS. Semantic Web 3(4), 397–407 (2012)
3. Arasu, A., Babu, S., Widom, J.: The CQL continuous query language: semantic foundations and query execution. The VLDB Journal 15, 121–142 (2006)
4. Artale, A., Kontchakov, R., Wolter, F., Zakharyaschev, M.: Temporal description logic for ontology-based data access. In: Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence. pp. 711–717. IJCAI'13 (2013)
5. Borgwardt, S., Lippmann, M., Thost, V.: Temporal query answering in the description logic DL-Lite. In: Fontaine, P., Ringeissen, C., Schmidt, R.A. (eds.) Frontiers of Combining Systems. LNCS, vol. 8152, pp. 165–180. Springer (2013)
6. Calbimonte, J.P., Jeung, H., Corcho, O., Aberer, K.: Enabling query technologies for the semantic sensor web. Int. J. Semant. Web Inf. Syst. 8(1), 43–63 (Jan 2012)
7. Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., Poggi, A., Rodríguez-Muro, M., Rosati, R.: Ontologies and databases: The DL-Lite approach. In: Tessaris, S., Franconi, E. (eds.) Semantic Technologies for Informations Systems (RW 2009), LNCS, vol. 5689, pp. 255–356. Springer (2009)
8. Chakravarthy, U.S., Grant, J., Minker, J.: Foundations of semantic query optimization for deductive databases. In: Foundations of deductive databases and logic programming, pp. 243–273. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1988)
9. Chakravarthy, U.S., Grant, J., Minker, J.: Logic-based approach to semantic query optimization. ACM Trans. Database Syst. 15(2), 162–207 (1990)
10. Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M.J., Hellerstein, J.M., Hong, W., Krishnamurthy, S., Madden, S., Raman, V., Reiss, F., Shah, M.A.: TelegraphCQ: Continuous dataflow processing for an uncertain world. In: CIDR (2003)
11. Cugola, G., Margara, A.: TESLA: A formally defined event specification language. In: Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems. pp. 50–61. DEBS '10, ACM, New York, NY, USA (2010)
12. Della Valle, E., Ceri, S., Barbieri, D., Braga, D., Campi, A.: A first step towards stream reasoning. In: Domingue, J., Fensel, D., Traverso, P. (eds.) Future Internet – FIS 2008, LNCS, vol. 5468, pp. 72–81. Springer Berlin / Heidelberg (2009)
13. Hwang, J.H., Xing, Y., Çetintemel, U., Zdonik, S.B.: A cooperative, self-configuring high-availability solution for stream processing. In: ICDE. pp. 176–185 (2007)
14. Kharlamov, E., Solomakhina, N., Özcep, Ö.L., Zheleznyakov, D., Hubauer, T., Lamparter, S., Roshchin, M., Soylu, A.: How semantic technologies can enhance data access at siemens energy. In: Proceedings of the 13th International Semantic Web Conference (ISWC 2014) (2014)
15. Kontchakov, R., Lutz, C., Toman, D., Wolter, F., Zakharyaschev, M.: The combined approach to ontology-based data access. In: Walsh, T. (ed.) IJCAI. pp. 2656–2661. IJCAI/AAAI (2011)
16. Krämer, J., Seeger, B.: Semantics and implementation of continuous sliding window queries over data streams. ACM Trans. Database Syst. 34(1), 1–49 (Apr 2009)
17. Lutz, C., Toman, D., Wolter, F.: Conjunctive query answering in the description logic $\mathcal{EL}$ using a relational database system. In: Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI-09). AAAI Press (2009)

18. Lutz, C., Toman, D., Wolter, F.: Conjunctive query answering in $\mathcal{EL}$ using a database system. In: In Proceeding of the 5th International Workshop on OWL: Experiences and Directions (OWLED 2008) (2008)
19. Özçep, Ö. L.., Möller, R.: Ontology based data access on temporal and streaming data. In: Koubarakis, M., Stamou, G., Stoilos, G., Horrocks, I., Kolaitis, P., Lausen, G., Weikum, G. (eds.) Reasoning Web. Reasoning and the Web in the Big Data Era. Lecture Notes in Computer Science, vol. 8714. (2014)
20. Özçep, Ö.L., Möller, R., Neuenstadt, C., Zheleznyakov, D., Kharlamov, E.: Deliverable D5.1 – A semantics for temporal and stream-based query answering in an OBDA context. Deliverable FP7-318338, EU (October 2013)
21. Özçep, Özgür.L., Möller, R., Neuenstadt, C.: A stream-temporal query language for ontology based data access. In: KI 2014. LNCS, vol. 8736, pp. 183–194. Springer International Publishing Switzerland (2014)
22. Phuoc, D.L., Dao-Tran, M., Parreira, J.X., Hauswirth, M.: A native and adaptive approach for unified processing of linked streams and linked data. In: International Semantic Web Conference (1). pp. 370–388 (2011)
23. Schlatte, R., Möller, R., Giese, C.N.M.: Deliverable D1.1 – joint phase 1 evaluation and phase 2 requirement analysis. Deliverable FP7-318338, EU (October 2013), publicly available at `http://www.optique-project.eu/results/public-deliverables/`
24. Tsangaris, M.M., Kakaletris, G., Kllapi, H., Papanikos, G., Pentaris, F., Polydoras, P., Sitaridi, E., Stoumpos, V., Ioannidis, Y.E.: Dataflow processing and optimization on grid and cloud infrastructures. IEEE Data Eng. Bull. 32(1), 67–74 (2009)