On Bounded-Memory Stream Data Processing with Description Logics

Özgür L. Özçep and Ralf Möller

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

Abstract. Various research groups of the description logic community, in particular the group of Franz Baader, have been involved in recent efforts on temporalizing or streamifying ontology-mediated query answering (OMQA). As a result, various temporal and streamified extensions of query languages for description logics with different expressivity were investigated. For practically useful implementations of OMQA systems over temporal and streaming data, efficient algorithms for answering continuous queries are indispensable. But, depending on the expressivity of the query and ontology language, finding an efficient algorithm may not always be possible. Hence, the aim should be to provide criteria for easily checking whether an efficient algorithm exists at all and, possibly, to describe such an algorithm for a given query. In particular, for stream data it is important to find simple criteria that help deciding whether a given OMQA query can be answered with sub-linear space w.r.t. the length of a growing stream prefix. An important special case dealt with under the term "bounded memory" is that of testing for constant space. This paper discusses known syntactical criteria for bounded-memory processing of SQL queries over relational data streams and describes how these criteria from the database community can be lifted to criteria of bounded-memory query answering in the streamified OMQA setting. For illustration purposes, a syntactic criterion for bounded-memory processing of queries formulated in a fragment of the stream-temporal query language STARQL is given.

Keywords: streams, bounded memory, ontology-mediated query answering, ontology-based data access

1 Introduction

Ontology-mediated query answering (OMQA) [8] is a paradigm for accessing data via declarative queries whose intended sets of answers is constrained by an ontology. Usually, the ontology is represented in a formal logic such as a description logic (DL). Though OMQA has been of interest both for researchers as well as users from industry, a real benefit for the latter heavily depends on the

possibility to handle temporal and streaming data. So, various research groups of the DL community, in particular the group of Franz Baader (see, e.g., [6,10,27]), have been involved in recent efforts on temporalizing or streamifying classical OMQA. As a result, various temporal and streamified extensions of query languages for description logics with different expressivity were investigated.

For practically useful implementations of OMQA systems over temporal and streaming data, efficient algorithms for answering continuous queries are indispensable. But, depending on the expressivity of the query and ontology languages, finding an efficient algorithm may not always be possible. Hence, the aim should be to provide criteria for easily checking whether an efficient algorithm exists at all and, possibly, to describe such an algorithm for a given query. For those queries that are provably bounded-memory computable, one knows that there exists an algorithm using only constant space [3]. That means, if one had a (preferably simple) criterion for testing whether a given query is bounded-memory computable and, moreover, if one had a constructive procedure to generate a memory-bounded algorithm producing exactly the answers of the original query (over all streams), then one would make a considerably big step towards performant stream processing.

A special sub-scenario for performant query answering over streams is to provide simple criteria that help deciding whether a given OMQA query can be answered with sub-linear space w.r.t. the length of the growing stream prefix. An even more special (but important) case dealt with under the term "bounded memory" [13] is that of testing for computability in constant space. We note that, in particular in research on low-level data stream processing for sensor networks [1], there is an equal interest in considering other sub-linear space constraints such as (poly)logarithmic space. In this paper, we focus on constant space requirements, however.

Usually, when considering bounded-memory computability one is interested in what we call here *bounded-memory computability w.r.t. the input*: It denotes the constraint that at most constant space (in the length of the input) is required to store the relevant information of the ever growing input stream(s). Following the approach of [3] we are also going to consider what we call *bounded-memory computability w.r.t. the output*: This notion denotes the constraint of constant space required to memorize the required information of the output produced so far in order to compute new output correctly. This kind of constraint is required in particular for a processing model where the output in each time point consists only of the delta w.r.t. the output written in earlier time points. Such an output model is implemented, e.g., with the so-called IStream operator ("I" for "inserted") in the relational stream query language CQL [4].

As bounded-memory computability is motivated by implementability and performance, it has a flavour of a low-level issue that should be handled when considering the implementation of an OMQA system. However, it would be an asset to have criteria for bounded-memory computability at the ontology level, i.e., to have criteria deciding whether a given query w.r.t. a stream of abox assertions, an ontology and, possibly, integrity constraints can be computed in

constant space w.r.t. the length of the stream of abox axioms. The reason is that ontology axioms or integrity constraints may have effects on bounded-memory computability, either positively or negatively. For example, if the ontology allows to formulate rigidity assumptions, then bounded-memory computability may not hold anymore [12]. On the other hand, functional integrity constraints over the whole stream may lead to a bound on an otherwise unbounded set of possible values, thereby ensuring bounded-memory computability.

Of course, if one considers ontology-mediated query answering in the strict sense, namely so-called ontology-based data access (OBDA), there is an obvious alternative approach for testing bounded-memory computability. In OBDA, a query is rewritten w.r.t. the tbox and then unfolded w.r.t. some mappings into an SQL query, so that answers to the original query can be calculated by answering a streamified SQL query over a backend data stream management system. For streamified OBDA this means that one can reduce the bounded-memory test of a query on the ontology level to a bounded-memory test of the transformed query over the backend data stream and then use the known bounded-memory criteria for queries on relational data streams. In this paper, we do not deal with the aspects of mapping and unfolding. Instead we consider how to lift the known criteria [3] for relational stream queries to ontological queries. For this we consider the case of lightweight description logics as representation languages for ontologies so that perfect rewriting of queries according to the OBDA paradigm is possible.

Quite a common scenario of stream processing (in particular for model checking of infinite words [7]) is that the queries on a stream have to be answered over the whole growing prefixes of a stream. Using the window metaphor, this corresponds to applying a window whose right end slides whereas its left end is set to a constant, i.e., to a fixed time point. Of course, the question arises whether the problem of ensuring bounded-memory computability is not solved by using a *finite* sliding window over the data stream. If one considers only row-based windows, i.e., windows where the width-parameter denotes the number of elements that make up its content (see [4]), then bounded-memory computability is always guaranteed by definition of the semantics of row-based windows. But sometimes one cannot easily decide on the appropriate width of the window that is required to capture relevant information on the prefixes of the streams. And even if it would be possible, the necessary size of the window could still be too big so that in optimizing algorithms one could benefit from the use of less memory.

For example, a naively implemented query that requires a quadratic number of comparisons such as a query asking for the monotonic increase of temperature value sensors, may be implementable more efficiently with a data structure storing a state with relevant data that are updated during stream processing. In general, any optimized algorithm would have to rely on some appropriate state data structure. The data structure for states we are going to consider stores values in registers and allows manipulating them with basic arithmetical operators. In low-level stream processing scenarios, where the queries (such as top-k)

are required to be answered only approximately, the state data structures are called *sketches*, *summaries* or *synopses*, as these data structures really give some approximate summary of the stream prefixes [14,1].

In this paper we discuss known syntactical criteria for bounded-memory processing of SQL queries over relational data streams [3] and describe how and to what extent these criteria from the database community can be lifted to criteria of bounded-memory query answering in the streamified OMQA setting. For illustration purposes we consider a syntactic criterion of bounded-memory computability applied to a fragment of STARQL [21,23,24,19,18] which was developed as a general query framework for accessing temporal and streaming data in the OMQA paradigm.

2 The STARQL Framework

STARQL is a stream query language framework for OMQA scenarios with temporal and streaming data. As such, it is part of recent efforts of streamifying and temporalizing OMQA [6,5,9,15,11,26,23,28,10] with, amongst others, contributions by Franz Baader and members of his group. We are referring to STARQL as a framework, because it describes a whole class of query languages which differ regarding the expressivity of the DL used for the tbox and regarding the embedded query languages used to query the individual intra-window aboxes constructed in the sequencing operation (see below).

2.1 Example

The following example for an information need in an agent scenario illustrates the main constructors of STARQL. A rational agent has different sensors, in particular different temperatures attached to different components. The agent receives both, high-level messages and low-level measurement messages, from a single input stream `Sin`. The agent has stored in a tbox some background knowledge on the sensors. In particular, the tbox contains an axiom stating that all temperature sensors are sensors and that all type-X temperature sensors are temperature sensors. Factual knowledge on the sensors is stored in a (static) abox. For example, the abox may contain assertions *type-X-temperature-Sensor(tcc125)*, *attachedTo(tcc125,c1)*, *locatedAt(c1,rear)* stating that there is a temperature sensor of type X named *tcc*125 that is attached to some component $c1$ at the rear. There is no explicit statement that *tcc*125 is a temperature sensor, this can be derived only with the axioms of the tbox.

The agent has to recognize whether the sensed temperature is critical. Due to some heuristics, a critical state is identified with the following pattern: In the last 5 minutes there was a monotonic increase on some interval followed by an alert message. As we assume that temperature values have been pre-processed via a smoothing operation, monotonic increase is not prevented from appearing quite often. The agent is expected to output every 10 seconds all temperature sensors showing this pattern and to mark them as critical. A STARQL formalization of

the information need the agent is going to satisfy is given in the listing of Figure 1.

```
1  CREATE STREAM Sout AS
2  CONSTRUCT  GRAPH NOW { ?s a :inCriticalState }
3  FROM   Sin[NOW-5min, NOW]->10s
4         <http://www.ifis.uni-luebeck.de/abox>
5         <http://www.ifis.uni-luebeck.de/tbox>
6  USING PULSE AS START = 0s, FREQUENCY = 10s
7  WHERE { ?s a :TempSens }
8  SEQUENCE BY StdSeq
9  HAVING
10 EXISTS i1, i2, i3:
11  0 < i1 AND i2 < MAX AND plus(i2,1,i3) AND i1 < i2
12  GRAPH i3 { ?s :message ?m  . ?m a :AlertMessage } AND
13  FORALL i, j, ?x,?y:
14    IF  i1 <= i  AND i <  j  AND j <= i2  AND
15        GRAPH i { ?s :val ?x }  AND GRAPH j { ?s :val ?y }
16    THEN ?x <= ?y
```

Fig. 1: Example STARQL query

The `CONSTRUCT` operator (line 2) fixes the format of the output stream. Here, as well as in the `HAVING` clause (see below), STARQL uses the named-graph notation of the W3C recommended RDF[1] query language SPARQL[2] for specifying a basic graph pattern (BGP) and attaching a time expression. The output stream contains expressions of the form

```
GRAPH NOW { ?s a :inCriticalState }
```

where `NOW` is instantiated by time points and `?s` by constants fulfilling the required conditions as specified in the following lines of the query. The evolvement of the time `NOW` is specified in the pulse declaration (line 6).

The resources to which the query refers are specified using the keyword `FROM` (line 3–5). Following this keyword one can refer to one or more streams (by names or further stream expressions) and to URIs to a tbox and an abox, which are understood as static knowledge bases. In this example, only one stream is referenced, the stream named S_{in}. In this case, the stream consists, first, of timestamped triples matching the BGPs of the form

```
GRAPH t1 { ?s :val ?y }
```

[1] `https://www.w3.org/RDF/`
[2] `https://www.w3.org/TR/rdf-sparql-query/`

stating that `?s` has value `?y` at time `t1`. In logical notation, these subgraphs would be written as timestamped abox assertions of the form *val(?s,?y)*$\langle t1 \rangle$. Secondly, the input stream may contain timestamped triples matching BGPs of the form

```
GRAPH t2 { ?m a :AlertMessage }
```

stating that at time point `t2` an alert message arrived. In DL-notation this would be expressed as: *AlertMessage(?m)*$\langle t2 \rangle$. The window operator attached to the input stream, `[NOW-5min, NOW]->10s`, is meant to give snapshots of the stream with the slide of 10s (update frequency) and range of 5 minutes (all stream elements within last 5 minutes).

For both types of BGPs the number of possible triples in a stream are unbounded: in the first case this is due to the attribute `val` with its range being real numbers, an infinite (even dense and continuous) domain to represent possible measurement values. In the second case, this is due to the possibly infinite number of messages that are generated. We think of messages being produced by a controller that generates message IDs from a (discrete but still) infinite domain.

The `WHERE` clause (line 7) specifies the sensors `?s` that are relevant for the information need, namely temperature sensors. Already here it becomes clear that the agent has to incorporate his background knowledge from the tbox: in order to get all temperature sensors `?s` it also has to find all type-X sensors. The `WHERE` clause is evaluated only against the static abox. The stream-temporal conditions are specified in `HAVING` clause.

For every binding of `?s`, the query evaluates conditions that are specified in the `HAVING` clause (lines 9–16). A sequencing method (here `StdSeq`) maps an input stream to a sequence of aboxes, annotated by states[3] i, j, according to a grouping criterion. Note that the index variables for states $i1, i2$ are not prefixed by a question mark, as is done for the other variables. This is to indicate the different types of variables. Index variables need to be bound by a quantifier, they are not allowed as answer variables. The built-in sequencing method `StdSeq` is called *standard sequencing*. It puts all stream elements with the same timestamp into the same mini abox. Note that abox sequencing gives the user the flexibility of defining its own abox sequence—whereas most of the other approaches of temporalized and streamified OMQA, such as [6] already presuppose a sequence of aboxes. This flexibility, on the other hand, means a burden for classical OBDA where queries have to be transformed to queries over the backend. But fortunately for simple sequencing strategies (and possibly for others) such as standard sequencing one can get rid of the additional sequencing layer by reducing the state indexes to the timestamps of the triples in the stream (see [24]).

[3] Note that we prefer to use the term "state" instead of the temporally connotated "stage", because we allow in principle sequencing methods that are not temporal, e.g., sequencing by clustering.

Testing for conditions at a state is done with the SPARQL sub-graph mechanism. So, e.g.,

```
GRAPH i3 {?s :message ?m . ?m a :AlertMessage}  (line 12)
```

asks whether `?s` showed an alert message at a state annotated by the variable `i3`. State `i3` is further determined as the successor of the end state `i2` in the interval `[i1, i2]` (line 11). Over the interval `[i1, i2]` the usual monotonicity condition (`FORALL` condition, lines 13–15) is expressed using a first-order logic pattern. Note that a naive implementation of this condition would store all values received so and make a quadratic number of comparisons over them.

As in the case of the `WHERE` clause, also for the evaluation of the `HAVING` clause the background knowledge (static tbox and static abox) must be incorporated in order to guarantee a complete set of answers. For example, the tbox may contain a taxonomy of different types of messages, in particular different sub-types of alert messages. If only instances of these subtypes are mentioned in the abox, then their super-types have to be inferred by the agent.

2.2 Syntax

The example in the previous subsection illustrated the syntax and the intended semantics of STARQL. For the sake of completeness we recapitulate here the grammar that captures the syntax of STARQL. This grammar leads to a subfragment of the original STARQL language [22]. In particular, the `HAVING` clauses are less expressive than the original ones. Further we leave out aggregation constructors and macro definitions. For a full description see [22,23]. For the full STARQL language, the bounded-memory results of this paper may not hold anymore.

The grammar (Figure 2) is denoted STARQL (OL,ECL) and it contains parameters that have to be specified in its instantiations: the ontology language OL and the embedded condition language ECL. OL constrains the languages of the aboxes and the tboxes that are referred to in the grammar (underlined in Figure 2). ECL is a query language referring to the signature of the ontology language. STARQL uses ECL conditions in its `WHERE` and `HAVING` clauses. The adequate instantiation of STARQL(OL, ECL) may vary depending on the requirements of the use case.

We are not going to discuss the whole grammar but only make some comments on the most interesting part, which is the set of rules for the specification of `HAVING` clauses (abbreviated hCl in the grammar). In the full STARQL grammar (see [23]) `HAVING` clauses are allowed to use arbitrary first-order logic constructors, in particular all boolean connectors, and also exists- as well as forall-quantifiers. As STARQL allows infinite domains (such as the real numbers in order to specify, say, temperature values) queries using FOL constructors have to be used with care in order to give safe queries, i.e., queries that output only finite sets of bindings. A query such as $\phi(y) = \neg val(tcc125, y)$ for example is not safe as it would require outputting all of the infinitely many ys not being values of *tcc125*.

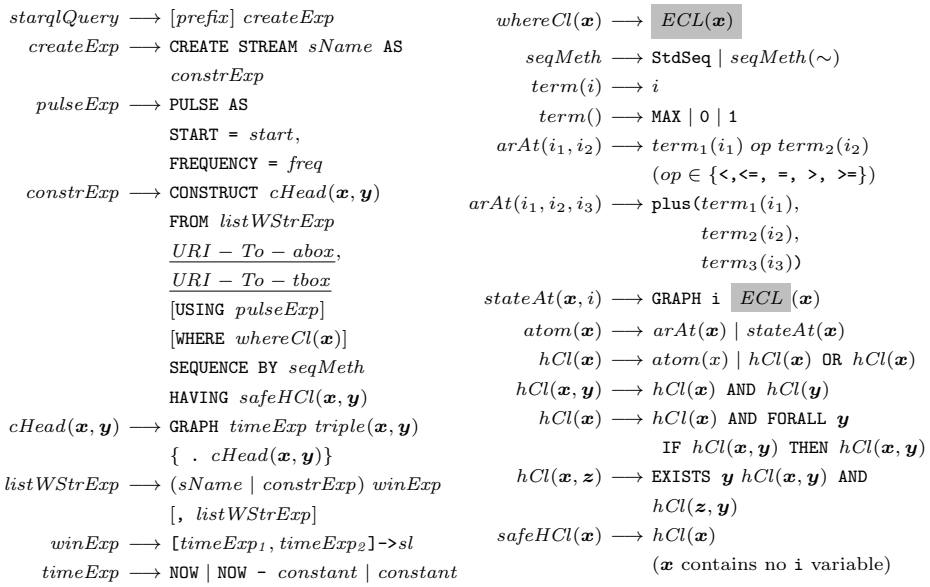

Fig. 2: Syntax for STARQL(OL, ECL) template.

This problem is known since the beginning of classical DB theory and it has been handled by describing syntactical rules guaranteeing safeness. A similar approach for handling safeness, but relying on adornments, is described in [23]. The grammar presented here has no adornments but still reflects safety conditions. For example, the boolean connector for disjunction (`or`) is allowed to be applied only for disjunctions with the same set of open variables. Furthermore, the existential and the forall quantifiers are allowed to quantify only over variables which are guarded. Hence, an exists quantifier over x is allowed only if x is bounded by a safe *hCL* clause appearing as conjunction in the scope of the exists quantifier. And universally bounded variables are allowed only if they are guarded in with the antecedent of an implication in the scope of the for-all quantifier.

We further note that the grammar allows also unbounded windows, that is, windows of the form `[`*constant*`,NOW]` where the left interval point is fixed, so that the window content is going to contain the whole prefixes beginning with the start time point of the query (set to *constant*).

2.3 Semantics

The explication of the semantics for STARQL queries rests on the semantics of the instantiations of the parameter values OL and ECL. The only presumption we make is that the OL and ECL have to fulfill the following condition: There must be a notion of a certain answer of an ECL w.r.t. an ontology. The

motivation for such a layered—or as we call it here: separated—definition of the semantics is a strict separation of the semantics provided by the embedded condition languages ECL and the semantics for the stream query language on top of it. Hence the separated semantics has a plug-in-flavor, allowing users to embed any preferred ECL without repeatedly redefining the semantics of the whole query language.

For ease of exposition we assume that the query specifies only one output sub-graph pattern and that there is exactly one static abox $\mathcal{A}_{st}$ and one tbox $\mathcal{T}$. Similar to the approach of [9], the tbox is assumed to be non-temporal in the sense that there are no special temporal or stream constructors. We give a denotational specification $[\![S_{out}]\!]$ of S_{out} recursively by defining the denotations of the components. We will refer to the notion of a temporal abox within this denotation semantics and also later on. A *temporal abox* or *intra-window abox* is a finite set of timestamped abox axioms $ax\langle t\rangle$, with $t \in T$. We call structures of the form $\langle(\mathcal{A}_i)_{i\in[n]}, \mathcal{T}\rangle$ consisting of a finite sequence of aboxes and a pure tbox a *sequenced ontology (SO)*. The index i of the abox $\mathcal{A}_i$ is called its *state index*.

So assume that the following query template is given.

$$
\begin{aligned}
S_{out} = \texttt{CONSTRUCT GRAPH } & \mathit{timeExpCons}\ \Theta(\boldsymbol{x}, \boldsymbol{y}) \\
\texttt{FROM } & S_1\ \mathit{winExp}_1, \ldots, S_m\ \mathit{winExp}_m, \mathcal{A}_{st}, \mathcal{T} \\
\texttt{WHERE } & \psi(\boldsymbol{x})\ \texttt{SEQUENCE BY}\ \mathit{seqMeth}\ \texttt{HAVING}\ \phi(\boldsymbol{x}, \boldsymbol{y})
\end{aligned}
$$

Windowing Let $[\![S_i]\!]$ for $i \in [m]$ be the streams of timestamped abox assertions. The denotation of the windowed stream $ws_i = S_i\ \texttt{[}\mathit{timeExp}^i_1, \mathit{timeExp}^i_2\texttt{]->}sl_i$ is defined by specifying a function F^{winExp_i} s.t.: $[\![ws_i]\!] = F^{winExp_i}([\![S_i]\!])$.

$[\![ws_i]\!]$ is a stream with timestamps from the set $T' \subseteq T$, where $T' = (t_j)_{j\in\mathbb{N}}$ is fixed by the pulse declaration with t_0 being the starting time point of the pulse. The domain of the resulting stream consists of temporal aboxes.

Assume that $\lambda t.g^i_1(t) = [\![\mathit{timeExp}^i_1]\!]$ and $\lambda t.g^i_2(t) = [\![\mathit{timeExp}^i_2]\!]$ are the unary functions of time denoted by the time expressions in the window. For example, if $\mathit{timeExp}^i_2$ is `NOW - 5`, then the function g^i_2 is just the function $\lambda t.(t-5)$. We assume that for all t $[\![\mathit{timeExp}^i_1]\!](t) \leq [\![\mathit{timeExp}^i_w]\!](t)$, as otherwise the window would not denote a proper interval. We have to define for every t_j the temporal abox $\tilde{\mathcal{A}}^i_{t_j} \in [\![ws_i]\!]$. If $t_j < sl - 1$, then $\tilde{\mathcal{A}}^i_{t_j} = \emptyset$. Otherwise set first $t^i_{start} = \lfloor t_j/sl \rfloor \times sl$ and $t^i_{end} = max\{t_{start} - (g^i_2(t_j) - g^i_1(t_i)), 0\}$, and define on that basis $\tilde{\mathcal{A}}^i_{t_j} = \{ax\langle t\rangle \mid ax\langle t\rangle \in [\![S]\!]$ and $t^i_{end} \leq t \leq t^i_{start}\}$. Now, the denotations of all windowed streams are joined w.r.t. the timestamps in T': $js([\![ws_1]\!], \ldots, [\![ws_m]\!]) := \{\big(\bigcup_{i\in[m]} \tilde{\mathcal{A}}^i_t\big)\langle t\rangle \mid t \in T'$ and $\tilde{\mathcal{A}}^i_t\langle t\rangle \in [\![ws_i]\!]\}$.

Sequencing The stream $js([\![ws_1]\!], \ldots, [\![ws_m]\!])$ is processed according to the sequencing method specified in the query. The output stream has timestamps from T'. The stream domain now consists of finite sequences of pure aboxes.

The sequencing methods used in STARQL refer to an equivalence relation $\sim$ to specify which assertions go into the same intra-window abox. The relation

$\sim$ is required to respect the time ordering, i.e., it has to be a congruence over T. The equivalence classes are referred to as states and are denoted by variables i, j etc.

Let $\tilde{\mathcal{A}}_t\langle t\rangle$ be the temporal abox of $js([\![ws_1]\!], \ldots, [\![ws_m]\!])$ at t. Let $T'' = \{t_1, \ldots, t_l\}$ be the time points occurring in $\tilde{\mathcal{A}}_t$ and let k' be the number of equivalence classes generated by the time points in T''. Then define the sequence at t as $(\mathcal{A}_0, \ldots, \mathcal{A}_{k'})$ where for every $i \in [k']$ the abox $\mathcal{A}_i$ is $\mathcal{A}_i = \{ax\langle t'\rangle \mid ax\langle t'\rangle \in \tilde{\mathcal{A}}_t$ and t' in i^{th} equiv. class$\}$. The standard sequencing method `StdSeq` is just the one using the identity $=$ as equivalence relation. Let $F^{seqMeth}$ be the function realizing the sequencing.

WHERE Clause In the `WHERE` clause only $\mathcal{A}_{st}$ and $\mathcal{T}$ are relevant for the answers. So, purely static conditions (e.g. asking for sensor types as in the example above) are evaluated only on $\mathcal{A}_{st} \cup \mathcal{T}$. The result are bindings $\boldsymbol{a}_{wh} \in cert(\psi(\boldsymbol{x}), \langle \mathcal{A}_{st}, \mathcal{T}\rangle)$. This set of bindings is applied to the `HAVING` clause $\phi(\boldsymbol{x}, \boldsymbol{y})$.

HAVING Clause STARQL's semantics for the `HAVING` clauses relies on the certain-answer semantics of the embedded ECL conditions.

The semantics of $\phi(\boldsymbol{a}_{wh}, \boldsymbol{y})$, , i.e., the set of certain answers containing bindings for $\boldsymbol{y}$, is defined for every binding $\boldsymbol{a}_{wh}$ from the evaluation of the `WHERE` clause. The semantics depends on t. Assume that the sequence of aboxes at t is $seq = (\mathcal{A}_0, \ldots, \mathcal{A}_k)$. We define the set of *separation-based certain answers*, denoted: $cert_{sep}(\phi(\boldsymbol{a}_{wh}, \boldsymbol{y}), \langle \mathcal{A}_i \cup \mathcal{A}_{st}, \mathcal{T}\rangle)$.

If for any i the pure ontology $\langle \mathcal{A}_i \cup \mathcal{A}_{st}, \mathcal{T}\rangle$ is inconsistent, then we set $cert_{sep} =$ NIL, where NIL is a new constant not contained in the signature. In the other case, the bindings are defined as follows. For t one constructs a sorted first-order logic structure $\mathfrak{I}_t$: the domain of $\mathfrak{I}_t$ consists of the index set $\{0, \ldots, k\}$ as well as the set of all individual constants of the signature. For every state atom *stateAt* `GRAPH i` $ECL(\boldsymbol{z})$ in $\phi(\boldsymbol{a}_{wh}, \boldsymbol{y})$ with free variables $\boldsymbol{z}$ having length l, say, introduce an $(l+1)$-ary symbol R and replace `GRAPH i` $ECL(\boldsymbol{z})$ by $R(\boldsymbol{z}, i)$. The denotation of R in $\mathfrak{I}_t$ is then defined as the set of certain answers of the embedded condition $ECL(\boldsymbol{z})$ w.r.t. the i^{th} abox $\mathcal{A}_i$: $R^{\mathfrak{I}_t} = \{(\boldsymbol{b}, i) \mid \boldsymbol{b} \in cert(ECL(\boldsymbol{z}), \langle \mathcal{A}_i \cup \mathcal{A}_{st}, \mathcal{T}\rangle)\}$. Constants denote themselves in $\mathfrak{I}_t$. This fixes a structure $\mathfrak{I}_t$ with finite denotations of its relation symbols. The evaluation of the `HAVING` clause is then nothing more than evaluating the FOL formula (after substitutions) on the structure $\mathfrak{I}_t$.

Let $F^{\phi(\boldsymbol{a}_{wh}, \boldsymbol{y})}$ be the function that maps a stream of abox sequences to the set of bindings $(\boldsymbol{b}, t)$ where $\boldsymbol{b}$ is the binding for $\boldsymbol{y}$ in $\phi(\boldsymbol{a}_{wh}, \boldsymbol{y})$ at time point t.

Summing up, the following denotational decomposition results:

$$[\![S_{out}]\!] = \{\texttt{GRAPH } [\![timeExpCons]\!]\, \Theta(\boldsymbol{a_{wh}}, \boldsymbol{b}) \mid \boldsymbol{a_{wh}} \in cert(\psi(\boldsymbol{x}), \mathcal{A}_{st} \cup \mathcal{T}) \text{ and } (\boldsymbol{b}, t) \in F^{\phi(\boldsymbol{a_{wh}}, \boldsymbol{y})}\big(F^{seqMeth}(js(F^{winExp_1}([\![S_1]\!]), \ldots, F^{winExp_m}([\![S_m]\!])))\big)\}$$

Regarding the following considerations on bounded-memory processing we note two points: First, the output is controlled by the pulse. At each evolving

time point the whole set of elements with timestamps falling into the current time interval of the window is considered for the calculation of the output. This means that there may be more than one RDF tuple to be processed at each time point. We assume that at each time point the set of RDF tuples to be processed is bounded by a constant, otherwise the stream system could eventually fall behind the pulse. (Of course, it may also fall behind the pulse without the assumption on boundedness by a constant.) But even under this restrictive assumption, bounded-memory computability is an issue due to the non-bounded number of triples in the ever growing prefixes of the input streams. Hence a systematic consideration is in order.

Further we note that the semantics is defined such that at every time point the whole set of bindings that make the `WHERE` clause and the `HAVING` clause true is returned, and not the delta of new bindings. That is, the semantics of STARQL follows the idea of the RStream operator of the relational stream query language CQL [4] and not that of the IStream operator.

2.4 Properties of STARQL

Non-reified approach A relevant question from the representational point of view is how to represent events and, in particular, time in the query language. For STARQL, the decision was to use a non-reified approach, where time is handled as an annotation for sentences whose evaluation depends on the associated time. As illustrated by the agent example above, the abox assertions (RDF triples in SPARQL speak) are tagged with timestamps. This method is similar to adding an extra time argument for concept and roles as in [5]. The non-reified approach allows for representing time-dependent facts such as the fact that some sensor showed some value at a given time point. This time point is relevant for the window semantics in STARQL.

As the reified approach is more conservative and does not require to change the semantics (the time attribute is treated as an ordinary attribute), a natural question is why STARQL follows the non-reified strategy. The main reason is that time requires a special treatment as it has specific constraints for reasoning. For example, in the measurement scenario one would like to express the constraint that, at every time point, a sensor shows at most one value. This can be done with a classical DL-Lite axiom by stating $(func\ val) \in \mathcal{T}$. Note that under such a constraint it is necessary that the window semantics preserves the timestamps, as is indeed the case for the STARQL window semantics. Otherwise two timestamped stream elements of the form $val(tcc125, 92°)\langle 3s\rangle$ and of the form $val(tcc125, 95°)\langle 5s\rangle$ would lead to an inconsistency.

On the other hand, if one follows the reified approach such a time-dependent constraint is not expressible in a DL: One would have to formulate that there are no two measurements with the same associated sensor and same timestamp but different values. As DLs are concept oriented, they are not suited to expressing non-tree-shaped constraints with tbox axioms.

Homogeneous interface For the syntax and the semantics of STARQL queries the exact resource of the input stream is not relevant: It may be a stream of elements arriving in real-time via a TCP port, but equally it can be a simulated stream of data produced by reading out a text file or a temporal database. In the former case, one can speak of (genuine) stream querying, whereas in the latter case we use the term *history querying*. So STARQL offers the same interface to real-time queries (as required, for example, in monitoring scenarios) and history queries (as required, e.g., for reactive diagnostics). And, indeed such a homogenous interface to two different modes of querying has proved useful for real industrial use cases, in particular, for the turbine-diagnostics use case of SIEMENS in the context of the OPTIQUE project [20,17,19].

Separation between static and temporal conditions As illustrated in the example above, STARQL allows to separate the conditions expressed in an information need into conditions that concern only the static part of the background knowledge (tbox $\mathcal{T}$ and static abox $\mathcal{A}_{st}$) and into conditions which require both, the static part and the streams. The former can be queried in the `WHERE` clause, the latter in the `HAVING` clause. For the semantics of `HAVING` clauses we also incorporated the tbox and the static abox (which is always added to each abox in the sliding window). And indeed, this reference, at first sight, is not eliminable. The reference to the tbox can be eliminated by just rewriting the `HAVING` clause into a new `HAVING` clause using the standard perfect rewriting technique. Still, the theoretical question remains whether it is possible to push all references to the static abox (all occurrences of concept and role symbols that appear in the static abox) into the `WHERE` clause, so that the `HAVING` clause can be evaluated only on the streams and the bindings resulting from the evaluation of the `WHERE` clause. In other terms, is the `HAVING` clause separable in a pure static part and a part containing only role and concept symbols not part of the static abox? This is an open problem.

Even if separability in the sense above holds, in terms of feasible implementation, the reference to a large static abox remains a challenging problem. As far as we know, this problem has not been solved satisfactorily by any of the current temporal and streamified OMQA systems.

OBDA rewritability STARQL queries with standard sequencing can be rewritten into queries over backend data stream management system. This is possible because the two layers in STARQL, the semantics of the outer temporal FOL template and the semantics of the embedded ECL queries, are separated. This is similar to the temporal conjunctive queries (TCQs) of [9]. For the details of rewritability and a comparison of STARQL with the query languages of TCQ we refer the reader to [24].

An alternative operational semantics The window semantics defined above is denotational and mimics the window operator definitions for CQL [4], which

is one of the first relational data stream query languages. From the implementation point of view, an operational semantics is more helpful—at least it gives a different perspective on the intended semantics of windows. Furthermore, the operational view also sheds light on why the window definition was chosen exactly the way as stated above. For the details of the operational semantics we refer the reader to [18].

3 A Criterion for Bounded-Memory Computability for SQL Queries over Streams

We have seen that in STARQL, queries can refer to streams that may contain infinitely many different RDF triples. Moreover, we saw that naive implementations of queries such as the linear-space, quadratic-time implementation of monotonicity may lead to non-efficient query processing. Hence, finding good criteria for bounded-memory processing of STARQL queries is a real issue. In order to find such criteria, in the following, we consider criteria known to hold for SQL queries over relational data streams.

The early work of Arasu and colleagues [3] gives syntactic criteria for bounded memory computability of queries in the SPJ (select-project-join fragment) of SQL and also for an extension of SPJ with aggregation operators. In each case they consider both natural set semantics and multi-set (alias: bag) semantics. Moreover they describe an algorithm that in case of bounded-memory computability constructs a corresponding bounded-memory stream algorithm. Though SQL, per se, does not provide stream specific operators as in specific stream query languages (such as CQL [4]), the results are still fundamental enough in order to be adaptable to genuine stream query languages.

The underlying computation model for the bounded-memory results is described only informally in [4]. It is a register machine model extended to handle infinite input streams. Such a computation model can be formally described by streaming abstract state machines [16].

3.1 Query Language and the Query Model

We assume that the user is familiar with the SPJ-Fragment of SQL. We just restate some SPJ queries from [3] in order to illustrate the memory-boundedness criterion.

Assume that you have two homogeneous data streams, one containing tuples of the form $S(A, B, C)$ with a ternary relation S and a stream containing tuples of the form $T(D, E)$. All attributes (here A, B, C, D, E) are assumed to range over the integers. The queries are constructed using a projection operator $\Pi \in \{\pi, \dot{\pi}\}$, where π is the duplicate eliminating projection operator and $\dot{\pi}$ is the duplicate-preserving operator. The selection operator σ is restricted to conjunctions of atoms of the form $X = Y$ and $X > Y$, where X, Y are either attributes or integer constants. The join is a full join with the cartesian product $\times$. An example

query which is evaluated with multi-set semantics, i.e., Duplicate Preserving, is the following query:

$$Q_3^{DP} = \dot{\pi}_A(\sigma_{(A=D \wedge A>10 \wedge D<20)}(S \times T))$$

The query asks for all values A (with duplicates) with $20 > A > 10$ such that there are tuples of the form $S(A, \cdot, \cdot)$ and $T(A, \cdot)$. In logical notation, this is the conjunctive query

$$\exists B, C, D, E.S(A, B, C) \wedge T(D, E) \wedge A = D \wedge A > 10 \wedge D < 20$$

What is the process model for evaluating this query, when S and T do not stand for static tables but streams?

The query is answered over one inhomogeneous big stream of tuples. The stream is inhomogeneous in the sense that tuples belonging to different relations may arrive (in case of the above query: tuples from S and tuples from T.) This is usually the case in the area of complex event processing (see, e.g., [2]). The idea is that the big stream is the result of merging—or *interleaving* as Arasu and colleagues call merging—many homogeneous streams (i.e. streams where every tuple belongs to exactly to one relation, here: the two homogeneous streams associated with S and T). Interleaving means that an arbitrary sequence of tuples is fixed which consists of tuples from the referenced homogeneous stream.[4] For example, if $S = \langle S(1,1,1), S(2,2,2), S(3,3,3), \dots \rangle$ and $T = \langle T(1,1), T(2,2), T(3,3), \dots \rangle$, the following big stream is a possible interleaving

$$BS_1 = \langle S(1,1,1), S(2,2,2), S(3,3,3), T(1,1), T(2,2), T(3,3), \dots \rangle$$

Another is

$$BS_2 = \langle S(1,1,1), T(1,1), S(2,2,2), T(2,2), S(3,3,3), T(2,2), \dots \rangle$$

and so on. In many modern stream query languages following a pipeline architecture, these kinds of interleavings are not completely outsourced to a system but are controlled with the query language using cascading of stream queries. Such a control is given also in STARQL. The criteria of memory-boundedness mentioned below are to be understood to hold for all (!) possible interleavings.

Now, how is a query such as Q_3^{DP} evaluated? Every time t a new tuple in the big stream BS arrives it is stored in an ordinary SQL DB containing all tuples arrived so far. The query is evaluated on the accumulated DB with the last tuple. So, one has a notion of an output of a query Q at time t over the big input stream BS, $ans(Q, t, BS)$, which is defined as $Q^{DB(BS^{\leq t})}$ that is the answer of the query Q on the accumulated DB from the t-prefix of the stream BS. The output at every time t is a set (or multi-set). This definition of the output stream corresponds to the IStream semantics of CQL [4].

[4] We note that there is no fairness assumption for the interleavings in [3].

Now one could associate with a query and a big stream BS the stream of answers $(ans(Q,t,BS)_{t\in\mathbb{N}})$. But actually, the authors of [3] associate an output stream with a query over the input big stream in a different way as they want to have a stream of tuples again. So they consider a stream of elements produced so far and consider the multi-set-union over this prefix as the intended answer of the query. As the authors consider only monotonic queries they assume that the answer stream can be given by reference to the answers produced so far multi-unioned with the answer produced at the current time stamp. To formalize this, let us assume that a query Q maps an input stream BS_{in} into an output stream $Q(BS_{in}) = BS_{out}$. Then, one demands that for every arrival time point t one has $ans(Q, BS_{in}^{\leq t}) = \Cup BS_{out}^{\leq t}$, where $\Cup$ is defined for a sequence of elements $(s_i)_{i\leq t}$ as the multi-set of elements by multi-union of all the $\{s_i\}$.

3.2 Criterion for SPJ Queries

The following table gives some example queries and states which of them are bounded-memory computable. Duplicate preserving queries have a DP superscript, duplicate eliminating ones have a DE superscript. As before, we assume two homogenous streams, with elements of the form $S(A,B,C)$ and the other with elements of the form $T(D,E)$.

Acronym	**Query**	**Memory-Bounded?**
Q_1^{DP}	$= \dot{\pi}_A(\sigma_{(A>10)}(S))$	yes
Q_1^{DE}	$= \pi_A(\sigma_{(A>10)}(S))$	no
Q_3^{DP}	$= \dot{\pi}_A(\sigma_{(A=D\wedge A>10\wedge D<20)}(S\times T))$	yes
Q_3^{DE}	$= \pi_A(\sigma_{(A=D\wedge A>10\wedge D<20)}(S\times T))$	yes
Q_4^{DP}	$= \dot{\pi}_A(\sigma_{(B<D\wedge A=10)}(S\times T))$	no
Q_4^{DE}	$= \pi_A(\sigma_{(B<D\wedge A=10)}(S\times T))$	yes

The first query Q_1^{DP} is memory bounded as it acts as a simple filter: there is no join condition and the query answering system does not have to eliminate duplicates. This is different for the query Q_1^{DE} which is the same as the first except for using duplicate elimination: As A is not bounded from above, the system would have to store any $A > 10$ arrived in S so far in order not to output them a second time.

Both queries Q_3^{DP}, Q_3^{DE} are memory-bounded. The algorithm in the duplicate-preserving case has synopses for S and T, resp. Both synopses consist of registers for all integer values v in the range $[11, 19]$. A register for value v in the S-synopsis stores the number of S-tuples having $A = v$. Similarly the register for value v in the T-synopsis counts all T-tuples arrived so far with value $D = v$. Now, assume for example that the next element in the big stream is an S-tuple with $A = v$. If v is not in the interval $[11, 19]$, it is ignored. Otherwise one considers the number of T-tuples in the v-register of the T-synopsis. This number of v tuples is put onto the output stream. The duplicate-eliminating case is similar but one stores just boolean values in the registers instead of number counts.

In case of the pair of queries Q_4^{DP}, Q_4^{DE} the duplicate-preserving one is not bounded-memory computable, whereas the duplicate eliminating query is. Regarding the latter one constructs a synopsis for S where the minimum value of attribute B among all tuples of S with $A = 10$ having arrived so far are stored, and one has a T-synopsis, in which the maximum value of attribute D among all tuples of T so far is stored. For the duplicate-preserving query Q_4^{DP} it is not enough to know whether a stream joins with a past stream but one has to count the number—and these numbers are not bounded.

We state the syntactic criterion for the duplicate-eliminating case only as we are going to consider the set-semantics for STARQL only. The syntactic criterion is formulated for SPJ-queries that have a special form. These queries are called locally totally ordered queries, for short, *LTO queries*. As every query is equivalent to a union of LTO queries, a query is memory-bounded iff all its LTOs are (Theorem 5.3 in [3]). An *LTO query* Q is a query in which for every stream S referenced in Q the union of attributes in S and all constants occurring in Q are totally ordered.

Let P be a selection predicate, i.e., a conjunction of atoms of the form $X > Y, or X = Y$. Let P^+ denote the set of all atoms entailed by P that contain only symbols of P.

An attribute A is called *lower-bounded* (resp. *upper-bounded*) if there exists an atom $A > k \in P^+$ (resp. $A < k \in P^+$), or an atom $A = k \in P^+$ for some constant k. A is *bounded* if it is both upper-bounded and lower-bounded. Two elements e and d (variables or constants) are called equivalent w.r.t. a set of predicates iff $e = d \in P^+$. Then $\mid E \mid_{eq}$ denotes the number of equivalence classes into which a set of elements E is partitioned according to the equivalence relation above.

Now consider a stream S_i referenced in a query Q. $MaxRef(S_i)$ is defined as the set of all lower bounded but not upper-bounded attributes A of S_i such that A appears in a non-redundant inequality join $(S_j.B < S_i.A)$, $i \neq j$, in P^+. The definition of $MinRef(Si)$ is the dual of the definition of $MaxRef(S_i)$. With this definition the following characterization can be proved.

Theorem 1 ([3], Thm 5.10). *Let $Q = \pi_L(\sigma_P(S_1 \times \cdots \times S_n))$ be an LTO query. Q is bounded-memory computable iff:*

C1: *Every attribute in the list L of projected attributes is bounded.*
C2: *For every equality join predicate $(S_i.A = S_j.B)$, $i \neq j$, $S_i.A$ and $S_j.B$ are bounded.*
C3: $\mid MaxRef(S_i) \mid_{eq} + \mid MinRef(S_i) \mid_{eq} \leq 1$ *for* $i \in \{1, \ldots, n\}$.

4 Lifting the Criteria of Bounded-Memory Criteria Bounded-Memory Computability to OMQA

As STARQL can be used with unbounded windows, in STARQL, we face problems similar to the ones for the model of SQL stream processing described in the previous section. Even if one were to consider finite windows, considerations on

bounded-memory stream processing could give insights into optimization means. In the previous section the relational tuples were defined over the integers, a discrete, infinite domain. If one considers also dense domains, then one has a similar if not a more difficult problem of ensuring bounded-memory computability. But even here one can sometimes guarantee bounded-memory computability as the following monotonicity example (a variant of the example from the beginning) suggests.

Consider the simple monotonicity query in Fig. 3, asking every second whether the temperature in sensor $s0$ increased monotonically up to the current time point.

```
...
{FROM SMsmt [0, NOW]->1s }
...
{HAVING FORALL i < j IN SEQ1,?x,?y:
       IF {s0 val ?x}<i>  AND {s0  val ?y}<j>
       THEN ?x <= ?y  }
```

Fig. 3: Simple Monotonicity Query

A straight-forward implementation of this query is to construct from scratch sequences of `[0, NOW]`-windows and test on these the monotonicity condition by iterating trough all possible state-pairs (i, j). But this results in a test of quadratic order (w.r.t. time). It is not hard to see that first one can find a complete and correct algorithm that is not quadratic in time and uses constant space only (w.r.t. the uniform cost measure in register machines): it just stores the maximal temperature value for the last time point and compares it with the values arriving at the current time point. Of course, this optimization is possible only if it can be guaranteed that the input streams are not out of sync, i.e., if tuples with earlier time stamps than the current time point are excluded. In such an asynchronous case one would have to store all possible measurement values.

The results of [3] can be adapted to formulate criteria for bounded-memory computability on STARQL queries. For this we consider the following variant of STARQL, which, syntactically, is a simple fragment, called CQ-fragment of STARQL and denoted STARQL$^{\mathrm{CQ}}$, but which, semantically, differs in applying the IStream semantics and not the RStream semantics.

Definition 1. *Syntactically STARQL^{CQ} is defined as that fragment of STARQL adhering to the following constraints:*

1. *The* `FROM` *clause refers only to streams that are streams of abox axioms (or RDF tuples).*

2. *All sliding windows are unbounded windows (with the same slide which is identical to the pulse).*
3. *The* `WHERE` *clause is allowed to be any reasonable query language allowing a certain answer semantics.*
4. *The sequencing strategy is that of standard sequencing.*
5. *The* `HAVING` *fragment is restricted to consist of conjunctive queries of the form*

$$\texttt{EXISTS}\ y_1, y_2, \ldots, y_n A(x_1, \ldots, x_m, y_1, \ldots, y_n)$$

where the x_i are non-state variables which may occur as free variables in the `WHERE` *clause or the x_i are state variables, and the y_i are state variables or non-state variables bounded by the* `EXISTS` *quantifier. The expression after the quantifiers $A(x_1, \ldots, x_m, y_1, \ldots, y_n)$ is a conjunction of atoms in which only variables $x_1, \ldots, x_n, y_1, \ldots, y_n$ may occur and which have one of the following forms.*

- `GRAPH i` $r(x, y)$ *where*
  r is a role symbol, x is a constant or a variable in $\{x_1, \ldots, x_m, y_1, \ldots, y_n\}$
- `GRAPH i` $C(x)$ *where*
  C is an atomic concept symbol, x is a constant or a variable in the set $\{x_1, \ldots, x_m, y_1, \ldots, y_n\}$
- $x\ op\ y$ *where*
  $op \in \{<, >, =\}$ and x is a constant or variable in $\{x_1, \ldots, x_m, y_1, \ldots, y_n\}$

Semantically, STARQL^{CQ} uses the IStream semantics.

Please note that in the first item of the definition we exclude the reference to streams that are constructed with other STARQL queries. Otherwise we would have to consider criteria for composed queries. We do not exclude that this is possible, but the adaptation would be rather awkward.

Now we get the following adapted version of the syntactic criterion.

Proposition 1. *Let Q be an LTO query in the STARQL^{CQ} fragment. We make the following assumptions*

1. *The streams are interleaved in a synchronized way, i.e., the time points of tuples adheres to the arrival ordering.*
2. *At every time point only a finite number of elements bounded by some constant can arrive.*
3. *The tbox is empty. (But see Corollary 1 where this assumption is dropped.)*

Then, Q is memory-bounded iff it fulfills the following constraints.

C1: Every variable appearing in the `HAVING` *clause is either bounded by* `EXISTS` *or is a-bounded or occurs free in the* `WHERE` *clause.*

C2: For every non-state variable that occurs in two atoms or in an identity atom: if it is not a state-variable, then it is bounded or occurs free in the `WHERE` *clause.*

C3 : $| MaxRef(S_i) |_{eq} + | MinRef(S_i) |_{eq} \leq 1$ *for* $i \in \{1, \ldots, n\}$.

Please note that we have the following differences w.r.t. the criterion of [3]: First of all, there is a `WHERE` clause. Evaluating the variables occurring in a `WHERE` clause does not pose a problem regarding bounded-memory computability as these variables refer to the static database (which is finite). So all variables bounded by the `WHERE` clause can be considered to be bounded in the sense of Arasu.

The STARQL query language allows composing queries, i.e., a STARQL query may refer to streams produced by other STARQL queries. This composition leads to constraints regarding interleaving which are not captured by the criteria of Arasu (hence we considered only non-cascaded queries).

In the case of STARQL and STARQL^{CQ} the time flow is not restricted to a discrete domain. In the case of non-discrete time domains it is important to know that the streams are synchronized. Otherwise, as we mentioned above in the monotonicity example, one would have to store measurement values for all time points of tuples having arrived so far as one cannot exclude the case that values for time points in between may arrive.

As the proposition shows, the impossibility of bounded-memory processing may also be due to non-bounded memory computability w.r.t. the output. This was one reason why, in the original definition of STARQL, we decided to use the RStream semantics for STARQL. The other reason was that for non-monotonic queries and aggregation queries one cannot rely on IStream semantics. On the other hand, the use of an RStream semantics means that due to the possibility of unbounded sets of answers at every time point, the system may fall behind the pulse requirements. So, we have here a classical opposition of time and space constraints.

In the proposition we assume that the tbox is empty. In the STARQL framework the tbox is assumed to be atemporal. But even then one cannot exclude that a tbox axiom may lead to the loss of bounded-memory computability, although the query w.r.t. the empty tbox is bounded-memory computable. For example, a simple role inclusion axiom may lead to a self join in the query, which is handled via other syntactic criteria. It is an interesting open question to find criteria on the tbox that preserve the correctness and completeness of the syntactical criteria for bounded-memory computability

On the other hand, if we consider OBDA, which allows for perfect rewriting of queries, then we can apply the criterion of Proposition 1 to each completion of each conjunctive query in the rewritten query (which is a union of CQs, for short a UCQ).

Corollary 1. *Let Q be a query in the STARQL^{CQ} fragment w.r.t. some DL allowing for perfect rewritability. Let Q_{rew} be the rewritten UCQ. We assume that each prefix of the abox stream is consistent with the ontology.*

If each of the LTO queries to each CQ in Q_{rew} fulfills the conditions mentioned in Proposition 1, then and only then, Q is bounded-memory computable.

Note that we assume consistency of the abox-stream prefixes with the tbox. Otherwise one would have to test for inconsistency. This test can be reduced

to first-order logic queries but the resulting queries are not bounded-memory computable. Consider, e.g., a negative inclusion $A \sqsubseteq \neg B$ which would lead to an unbounded query $\exists A(x) \wedge B(x)$. Similar considerations follow for functional constraints.

On the other hand, when considering such axioms to hold only on the extensional part of the ontology, i.e., if we consider integrity constraints on the abox streams, then it is possible to gain bounded-memory computability. In the the monotonicity example above, we already discussed a similar form of integrity constraint, though it was not formulated as a DL axiom. A systematic study of the consequences of integrity constraints for bounded memory computability is left for future work.

5 Conclusion

The study of bounded-memory computability of OMQA queries can profit from corresponding results on bounded-memory processing over relational data streams. The adaptations are not trivial in the presence of tboxes—for languages in which the tbox cannot be compiled away. We presented a syntactical criterion for the information processing paradigm of OBDA in which the test of bounded-memory processing w.r.t. a non-empty tbox could be reduced to a test with an empty tbox. The question of how the syntactical criterion of [3] can be adapted to the general OMQA case, in particular for tboxes in which temporal constructors are allowed [5], is an open problem. We guess that due to the syntacticality of the criterion, the adaptation is not going to be obvious. Hence, as a further future research topic we think of an equivalent semantic criterion for bounded-memory processing using the framework of dynamic complexity [25].

References

1. Aggarwal, C.C. (ed.): Data Streams - Models and Algorithms, Advances in Database Systems, vol. 31. Springer (2007)
2. Agrawal, J., Diao, Y., Gyllstrom, D., Immerman, N.: Efficient pattern matching over event streams. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data. pp. 147–160. SIGMOD '08, ACM, New York, NY, USA (2008)
3. Arasu, A., Babcock, B., Babu, S., McAlister, J., Widom, J.: Characterizing memory requirements for queries over continuous data streams. ACM Trans. Database Syst. **29**(1), 162–194 (Mar 2004)
4. Arasu, A., Babu, S., Widom, J.: The CQL continuous query language: semantic foundations and query execution. The VLDB Journal **15**, 121–142 (2006)
5. 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, AAAI Press (2013), `http://dl.acm.org/citation.cfm?id=2540128.2540232`
6. Baader, F., Borgwardt, S., Lippmann, M.: Temporalizing ontology-based data access. In: CADE-13 (2013)

7. Bauer, A., Küster, J.C., Vegliach, G.: From propositional to first-order monitoring. In: Legay, A., Bensalem, S. (eds.) Runtime Verification, Lecture Notes in Computer Science, vol. 8174, pp. 59–75. Springer Berlin Heidelberg (2013), http://dx.doi.org/10.1007/978-3-642-40787-1_4
8. Bienvenu, M.: Ontology-mediated query answering: Harnessing knowledge to get more from data. In: Kambhampati, S. (ed.) Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016. pp. 4058–4061. IJCAI/AAAI Press (2016), http://www.ijcai.org/Abstract/16/600
9. 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 Berlin Heidelberg (2013)
10. Borgwardt, S., Lippmann, M., Thost, V.: Temporalizing rewritable query languages over knowledge bases. Journal of Web Semantics **33**, 50–70 (2015). https://doi.org/https://doi.org/10.1016/j.websem.2014.11.007, http://www.sciencedirect.com/science/article/pii/S157082681400119X
11. 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). https://doi.org/10.4018/jswis.2012010103, http://dx.doi.org/10.4018/jswis.2012010103
12. Chomicki, J.: Efficient checking of temporal integrity constraints using bounded history encoding. ACM Trans. Database Syst. **20**(2), 149–186 (Jun 1995)
13. Chomicki, J., Toman, D.: Temporal databases. In: Handbook of Temporal Reasoning in Artificial Intelligence, vol. 1, pp. 429–467. Elsevier (2005)
14. Cormode, G.: The continuous distributed monitoring model. SIGMOD Record **42**(1), 5–14 (2013)
15. 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, Lecture Notes in Computer Science, vol. 5468, pp. 72–81. Springer Berlin / Heidelberg (2009)
16. Gurevich, Y., Leinders, D., Van Den Bussche, J.: A theory of stream queries. In: Proceedings of the 11th International Conference on Database Programming Languages. pp. 153–168. DBPL'07, Springer-Verlag, Berlin, Heidelberg (2007)
17. Kharlamov, E., Kotidis, Y., Mailis, T., Neuenstadt, C., Nikolaou, C., Özçep, Ö.L., Svingos, C., Zheleznyakov, D., Brandt, S., Horrocks, I., Ioannidis, Y.E., Lamparter, S., Möller, R.: Towards analytics aware ontology based access to static and streaming data. In: Groth, P.T., Simperl, E., Gray, A.J.G., Sabou, M., Krötzsch, M., Lécué, F., Flöck, F., Gil, Y. (eds.) Proceedings of the 15th International Semantic Web Conference (ISWC-16), Part II. LNCS, vol. 9982, pp. 344–362 (2016)
18. Kharlamov, E., Kotidis, Y., Mailis, T., Neuenstadt, C., Nikolaou, C., Özçep, Ö.L., Svingos, C., Zheleznyakov, D., Ioannidis, Y., Lamparter, S., Möller, R.: An ontology-mediated analytics-aware approach to support monitoring and diagnostics of static and streaming data. Journal of Web Semantics (2018), in Print
19. Kharlamov, E., Mailis, T., Mehdi, G., Neuenstadt, C., Özçep, O.L., Roshchin, M., Solomakhina, N., Soylu, A., Svingos, C., Brandt, S., Giese, M., Ioannidis, Y., Lamparter, S., Möller, R., Kotidis, Y., Waaler, A.: Semantic access to streaming and static data at Siemens. Web Semantics: Science, Services and Agents on the World Wide Web pp. 54–74 (2017). https://doi.org/http://dx.doi.org/10.1016/j.websem.2017.02.001, http://www.sciencedirect.com/science/article/pii/S1570826817300124

20. Kharlamov, E., Solomakhina, N., Özçep, Ö.L., Zheleznyakov, D., Hubauer, T., Lamparter, S., Roshchin, M., Soylu, A., Watson, S.: How semantic technologies can enhance data access at siemens energy. In: Mika, P., Tudorache, T., Bernstein, A., Welty, C., Knoblock, C.A., Vrandecic, D., Groth, P.T., Noy, N.F., Janowicz, K., Goble, C.A. (eds.) The Semantic Web - ISWC 2014 - 13th International Semantic Web Conference, Riva del Garda, Italy, October 19-23, 2014. Proceedings, Part I. Lecture Notes in Computer Science, vol. 8796, pp. 601–619. Springer (2014)
21. 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)
22. 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)
23. 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)
24. Özçep, Özgür.L., Möller, R., Neuenstadt, C.: Stream-query compilation with ontologies. In: Pfahringer, B., Renz, J. (eds.) Proceedings of the 28th Australasian Joint Conference on Artificial Intelligence 2015 (AI 2015). LNAI, vol. 9457. Springer International Publishing (2015)
25. Patnaik, S., Immerman, N.: Dyn-FO: A parallel, dynamic complexity class. Journal of Computer and System Sciences **55**(2), 199–209 (1997)
26. 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: Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N.F., Blomqvist, E. (eds.) The Semantic Web - ISWC 2011 - 10th International Semantic Web Conference, Bonn, Germany, October 23-27, 2011, Proceedings, Part I. Lecture Notes in Computer Science, vol. 7031, pp. 370–388. Springer (2011)
27. Thost, V.: Using Ontology-Based Data Access to Enable Context Recognition in the Presence of Incomplete Information. Ph.D. thesis, TU Dresden (2017)
28. Turhan, A., Zenker, E.: Towards temporal fuzzy query answering on stream-based data. In: Nicklas, D., Özçep, Ö.L. (eds.) Proceedings of the 1st Workshop on High-Level Declarative Stream Processing co-located with the 38th German AI conference (KI 2015), Dresden, Germany, September 22, 2015. CEUR Workshop Proceedings, vol. 1447, pp. 56–69. CEUR-WS.org (2015), `http://ceur-ws.org/Vol-1447/paper5.pdf`