Deliverable D5.1
A Semantics for Temporal and Stream-Based Query Answering in an OBDA Context

Due date of deliverable: \((T0+12)\)
Actual submission date: October 31, 2013

Start date of the project: **1st November 2012**  
Duration: **48 months**

Lead contractor for this deliverable: **UOXF**

Dissemination level: **PU – Public**
Executive Summary:
A Semantics for Temporal and Stream-Based Query Answering in an OBDA Context

This document summarises deliverable D5.1 of project FP7-318338 (Optique), an Integrated Project supported by the 7th Framework Programme of the EC. Full information on this project, including the contents of this deliverable, is available online at http://www.optique-project.eu/.

The main objective of WP5 in year one was to define a semantics for an ontology based temporal and stream query language meeting the Siemens use case requirements (O5.1). This objective was met by the development of STARQL (pronounced Star-Q-L), which is a Reasoning-based for Streaming and Temporal ontology Access.

As part of the development, the WP5 partners analyzed related query languages and their implementations, the conclusion being that there is still (not only due to the requirements of the use cases in OPTQUE) a need for a query language that uses ontologies for accessing streaming and temporal data.

Having in mind the requirements in the Sensor Measurement Scenarios in general as well as the technical requirements (Deliverable 1.1), user requirements (see query catalog in D8.1), and architectural (Deliverable D2.1) requirements of the Siemens Use Case in particular, STARQL was designed as an expressive and flexible stream query framework that offers the possibility to embed different (temporal) description logics as filter query languages over ontologies. Task 5.1, associated with objective O5.1, mentions the development of a rewritable query language. Rewritability can be reached within STARQL due to its plug-in feature, which makes it possible to embed lightweight description logics such as DL-Lite and filter query languages such as conjunctive queries—language combinations which are designed towards rewritability. On the other hand, STARQL can also embed other filter query languages that can be combined with more expressive DL ontology languages such as SHI. This shows STARQL’s flexibility to implement orthogonal approaches such as ABox modularization, a technique that enables accessing big data over very expressive ontologies (ABDEO).

STARQL’s syntax is founded formally by an (extended) context-free language and its semantics is founded formally by a denotational semantics. These formal descriptions lay the ground for the next WP5 tasks for year 2 and beyond. In particular, they are going to be used as references for the implementation of the streaming and temporal query answering module within the OPTIQUE platform.

Authors
Özgür L. Özçep (TUHH)
Ralf Möller (TUHH)
Christian Neuenstadt (TUHH)
Dmitriy Zheleznyakov (UOXF)
Evgeny Kharlamov (UOXF)

Contributors
Ian Horrocks (UOXF)
Thomas Hubauer (SIEMENS)
Mikhail Roshchin (SIEMENS)
## Contents

1 Introduction ............................................................. 4

2 Related Work ............................................................ 6

3 Streamified and Temporalized OBDA in the Sensor Measurement Scenario 10
   3.1 Stream Definition .................................................. 10
   3.2 The Sensor Measurement Scenario .............................. 10
   3.3 Lifting the Data to the Logical Level ......................... 12

4 The Query Language STARQL ........................................... 16
   4.1 A Basic Example .................................................... 16
   4.2 Dealing with Indefiniteness .................................... 18
   4.3 Embedded Filter Queries and Entailment ..................... 19
   4.4 Query Orthogonality and the Scope of Assertions .......... 22
   4.5 Multiple Streams .................................................. 23
   4.6 Temporal Reasoning in STARQL ................................. 24

5 Formal Syntax and Semantics ........................................ 26
   5.1 Syntax ................................................................. 26
   5.2 Semantics ............................................................. 30
      5.2.1 Window semantics ............................................. 30
      5.2.2 Merge for Multiple Streams ............................... 31
      5.2.3 The Sequencing Method .................................... 31
      5.2.4 The Semantics of the Where and Having Clauses ....... 31
      5.2.5 Arithmetic Aggregation Operators ....................... 32
      5.2.6 Semantics of Select Expressions ......................... 33

6 Summary ................................................................. 34

Bibliography .............................................................. 36

A Outline of Time and Streams Architecture ........................ 43

B Modularization Approach ............................................. 52
Chapter 1

Introduction

According to the requirements of the Siemens use case with sensor data (see deliverables 1.1 and 8.1) this article develops a language for specifying ontology-based queries w.r.t. an ever extended set of time-tagged data descriptions. We call a set of this kind a “stream” of data descriptions. In contrast to standard database queries, stream-based queries are continuously evaluated according to a set of “incoming” data descriptions. In the Siemens industrial use case, descriptions are generated for measurements with a large set of sensors mounted on various turbine assemblies. Queries are registered to a monitoring system for preventive and reactive diagnosis. In particular, for reactive diagnoses, measurement streams might be “simulated” and the very same queries might be answered w.r.t. different ontologies in order to understand certain effects occurring in a complex technical system. The power plant in which a turbine is installed is described formally, and answering queries for streams of data descriptions (dynamic descriptions) also involves a large set of (static) assembly or control unit structure descriptions. Queries referring to static structures definitely involve a huge set of data descriptions even in a streaming context, and queries can be formulated in a more convenient way by engineers if appropriate notions defined in an ontology can be used. Thus, scalable query answering w.r.t. possibly changing ontologies needs to be investigated in a stream-based setting. In WP5 of Optique we particularly explore a framework for query answering whose main idea is rewrite an ontology-based query into a new one that can be executed on a standard database system, while returning the same result (ontology-based data access, OBDA, see also Deliverable 6.1).

Besides OBDA, which provides support for lightweight ontologies only (e.g., DL-Lite), the framework should also be suited to provide access to big data with expressive ontologies in a streaming context, and this in combination with huge static data. Despite the fact that many people believe that OBDA and query rewriting is a necessary prerequisite for achieving scalability, recently, encouraging performance results for query answering also have been achieved for ontologies more expressive than DL-Lite (see [74] and Appendix B). Thus, there is good hope that scalable ontology-based data access need not be limited to so-called data-oriented description logics such as DL-Lite. In order to contrast rewriting-based query answering (OBDA) from other techniques for Accessing Big Data with Expressive Ontologies we call the latter techniques ABDEO approaches. With expressive description logics there is better support for developing conceptual data models because less needs to be made explicit using mapping rules, for which there is no notion such as, e.g., classification defined. We nevertheless focus on OBDA first, and presume prior knowledge in description logics (DLs) for understanding technical details of the semantics of the stream-based data processing framework discussed in this paper.

Stream processing has strong connections to the processing of temporal data in a temporal DB using temporal (logic) operators; but nonetheless, the scenarios, the objects of interest, the theoretical and practical challenges are different from those of temporal logic and temporal DBs (see also the next section). While query answering on temporal databases is a one-step activity on static historical data, answering queries on streams is a reactive, continuous activity (hence the notion of a continuous query). The architecture of the access to streaming and historical data in OBDA systems is outlined in Appendix A.

A challenging fact of stream processing is the high frequency of the data in some application contexts.
A second challenging fact is the changing frequency (burstiness) of incoming data, which is quite a natural phenomenon for event messages, which are produced in the moment the related events occur. Furthermore, in contrast to temporal DBs settings, stream scenarios come up with multiple queries as there may be many streams corresponding, e.g., to different sensors and control units, and as there are many features that one wants to query, such as different statistical and time-series features. We propose an orthogonal query language which can refer to streams in order to produce new streams etc. Details of this language will be derived and explained in this deliverable.

The article is structured as follows. We first discuss related work to motivate the query language introduced afterwards. With a set of example developed from an industrial use case scenario we motivate the main features of this language. Then, syntax and semantics of the query language are introduced formally. In the last section, the article concludes with a summary.
Chapter 2

Related Work

Processing time-dependent data has been investigated for a long time. Starting from early approaches in computer vision \cite{76, 77, 85}, temporal data processing using declarative techniques has been extensively investigated in artificial intelligence (e.g., \cite{62, 63, 51, 50, 23, 12}). However, the focus of this work is more towards event recognition rather than on techniques for answering continuous queries (efficiently). Nevertheless, all of these approaches use some form of temporal logic for application purposes.

From a more theoretical perspective, temporal logics have been extensively investigated in the literature as well. There are three main ways to introduce time in logic \cite{97}: First-order logic with temporal arguments, modal temporal logic, and reified temporal logic. The first one is the method of Temporal Arguments \cite{49, 17}. The main idea of this approach is to represent time just as another parameter in a first-order predicate calculus. Functions and predicates are extended with the additional temporal argument denoting the particular time at which they have to be interpreted. The second approach consists of including the concept of time in the semantic structure (interpretation) of a formula. Most approaches are based on a possible-worlds semantics \cite{66}, that was re-interpreted in the temporal context by making each possible world represent a different time. A Modal Temporal Logic extends propositional or first-order logic with modal temporal operators \cite{83, 41}. The third approach discussed in the literature is Temporal Reification \cite{3, 71}. The main idea of this approach is to allow one to talk about the truth of assertions while staying in a first-order logic. Reifying a logic involves moving into the meta-language where a formula of the initial language i.e., the object language, becomes a term—namely, a propositional term—in the new language. Thus, in this new logic one can reason about the particular aspects of the truth of expressions of the object language through the use of predicates such as true. Usually, the first-order logic is taken as the object language.

Different temporal applications require different time models; therefore, one can find a lot of different time models proposed in the literature. Analyzing these models, one can notice that there is a number of dimensions relating to time and a particular choice in those dimensions corresponds to the particular time model. The usual dimensions are the following:

- **Time Points vs. Time Intervals.** The temporal primitive, to which variables of a logic refer to, can be either a point \cite{26, 58, 71} or an interval \cite{2}.

- **Discrete vs. Dense.** Time is considered as a discreet collection of time elements or, instead, for any two elements there is always a third element between them.

- **Bounded vs. Unbounded.** This concerns the question of whether time is considered infinite in either of both directions.

- **Precedence.** The time domain can be linear, branching, parallel, circular etc.

Note that the complexity of reasoning strongly depends on the structure of time independently of the logic formalism (it is discussed for Modal Temporal Logic in \cite{48} and for many-sorted temporal logic in \cite{17}).

Snodgrass and Ahn \cite{89} introduced the notion of valid and transaction time; the former one denotes the time period during which a fact is or will be valid in the modelled reality, while the latter one denotes the time
period during which a fact stored in the data base is considered to be true. In particular for measurement data, timestamps for data descriptions denote acquisition times, which are generated by sensors. Since a global clock is usually not available, locally recorded acquisition times often need to be synchronized in application specific ways for query answering (user-defined time).

There are a lot of proposals of how to extend description logics with time; Artale and Franconi [10] and Lutz et al. [70] presented comprehensive overviews of the relevant results. These works are focused on expressing and reasoning with temporal constraints that (usually) allow for quantification over unknown time instances.

Avery and Yearwood [13] proposed to represent time changes in the ontology by means of versioning. Their approach, however, suffers from two main flaws: (i) information redundancy due to information repetition in different versions of the ontology, and (ii) changes in time can only be inferred by comparing the present and the past states and cannot be directly represented in the ontology.

Welty and Fikes [98] and Hobbs and Pan [53] proposed to incorporate time into OWL ontologies as an additional dimension. In the similar fashion, Kim et al. [59] proposed an (interval-based) temporal web ontology language TL-OWL, extending OWL by adding TemporalClass, TemporalVariable, temporalDescription and temporalRelation constructs into the language. Baratis et al. [18] developed a Temporal Ontology Query Language (TOQL) for querying ontologies represented using the approach of [98].

An alternative approach of adding time to OWL was proposed by Milea et al. [72, 73]. They introduced Temporal OWL (tOWL), where time instances were encoded as values of a particular concrete domain. Papadakis et al. [78] presented a Prolog Reasoner for Temporal ONtologies in OWL (PROTON) that is able to manage temporal information over tOWL ontologies.

Temporal data processing has been a very important topic in the relational database community for a long time (cf., e.g., [20, 35]). Following early standardisation approaches (e.g., TSQL2 [90]), temporal data management and temporal query language features have made its way into the SQL:2011 standard [67]. The main idea of SQL:2011 is to add two special columns to a table, which are declared to stand for a starting and an end point of a period during which the atomic fact corresponding to a row holds. SQL:2011 does not introduce a new datatype for temporal intervals. As we will see below, temporal data processing is also supported by extensions to standard database systems such as PostgreSQL (www.postgresql.org).

While temporal databases provide techniques for analysing persistent temporal data with ad hoc queries, in some applications queries are known in advance, but the applications require answers on-line, possibly for many queries registered at the data management system (so-called continuous queries). This has led to the view of stream-based data processing. We distinguish the sensor perspective (semantic sensor networks) and the database perspective (data stream management).

The sensor perspective of stream-based data processing pursues the idea of pushing data processing to the sensors and, as a consequence, investigates approximations for statistical data analysis operations in order to cope with memory and power limitations of sensors [37]. Query languages for sensor networks are investigated in [25, 43, 44, 42, 95]. Also here, data processing is pushed to the sensor level. Despite the practical importance of, for instance, streaming video data from sensors and to users, we do not consider this kind of data processing scenario in this paper because sensor-level processing is dedicated to a fixed set of predefined services. We therefore assume that sensor data can conveniently be transferred to a computer center on-line. Also, extremely high-frequency stream computations, for instance, carried out on specific hardware (see, e.g., [82]) is out of the scope of this article.

So-called “semantic sensor networks” are investigated in [88, 81, 61, 45]. Semantic web technology is used to annotate sensor data with spatial, temporal, and thematic semantic metadata (funding was provided as part of the EU project SensorGrid4Env). A Sensor Network (SSN) ontology is proposed to model sensor devices, systems, processes, and observations [36]. See also the descriptions below about stream-based query answering systems for RDF.

Also from a database perspective, with data passed from sensors to a central computer system, many stream related investigations by different research groups have been pursued in the period from 2003 to 2008. The results are data stream management systems (DSMSs), mainly with SQL-like stream query languages such as, for instance, CQL [6]. The languages are semantically well-founded, e.g., for CQL one finds a
denotational semantics in [8]. Systems have been tested with well-defined benchmarks (e.g., [7]). Besides theoretical work and academic prototypes, such as STREAM from Stanford [6], TelegraphCQ from Berkeley (based on extensions to PostgreSQL) [34], Aurora/Borealis from Brandeis, Brown and MIT [56], or PIPES from Marburg University [64,30,32,31,65], one can also see various commercial systems, either standalone systems such as StreamBase, Truviso (an extended version of TelegraphCQ and now part of commercial Cisco products), or stream extensions to commercial relational DBMSs (MySQL, PostgreSQL, DB2 etc.). Also PIPES now has a commercial successor developed by Software AG. Nevertheless, as of now, the stream community is far from having a standard for data management and querying (but see [57] for some first ideas).

Also for the RDF data model, the representation of temporal phenomena has been investigated for quite some time. Gutierrez et al. [46] presented a comprehensive framework for representing and querying validity time in RDF. They proposed semantics for temporal RDF graphs and studied their properties; besides, they provided a syntax to incorporate that framework into a standard RDF graphs. Hurtado and Vaisman [55] extended that framework with a class of temporal constraints involving anonymous timestamps. Motik [75] presented an approach that is based on the previous framework; however this new approach differs from the the one from [46] in several ways: (i) the new approach is applicable not only to RDF, but also to the languages of the OWL 2 family, (ii) it also allows to deal with unbounded time.

Pugliese et al. [84] extended the framework of [46] in several ways: (i) they proposed a concept of a Temporal RDF (tRDF) database that extended the notion of [46] by allowing temporal triples to unknown time points; (ii) they defined a temporal query language to query tRDF databases; (iii) they presented the index structure tGRIN suitable for graph pattern queries. However, they considered only conjunctive queries, thus ignoring some of the more advanced features of SPARQL.

Tappolet and Bernstein [93] discussed the implementation of the approach of [46]: they proposed an RDF-compatible syntax for temporal data, provided a strategy for efficiently storing RDF data, and introduced a temporal query language $\tau$-SPARQL that extends SPARQL.

Gutierrez-Basulto and Szymon [47] introduced a basic framework for representing temporal data (more precisely, temporal ABoxes with standard TBoxes) in arbitrary DLs; moreover, they proposed a general mechanism of defining corresponding temporal query language.

Udrea et al. [94] developed an extension of RDF, called Annotated RDF, with annotation that can be used, in particular, to annotate RDF data with time references. Straccia et al. [92] extended this proposal with a more general reasoning formalism, and Lopes at el. [69] presented a language to query Annotated RDF. In more recent work in the EU project TELEIOS[1], the Athens partner of Optique has defined the data model stRDF and the query language stSPARQL that allows the modeling and querying of geospatial data that change over time. The change in time is modeled by introducing the notion of a valid time of a triple. The approach has been fully implemented in the spatiotemporal RDF store Strabon[2] and has been shown in [21] to surpass in performance all other existing systems with similar functionality.

Stream-based processing has also been investigated w.r.t. the RDF data model. Prominent systems are C-SPARQL [40] (developed within LarKC, www.larkc.eu), SPARQLStream [27], or CQELS [80]. For querying time-tagged triples all systems use extensions to SPARQL. In the spirit of CQL also these systems use some form of “window” concept to support reactive data processing with continuous queries. C-SPARQL introduces the notion of “stream reasoning”, referring to answering window-based continuous queries w.r.t. RDFS+ ontologies.

Recently, processing temporal data has been discussed under the topic of complex event processing or information flow processing, and corresponding data management and query answering engines have been developed, e.g., EP-SPARQL/ETALIS [5] [4], T-REX [38,39], or Commonsens [91], with ESPER (esper.codehaus.org) being an open source successor. The latter ones are inspired by CQL in their treatment of streams. Prominent applications are object tracking and intrusion detection [22].

It should be noted that continuous queries as well as complex events as described above are usually
processed without considering a conceptual domain model (i.e., an ontology). Prototypes for event detection and data mining w.r.t. ontologies have been described in [100, 99, 14, 33, 52]. Description logic reasoning systems are used to incorporate a conceptual domain model into query answering, and, as a consequence, event detection. The advantage of using ontologies for specifying events and queries is that event definitions are much simplified, and various versions of the conceptual domain model can be explored without changing the queries or event specifications. Consequences of the conceptual data model (ontology) are indeed automatically identified by event detection and query answering systems supporting logical inferences (e.g., description logic inference systems). However, early proof-of-concept prototypes were not designed to meet big data scalability expectations.

To better support scalability for ontology-based query answering, a technique called “ontology based data access” (OBDA) has been proposed. In ontology-based data access, an ontology with moderate expressivity (DL-Lite [28]) is used to represent the conceptual domain model such that queries (or event specifications) can be rewritten in a sound and complete way in order to execute them on a conventional database system or triplestore. In addition, OBDA introduces the use of so called mapping rules for mapping database representations into ontology representations, with the advantage that ontology representations need not be materialised since mapping rules are considered in the query rewriting process (see, e.g., [86]).

Both, temporalising OBDA [24, 16, 15, 11] and streamifying OBDA (see, e.g., [40, 19] for RDFS++) are hot topics in current investigations. In particular, state sequences are investigated w.r.t. to OBDA on streaming data such that queries can be specified using formulas from linear temporal logic (LTL, [14, 11, 16]). Stream reasoning for EL++ is investigated in [68] in the context of behavior prediction and machine learning in general. Nonetheless, tests with different streaming benchmarks show [102, 79] that implemented systems for stream processing w.r.t. conceptual data models are just at the beginning of their development.

The plethora of approaches indicates that there is no single way to deal with streaming data, and hence, in the next section we will explore an extensible framework based on the ideas of ontology-based data access for scalability and window-based query specification for expressivity and flexibility such that event specifications or specifications in linear temporal logic can be integrated.
Chapter 3

Streamified and Temporalized OBDA in the Sensor Measurement Scenario

We will introduce the features of our streaming query language step by step with examples from sensor measurement scenarios. So we first discuss the notion of a stream, and then describe a concrete example of a sensor measurement scenario, namely the SIEMENS gas turbine use case scenario of streamed measurement data as well as event data generated by control units.

3.1 Stream Definition

Following the usual definition of temporal streams in the literature, we consider a temporal stream to be a set of pairs \((d, t)\). The first argument is instantiated by an object \(d\) from a domain \(D\), which we call the domain of streamed objects. The second argument is instantiated by a timestamp \(t\) from a structure \((T, \leq)\) of linearly ordered timestamps. We call this structure the time domain. In the examples introduced below, we can safely assume that \(T\) stands for the set of natural numbers with the usual ordering.

According to the definition, a stream may contain two different objects \(d_1\) and \(d_2\) with the same timestamp \(t\). Moreover, it may be the case that there are timestamps which do not occur in the stream; the latter is useful in particular for those situations where there is no information at some time point from \(T\)—and hence this representation is also useful to model varying frequencies of stream element occurrences.

The elements of the stream are thought to be arriving at the query answering system in some order. In a synchronized stream setting, one demands that the timestamps in the arrival ordering make up a monotonically increasing sequence. In an asynchronous stream setting, it may be the case that elements with earlier timestamps arrive later than data with later timestamps.

In the context of OBDA for streams, we will have to deal with two different domains \(D\) of streamed objects. The first domain consists of relational tuples from some schema; in this case we call the stream a relational stream. The second domain is made up by data facts, either represented as ABox assertions or as RDF triples.

3.2 The Sensor Measurement Scenario

The sensor measurement scenario is considered to be a typical application scenario for stream reasoning, and it can easily be embedded into the context of Linked Open Data and the Semantic Web. As an indicator for the relevance of this scenario for stream data management systems, one may consider the fact that weather report related sensor measurements are part of the SRBenchmark for streaming engines [102].

The general idea of reaching semantic operability for sensor measurements by representing sensor data and sensor related terminology in a formal logical representation format allowing for logical reasoning, resulted in a (nearly) standard ontology for semantic sensor networks. The ontology is authored by the members of
the W3C Sensor Network Incubator Group \cite{36} and it contains concepts and properties related to sensors, measurements, structures, observations, events etc. A general ontology of this kind can be extended for specific sensor measurement scenarios by introducing new names to the signature of the ontology and adding new ontology axioms. The signature of this ontology (i.e., the set of concept and role names) provides the interface through which the streamed data from the sensor can be queried. Axioms are used to capture the semantics of names relative to other names introduced in the signature, and thus, with axioms, the set of query results is extended. This also holds for continuous queries used in a streaming data processing scenario.

As a concrete example, we introduce in detail a sensor measurement scenario inspired by an industrial setting provided by SIEMENS \cite{54}. In this context, we discuss a gas turbine monitoring and control application. Few service centers are run in order to monitor, analyze, and control hundreds of turbines in various power plants. The storage system in the data center can be viewed as a central DB, which stores different types of information, the most relevant being static data on the one hand, such as turbine infrastructure data, as well as, on the other hand, time-stamped measurement data stemming from up to 2K sensors. In addition, so-called event data (“messages”) from control units are stored with timestamps indicating the creation time of the events. Control units are small computation units, seen as black boxes, getting input from sensors.

A (normalized) relational DB schema for the monitoring scenario is shown below for demonstration purposes. The SIEMENS Corporated Technology (CT) division identified it as a good representation of the schema used in the original central databases. For brevity, we ignore details such as attribute types (string, integer etc.). Keys are underlined, and foreign key specifications are introduced in the text. We assume that a foreign key \(A\) points to attribute \(A\) of the destination relation if not indicated otherwise.

The infrastructure data contains the description of sensors, their names, and their types, the components to which they are attached. \texttt{Sname} is a short name and \texttt{description} a longer description of the sensor in natural language. The foreign key \texttt{CID} of the schema \texttt{SENSOR} to the relation \texttt{COMPONENT} describes the component at which the sensor is directly attached. The type \texttt{TID}, a foreign key \texttt{TID} pointing to the \texttt{SENSORTYPE} relation, indicates whether they indicate temperatures, pressures, or rotations per time unit, or whether sensors are indeed control units, i.e., control units are seen as complex sensors in this simple relational data model. Some type indicator strings for \texttt{Tname} are introduced below.

\begin{verbatim}
SENSOR(SID, CID, Sname, TID, description)
SENSORTYPE(TID, Tname)
\end{verbatim}

The tree like structure of components is described in the component and assembly relations.

\begin{verbatim}
COMPONENT(CID, superCID, AID, Cname)
ASSEMBLY(AID, AName, ALocation)
\end{verbatim}

The \texttt{superCID} column specifies the next upper component at which the component with identifier \texttt{CID} is attached (foreign key to attribute \texttt{CID} of \texttt{COMPONENT} itself). The turbine assembly at which the component is (indirectly) attached is given by \texttt{AID} (foreign key from \texttt{COMPONENT} to \texttt{ASSEMBLY}).

There are two main types of timestamped data, one for measurements and the other for so-called events (also called event messages or messages for short). The former are represented by the schema

\begin{verbatim}
MEASUREMENT(MID, MtimeStamp, SID, Mval)
\end{verbatim}

where the key \texttt{MID} identifies a measurement having some timestamp \texttt{MtimeStamp} measured with a sensor \texttt{SID} (foreign key to \texttt{SENSOR}) and having a value \texttt{Mval}.

In the \texttt{MesEventText} column, event messages contain free text information about the assembly (turbine) status (with some some standardized formulations). Within the \texttt{catID} column describing a so-called message

\footnote{See also the final report at \url{http://www.w3.org/2005/Incubator/ssn/XGR-ssn-20110628/}.}

\footnote{A “global” reference \texttt{AID} of each component to its root assembly is a common pattern found in many database applications since plain SQL does not provide for recursion, needed to find the assembly with arbitrarily many hops up the hierarchy.}
category, a rough categorization of the event messages is given (catID is a foreign key to CATEGORY). Some messages contain warnings or failure hints, others indicate the operational status of the turbine (start up, running, stopping) and so on (attribute MesEventText). The assembly the message refers to is indicated by MesAssemblyID, a foreign key to the attribute AID of ASSEMBLY.

\[
\text{MESSAGE(MesID, MesTimeStamp, MesAssemblyID, catID, MesEventText)}
\]
\[
\text{CATEGORY(catID, catName)}
\]

In the original SIEMENS monitoring application the database is updated every 24 hours by processing some file containing all the traces of measurements from the sensors and control units within one day. This update granularity might be sensible for statistical and time series analysis that normally refer to a long time interval; but it is hardly appropriate for continuous queries when real time feedback is needed in acute critical situations. So rather than with a coarse update rate of 24 hours, the data is assumed to be updated with a higher frequency. With the new feature of continuous queries, new services can be provided to engineers, and we assume that this kind of “streaming” input can be provided in one way or another in industrial contexts such as turbine monitoring as well. Continuous queries are not only useful for online predictive diagnosis but can also be effectively used for event detection in a reactive diagnosis setting by “replaying” measurements, i.e., simulating NOW being advanced while starting in the past (maybe faster than in real-time). Different sets of conceptual domains models might be used during these “simulations”[^3]. Please note that ontology-based query answering is very important in this context because for different simulations one might use different ontologies without changing the queries manually.

This new architecture for complex event processing with continuous queries referring to streams of data is one of the streaming application scenarios where our temporal and stream query language comes into play. The streams are homogeneous relational streams, i.e., we have a stream of sensor measurements containing tuples adhering to the column schema of MEASUREMENT, and we have a stream of of tuples adhering to the column schema of MESSAGE. So, though it is more likely to have a stream of data for every sensor and every control unit, we assume here for ease of exposition only two streams of data corresponding to the two tables for measurements and messages, resp. These two streams are denoted in the following by $S_{\text{Msmt}}$, the stream of measurement data, $S_{\text{events}}$, the stream of event message data. These are streams in the classical sense—underlying also the CQL language [6], namely, homogeneous streams containing timestamped tuples of the same type (here: the same relation schema). So the data we have are on the one hand static data as well as temporal data (measurement and event data) stored in the central database, and, on the other hand, data arriving in a stream, one stream for measurements, the other stream for events.

It should be noted that in contrast to continuous queries, temporal queries can be indeed handled by standard database systems such as PostgreSQL. The query in Listing 3.1 shows how we retrieve monotonically increasing intervals in the sensor data for sensor ‘s0’ from the last day with PostgreSQL time functions.

First, we create a view (lines 1–7) containing time points at which the slope change from positive to negative. It uses the PostgreSQL window functions “lag” and “lead” in a tuple window for accessing the attributes in the previous and next rows (if no next or previous tuples exist, the default is set to 0), respectively, w.r.t. to the specified order (introduced with OVER). In the second step we create a view (lines 9–13) with all timestamps that stand for extrema. Finally, all increasing intervals from the second view are computed (lines 16–23) and returned as a result. The query is slightly simplified w.r.t. marginal timestamp values.

### 3.3 Lifting the Data to the Logical Level

Accessing data through the interface of an ontology presupposes a method to lift the data stored in a SQL database or arriving in a stream into the logical level of ontologies. In the classical OBDA setting the chosen method is that of declarative mappings, formally realized as rules with logical queries on the lefthand and SQL on the righthand.

[^3]: Imagine that for testing purposes, timestamps can be scaled, e.g., in order to simulate a higher data arrival pace.
In the SIEMENS sensor scenario, assume that the ontology signature contains a concept symbol `Sensor` and an attribute symbol `hasName`. The following mapping induces the set of ABox assertions stating which individuals are sensors and how they are named.

\[ \text{Sensor}(x), \text{hasName}(x, y) \leftarrow \text{SELECT f(SID) as x, Sname as y FROM SENSOR} \]

The lefthand side of the mapping is a conjunctive query and the righthand side is a SQL query in which all the variables of the CQ are used. The information in the row of the measurement table is mapped to unary facts (\(\text{Sensor}(x)\)) and binary atomic facts (\(\text{hasName}(x, y)\)). The term \(f(SID)\) denotes an individual constant, constructed as a functional term indicating the value of the attribute \(\text{SID}\) of the sensor. If the ontology language allows for datatypes or concrete domains—as we assume here—then we can use attribute values directly without the need of an additional functional symbol. This is demonstrated above for the column `Sname` containing strings.

As an example of a similar mapping we consider the mapping that determines all burner tip temperature sensors.

\[ \text{BurnerTipTempSensor}(x) \leftarrow \text{SELECT f(s.SID) as x} \]
\[ \text{FROM SENSOR s, SENSORTYPE t} \]
\[ \text{WHERE s.TID = t.TID} \]
\[ \text{AND t.Tname = 'burnertiptempsensor'} \]
The resulting set of ABox assertions induced by mappings of this form make up the logical view on the data. Mappings bridge the gap between the closed world of the DBs, where a relational DB stands for one interpretation, to the open world of DL logics, where an ABox may have many satisfying interpretations. Queries are formulated with respect to ABoxes and TBoxes, which may constrain the set of interpretations, and the answers are defined w.r.t. the whole set of interpretations (certain answer semantics).

The role of the TBox as a means to constrain the interpretations to the intended ones is demonstrated for the concepts of sensors and burner tip temperature sensors. In the minimal model, the mapping-induced extensions of both concepts should be such that all burner tip temperature sensors are also temperature sensors (for which we do not have a mapping, say) and also sensors. But there may be interpretations of the induced ABoxes which denote the extension of the concept BurnerTipTempSensor by a set containing objects that are not contained in the interpretation’s extension for Sensor. A TBox containing the axioms

\[
\text{BurnerTipTempSensor} \sqsubseteq \text{TempSensor}, \text{TempSensor} \sqsubseteq \text{Sensor}
\]

will disallow such interpretations of the ABox. For example assume that the mapping for burner tip temperature sensors generates (virtual) ABox assertions BurnerTipTempSensor(s0), BurnerTipTempSensor(s1), .... Then, the individual s0 is also an instance of TempSensor, or, to put it in other words, the TBox and ABox entail the assertion TempSensor(s0). So for a query asking for all temperature sensors, one has to use the intensional knowledge defined in the TBox to get all (correct) answers.

The perfect rewriting approach behind OBDA does not use classical reasoning on the TBox and ABox, but compiles a given query using the TBox axioms into a new query, which is evaluated directly onto the data and yields sound and complete answers. In our simple example the rewritten query would be just the query asking for all sensors that fulfill TempSensor(sensor) ∨ BurnerTipTempSensor(sensor). Such rewritings into first order formulas exist if the TBox uses lightweight description logics in the DL-Lite family [28], but they do not exist for more expressive logics. In particular, logics with transitive roles do not allow for FOL rewritability. Because some applications can also benefit from transitive roles and other features of description logics such a disjunction we also consider expressive logics such as SHI. That transitive relations are useful modelling means can be illustrated with the COMPONENT relation in the SIEMENS data model. The COMPONENT schema demonstrates the typical entity-relationship modelling methodology for representing chains or tree-like structures by a self-referencing foreign key; every component has a reference to a top component, which again has a reference to a top component, and so on until the upper most component on the level directly under the assembly level. A useful qualitative modelling of the component hierarchy relies on the part-of relation. A mapping, generating the direct part-of relation between components would be as follows:

\[
\text{partOf}(x, y) \leftarrow \text{SELECT } f(CID) \text{ AS } x, g(superCID) \text{ AS } y \text{ FROM COMPONENT}
\]

That the part-of relation is transitive can be declared by using a TBox axiom inducing further tuples in the part-of relation that are not directly visible in the SQL schema. So, if for example there are two ABox assertions partOf(comp1, comp2), partOf(comp2, comp3) induced by the mappings, then the transitivity condition entails partOf(comp1, comp3). A Boolean query w.r.t. the ABox and the extended TBox asking whether comp1 is a part of comp3 would thus have to be answered with yes. A similar query could not directly be formulated in pure SQL, which does not support recursion.

The axioms in the TBox have to be understood as stating that at every time point (!) every burner tip temperature sensor is a temperature sensor, similarly for temperature sensors and sensors. Our query semantics does not restrict the TBox expressivity to non-temporal classical DLs, but we will assume that the TBox is formulated in a pure description logic—without any additional temporal operators such as those in the temporal OBDA approach of [11]. The benefit of this assumption is that we may rely on existing reasoning and rewriting systems.

The mappings for temporal data in the central DB is managed by handling the timestamps as simple time attributes. And indeed, for describing measurements it makes sense to attach timestamps by an attribute
hasTime in the following classical mapping.

$$
\left\{
\begin{array}{l}
\text{measurement}(x), \\
\text{measHasVal}(x, y), \\
\text{hasTime}(x, z)
\end{array}
\right\} \leftarrow \text{SELECT } f(\text{MID}) \text{ AS m}, \text{ Mval AS y, MtimeStamp AS z}
\text{FROM MEASUREMENT}
$$

Assertions induced by these mappings are part of the static ABox: The assertions \text{measurement}(m_1), \text{measHasVal}(m_1, 90^\circ C), \text{hasTime}(m_1, 30s) hold at every time, and time is attached only by the attribute hasTime.

In some cases one wants to say that a fact holds at some time point (or time period), saying for example that a sensor has some value at that time point. In order to realize this idea, we follow the method of time tagging the assertions with timestamps. This method is similar to adding an extra time argument for concept and roles as in [11], or as tagging triples in RDF approaches such as the representation in Strabon [21]. As an example, we describe a mapping that gives the values y which a sensor x shows at time point (z).

$$
\text{hasVal}(x, y)(z) \leftarrow \text{SELECT } f(\text{SID}) \text{ AS x, Mval as y, MtimeStamp as z FROM MEASUREMENT}
$$

The ABox induced by this mapping is different from the static ABox, as its assertions contain time tags. We call it a \textit{temporal ABox}.

Quite similar to the mappings producing time tagged assertions we can construct mappings from relational streams to streams of time tagged assertions. In the mapping above, one just has to replace the reference to the table \textit{MEASUREMENT} by a reference to a named relational stream with measurements, say \textit{relS_{Msmt}} (cf. the stream mapping language S2O in [27]). To make it concrete, we assume that the relational streams are described in the stream relational query language CQL [6]. The stream mapping then would be

$$
\text{hasVal}(x, y)(z) \leftarrow \text{SELECT } R\text{stream}(f(\text{SID}) \text{ as x, Mval as y, MtimeStamp as z}) \text{ FROM relS_{Msmt [NOW]}}
$$

Using the default notation, the mapping above would be written just as

$$
\text{hasVal}(x, y)(z) \leftarrow \text{SELECT } f(\text{SID}) \text{ AS x, val AS y, timestamp AS z FROM relS_{Msmt}}
$$

The query language which will be introduced in the next section operates on the ontological level, its inputs being TBoxes, static ABoxes, temporal ABoxes and (not necessarily homogeneous) streams of ABox assertions. The stream of ABox assertions underlying most of the following examples is the measurement stream \textit{S_{Msmt}}. Its initial part, called \textit{S_{Msmt}^{\leq 5s}} here, contains timestamped ABox assertions giving the value of a temperature sensor \textit{s_0} at 6 time points starting with 0s.

$$
S_{Msmt}^{\leq 5s} = \{ \text{hasVal}(s_0, 90^\circ C)(0s), \text{hasVal}(s_0, 93^\circ C)(1s), \text{hasVal}(s_0, 94^\circ C)(2s), \text{hasVal}(s_0, 92^\circ C)(3s), \text{hasVal}(s_0, 93^\circ C)(4s), \text{hasVal}(s_0, 95^\circ C)(5s) \}$$
Chapter 4

The Query Language STARQL

Now let us see informally, how an appropriate query language for sensor data streaming scenarios could look like within the challenging paradigms of OBDA and ABDEO. We will describe the query language STARQL (Streaming and Temporal ontology Access with a Reasoning-based Query Language, pronounced Star-Q-L) on the abstract logical level, thereby assuming that the data in the databases and the relational streams have already been mapped to the logical level of TBoxes, static or temporal ABoxes, and ABox assertion streams.

STARQL combines elements from SQL, SPARQL, and from description logics. Instead of the logical notation of DLs we use the machine processable triple notation of SPARQL (http://www.w3.org/TR/rdf-sparql-query/). Moreover, in using SPARQL we can rely on its namespace handling mechanism (though we will skip most of the namespace declarations in the examples).

4.1 A Basic Example

For the ease of exposition let us first assume that the terminological TBox is empty. The engineer may be interested whether the temperature measured in the sensor $s_0$ grew monotonically in the last two seconds, i.e., in the interval $[\text{NOW} - 2s, \text{NOW}]$. We first present the solution in our new streaming language STARQL and use it to explain the main conceptual ideas, the most important one being that of ABox sequencing.

```
CREATE STREAM S_out AS
SELECT { s0 rdf:type RecentMonInc }<NOW>
FROM S_Msmt 0s<-[NOW-2s, NOW]->1s
SEQUENCE BY StdSeq AS SEQ1
HAVING FORALL i <= j IN SEQ1, x, y:
   IF ( { s0 hasVal ?x }<i> AND { s0 hasVal ?y }<j> )
   THEN ?x <= ?y
```

Listing 4.1: Basic STARQL example

The solution is centered around a window of range 2s (see Listing 4.1 line 4). Every 1 second, which is determined by the slide parameter and denoted above by $\rightarrow 1s$, the window moves forward in time and gathers all timestamped assertions whose timestamp lies in the interval $[\text{NOW}-2s, \text{NOW}]$. Here, \text{NOW} denotes the current time point. The development of the time has to be specified locally in the query (not shown here) or directly for all queries by some pulse function as follows:

```
CREATE PULSE AS START = 0s, FREQUENCE = 1s
```

We will follow in our exposition this synchronized approach.
The slack parameter in this and all the following example queries is set to be 0s meaning that the queries do not handle out of order stream assertions. The notation \( < - \) hints to the meaning of the slack as a temporal widening of the window back into the past.

At every time point, the window content is a set of timestamped assertions which together make up a temporal ABox. For the first two time points 0s, 1s we do not get well defined intervals for \([\text{Now} - 2s, \text{Now}]\), but it is natural to declare the contents at 0s and 1s as the set of timestamped ABox assertions which have arrived up to to second 0 resp. 1. For the other time points, we have proper intervals, and so the resulting temporal ABoxes from 0s to 5s are defined as follows.

<table>
<thead>
<tr>
<th>Time</th>
<th>Temporal ABox</th>
</tr>
</thead>
<tbody>
<tr>
<td>0s</td>
<td>{\text{hasVal}(s_0, 90\degree C)</td>
</tr>
<tr>
<td>1s</td>
<td>{\text{hasVal}(s_0, 90\degree C)</td>
</tr>
<tr>
<td>2s</td>
<td>{\text{hasVal}(s_0, 90\degree C)</td>
</tr>
<tr>
<td>3s</td>
<td>{\text{hasVal}(s_0, 93\degree C)</td>
</tr>
<tr>
<td>4s</td>
<td>{\text{hasVal}(s_0, 94\degree C)</td>
</tr>
<tr>
<td>5s</td>
<td>{\text{hasVal}(s_0, 92\degree C)</td>
</tr>
</tbody>
</table>

Now we have at every second a set of timestamped assertions. In order to apply ABox and TBox reasoning we group these assertions together into (pure) ABoxes. The result of the grouping is a finite sequence of ABoxes (ABoxes are sequenced w.r.t. to the order of their timestamps). We can use this sequence of classical ABoxes to filter out those values \( v \) for which it is provable that \( \text{hasValue}(s_0, v) \) “holds”.

The ABox sequencing operation is introduced by the keyword \texttt{SEQUENCE BY} in the query (line 5). There may be different methods to get the sequence, the most natural one is to merge all assertions with the same timestamp into the same ABox. The query above refers to this built-in sequencing method \texttt{StdSeq}. Other sequencing methods may be defined by following an SQL-like create declaration (for details see the next chapters).

In our simple example, the standard ABox sequencing leads to simple ABoxes, as the stream does not contain ABox assertions with the same timestamp. Please note that ABoxes in a sequence can be combined with larger static ABoxes (see below). Just for illustration, we note that the ABox sequence at time 5s is defined as follows:

\[
\begin{array}{|c|l|}
\hline
\text{Time} & \text{ABox sequence} \\
\hline
5s & \{\text{hasVal}(s_0, 92\degree C)|3s\}, \{\text{hasVal}(s_0, 93\degree C)|4s\}, \{\text{hasVal}(s_0, 95\degree C)|5s\} \\
\hline
\end{array}
\]

The sequence at 5s contains the timestamped ABoxes, e.g., the first one is the ABox \{\text{hasVal}(s_0, 92\degree C)\} with timestamp \langle 3s \rangle. At the writing of this deliverable, STARQL does not refer to the timestamps of the ABoxes in the sequence, but only to a natural number indicating its ordinal position within the sequence. Hence the actual sequence at time point 5s is the following:

\[
\begin{array}{|c|l|}
\hline
\text{Time} & \text{ABox sequence} \\
\hline
5s & \{\text{hasVal}(s_0, 92\degree C)|1\}, \{\text{hasVal}(s_0, 93\degree C)|2\}, \{\text{hasVal}(s_0, 95\degree C)|3\} \\
\hline
\end{array}
\]

Now, as there is at every time point a sequence of ABoxes, one can refer, at every time point, to every (pure) ABox within the sequence in order to apply DL reasoning. This is actually done in the query above within a so-called \texttt{HAVING} clause (Listing 4.1 line 6). Here, the \texttt{HAVING} clause is a boolean expression. In general, it may be some predicate logical formula with open variables. (For the details see Chapter 5).

The evaluation of the expression \{ \texttt{s0 hasVal ?x \langle 1 \rangle} \} (or \texttt{hasVal(s0, x)\langle i \rangle} in DL notation) in Listing 4.1 relies on DL deduction: Find all \( x \) that can be proved to be a filler of \( \text{hasValue} \) for \( s_0 \) w.r.t. to the ABox \( A_i \). A TBox and other (static) ABoxes used for deduction can be mentioned in a \texttt{USING} clause (see below). Similarly, \{ \texttt{s0 hasVal ?y \langle j \rangle} \} means that one wants to find all values \( y \) which are provably recorded in sensor \( s_0 \) w.r.t. to the \( j \)th ABox in the sequence. The resulting values \( x, y \) must be such that the \( y \) (recorded in the later ABox \( j \)) must be larger than the former.

The result of the query in Listing 4.1 is a stream of ABox assertions of the form \( \text{RecentMonInc}(s_0)\langle t \rangle \), defined by the form being specified after the \texttt{SELECT} keyword. The timestamp template \texttt{<NOW>} indicates
that the assertions for the output stream are timestamped with the evolving time parameter of the sliding window. Our language also allows for more complex timestamping functions such as $<\text{NOW}+1>$.

For the first seconds until 5s the output stream is defined as follows:

$$S_{\text{out}}^{\leq 5s} = \{\text{RecMonInc}(s_0)(0s), \text{RecMonInc}(s_0)(1s), \text{RecMonInc}(s_0)(2s), \text{RecMonInc}(s_0)(5s)\}$$

So, the query correctly generates assertions saying that there were recent monotonic increases according to the query for time points 0s, 1s, and 2s, and then again only at time point 5s.

Already in this simple query example, we see possible optimizations that should be taken into account by the query execution planner. We observe that there is no 2s-lasting monotonic increase at time point 3s. Then we know a priori that there cannot be monotonic increases at time points 3s and 4s. Hence the query planner will reformulate the query such that it does not need to completely build the ABox sequence and the monotonicity testing at time points 3s and 4s.

A slightly more complex query than the one above asks for all sensors that grew monotonically in the last 2s. Assuming that the stream $S_{Msmt}$ now may contain sensor values for many different sensors, we would just have to rewrite the query from above by substituting the constant $s_0$ by a variable $sens$. As the monotonicity condition could be useful for other different queries, our language also provides the possibility to define names for HAVING predicates. We call these definitions "aggregators". The modified query and the aggregator definition are demonstrated in Listing 4.2.

```starql
CREATE STREAM S_out_2 AS
SELECT {? sens rdf:type RecentMonInc}<NOW>
FROM S_Msmt 0s < -[NOW-2s, NOW]->1s
SEQUENCE BY StdSeq AS SEQ1
HAVING monInc(SEQ1, ?sens hasVal *)

CREATE AGGREGATE OPERATOR monInc(seq, f(*)) AS
FORALL i <= j in SEQ, x,y:
  IF (f(x)<i> AND f(y)<j>)
  THEN x <= y
```

Listing 4.2: STARQL example for aggregation definition

The aggregation operator $\text{monInc}(\cdot, \cdot)$ gets two arguments, the first being a sequence of ABoxes and the second being a boolean functional in one open argument, i.e., an expression standing for a term with one argument, which if substituted results in a boolean term. In the case of the new query, the functional is given by $?\text{sens hasVal *}. Here, the variable $?\text{sens}$ is thought to be already bound. The open argument is marked by a star *. Please note that $\text{monInc}(\cdot, \cdot)$ is just one example. Many built-in aggregation operators will be available in the final version of STARQL.

### 4.2 Dealing with Indefiniteness

The monotonicity condition in the examples above is just one of the many different ways in which the user may formulate monotonicity conditions as aggregation operations on the sequence of ABoxes. Though the underlying monotonicity condition seems to be the usual mathematical monotonicity condition, there are some subtleties due to the open world assumption underlying ontology-based representation approaches.

As is the case for many applications scenarios such as the one of SIEMENS, we have to deal with unclean, corrupted data leading to inconsistent or indefinite knowledge. Regarding the sensor measurements we may
have the following problem of one sensor showing different values at the same time. For instance, there may be two values for the sensor $s_0$ at time point $2s$.

\[
\{ \ldots \text{MEASUREMENT}(1, 2s, 's_0', 93^\circ C), \text{MEASUREMENT}(2, 2s, 's_0', 95^\circ C) \ldots \} 
\]

Such a situation is not unrealistic, as demonstrated by some non-normalized data set of SIEMENS: sensors of the same type and roughly the same location on the turbine were denoted by a single sensor name.

Clearly, this problem like many others could be solved by a preprocessing step on the data, leading to a new clean (normalized) data set, which is then mapped to virtual ABoxes and virtual streams. Following the general idea of the OBDA paradigm, according to which one does not touch the data directly, one would define mappings that simulate the cleaning. In the context of the small example, one could formulate a mapping with a SQL source query that selects all measurements for a given sensor (here) $s_0$ at the same time point (here 2s) and, e.g., calculates its mean value. The outcome is then stipulated as the value for the sensor.

\[
\text{hasVal}(x, y) \langle z \rangle \leftarrow \text{SELECT f(SID) AS x, AVG(val) AS y, timestamp AS z FROM MEASUREMENT GROUP BY SID, timestamp}
\]

A drawback of this mapping centered approach is the fact, that the diagnostic engineer, who does not have access to the mappings, cannot control the corrections by the mappings, thereby limiting his flexible modelling view on the ontological level. Hence, we design our query language in a way that allows the diagnostic engineer to define his own way of dealing with the problem of multiple values. The solution chosen in $\text{monInc}$ is to demand that all values at some early time point have to be smaller than all values at later time points. But we could equally have defined a monotonicity condition $\text{monIncMean}$ using FOL with arithmetics to mimic the aggregation in the mapping above.

The second type of problem with sensor measurements is that there may be no value at all for some sensor at a given time point in the stream of measurements, hence the virtual ABox stream may not contain an assertion $\text{hasValue}(s_0, v_{123}) \langle t_{123} \rangle$ for a time point $t_{123}$, at which one expects a sensor value. In the OBDA view this means, that one cannot prove whether the sensor shows some value (assuming that the TBox is empty) and what it is exactly. Hence, there may be models of the temporal ABox at second $t_{123}$ (and the TBox) in which the sensor is not assigned a value at all. The $\text{monInc}$ predicate does not incorporate those cases of unknown values. It follows a told value approach, considering only those values as input for the monotonicity macro, for which there is indeed a concrete value for the sensor.

### 4.3 Embedded Filter Queries and Entailment

Until now, in all examples for STARQL we focused on temporal ABoxes only, and did not refer to TBoxes and the entailment relation w.r.t. them. In this section, we demonstrate the effects of TBoxes and their use with a simple example extending the ones before.

Suppose that an engineer is interested in a subclass of sensors that grew monotonically in the last seconds, namely the subclass of temperature sensors. The SIEMENS data give more information regarding the sensors; so for example, there are temperature sensors which are attached to some components, such as burner tip temperature sensors. As described in the section on the SIEMENS data model, we assume first that the TBox $\mathcal{T}$ contains the axioms

\[
\text{BurnerTipTempSensor} \sqsubseteq \text{TempSensor}, \text{TempSensor} \sqsubseteq \text{Sensor}
\]

and that the mappings generate an ABox assertion $\text{BurnerTipTempSens}(s_0)$, among others. Further, we assume that there is no mapping for temperature sensors, and therefore we use the TBox to model the relationship between the concept names $\text{BurnerTipTempSensor}$ and $\text{TempSensor}$.

Now, we formulate the query asking for those temperature sensors, that grew monotonically in the last 2s (Listing 4.3). Within the $\text{WHERE}$ clause (line 6) we can specify relevant conditions. Here it is just the
A Semantics for Temporal and Stream-Based Query Answering in an OBDA Context

Listing 4.3: STARQL example for a USING clause

```sql
CREATE STREAM S_out_3 AS
SELECT {? sens rdf:type RecentMonInc}<NOW>
FROM S_Msmt Os -<[NOW-2s, NOW]>-1s
USING STATIC ABOX <http://optique.project.ifi.uio.no/Astatic>,
TBOX <http://optique.project.ifi.uio.no/TBox>
WHERE { ?sens rdf:type TempSensor }
SEQUENCE BY StdSeq AS stateSequence
HAVING monInc(stateSequence, ?sens hasVal *)
```

condition TempSensor(sens) asking for temperature sensors. As in SPARQL, it is also possible to specify OPTIONAL and FILTER conditions.

The evaluation of the condition in the WHERE clause and also in the HAVING clause incorporates not only the ABoxes in the generated sequence but also the TBox and the static ABox. The incorporation of these resources is specified in the query with the keyword USING. So, if $A_i$ is an ABox in the sequence, then the relevant local knowledge base at position $i$ in the sequence w.r.t. which the filters are evaluated is given by $KB_i = T \cup A_{\text{static}} \cup A_i$. And indeed, in order to identify the sensor $s_0$ in the input stream as a temperature sensor, one has to incorporate the fact from the static ABox stating that $s_0$ is a burner tip temperature sensor, and one has to incorporate the subsumption from the TBox, in order to conclude that $s_0$ is a temperature sensor.

A static ABox is a special case of an (infinite) temporal ABox, where all assertions have timestamps for all time points. Our query language also allows the incorporation of (historical) temporal databases that, e.g., hold past measurements and past events. Such a temporal DB is represented just as a finite set of timestamped ABox assertions $A_{\text{hist}}$. Now, the sequentializing can also be applied to such a historical ABox, so that the knowledge bases $KB_i$ w.r.t. which one issues the filter queries are given by the TBox, the current ABox $A_i$ resulting from the stream, the static ABox $A_{\text{static}}$ and the ABox $(A_{\text{hist}})_i$ resulting from the temporal ABox. Spatial resources can be incorporated as a static ABox.

In general, the filter conditions in the WHERE clause may be more complex, and in fact, for the two instantiations of our semantical framework (one for classical OBDA and the other for ABDEO) we consider two different query classes for the filter conditions. The first class are unions of conjunctive queries; this query filter language goes together with any member of the DL-Lite family that guarantees FOL rewritability. In particular, we will look at DL-Lite with concrete domains in order to be able to represent sensor values (and timestamps). The second class is the weaker class of grounded conjunctive queries, but the TBox language may have high expressivity up to the DL language $SRIQ$. This combination of query language and TBox language has been shown to allow for a special form of ABox modularization with which answering queries to the whole ABox can be reduced to answering the queries on rather small modules (the results of [74] are for $SHI$ but can be extended in a straightforward way to $SRIQ$).

Modularization is also relevant for evaluating queries in our query language, which may incorporate big (static) ABox by the keyword USING. The ABoxes generated by the sequencing operator within STARQL are small in general, hence the local knowledge bases $KB(i)$ referred to implicitly in the HAVING clause can be thought to consist of a TBox, a small ABox module and the temporal ABox generated by the sequencing. Modules for the static part have to be combined with small ABoxes defined by sequencing operators.

Let us give an example with a more complex condition in the HAVING clause using a further constructor and the average operator (Listing 4.4). We are interested in those temperature sensors with a recent monotonic increase which are known to be in an operational mode (not service maintenance mode) in the last 2s. Furthermore we want to get the average of the values in the last 2s. The query demonstrates that we may leave out the sequence argument if there is only one sequence. Next to the monotonicity condition we have
CREATE STREAM S_out_4 AS

SELECT { ?sens rdf:type RecentMonInc }<NOW>,
    mean(AVG(?sens hasVal *))<NOW>
FROM S_Msmt 0s<-[NOW-2s, NOW]->1s
USING STATIC ABOX <http://optique.project.ifi.uio.no/Astatic>,
    TBOX <http://optique.project.ifi.uio.no/TBox>
WHERE { ?sens rdf:type TempSensor }
SEQUENCE BY StdSeq as stateSequence
HAVING monInc(?sens hasVal *) AND
    FORALL i IN stateSequence { ?sens rdf:type InOperationalMode }<i>

Listing 4.4: STARQL example for complex filter condition

a further condition using the \texttt{FORALL} operator. It evaluates the boolean condition in its scope on all ABoxes in the sequence and outputs the truth value true if the condition holds in all ABoxes.

Even in the case of lightweight description logics such as DL-Lite, the TBox can contain constraints that lead to inconsistencies with ABox assertions. In scenarios as that of the SIEMENS use case such forms of inconsistencies are seldom; they can occur, for instance, through disjointness axioms. A practically more relevant source for potential inconsistencies are functionality axioms, which are directly connected to the problem of multiple sensor values from above. An example for a functionality axiom is the following one

\[(\text{funct hasVal})\]

It states that at every time point there can be at most one filler of the role hasVal in a particular state. Note that we assumed that the TBox holds universally for all time points in the time domain and does not state any conditions on the development of concepts and roles w.r.t. different points in time. So, surely the sensor may have different values and different time points. If the functionality declaration is not fulfilled in an ABox, then the whole KB becomes inconsistent, with strange consequences for query answering.

The general OBDA approach does not handle inconsistencies by repairing or revising the ABox (or even the TBox), but gives a means for detecting inconsistencies. Inconsistency testing is reduced to query answering for an automatically derived specific query, and thus, inconsistency checking can in principle be done by SQL engines as well.

The TBox has also effects on the modes of indefiniteness regarding the knowledge of values of a sensor. For example, the TBox may say that every sensor has a value (at every time point), formalized as

\[\text{Sensor} \sqsubseteq \exists \text{hasVal}\]

If there is a missing value for a sensor at some time point \(t_{123}\), then we know there is a value due to the above TBox axiom. However, we have indefinite knowledge since the value is not known. Such a TBox axiom could be useful for modeling a notion of trust in the sensors’ reliability (it shows a value at every time), but on the other hand skepticism regarding the reliability of the channels through which the sensors’ readings are delivered. Incorporating such unknown values into further processing steps, such as, e.g., counting or other forms of aggregation is known to lead either to implausible semantics or to high complexities \cite{60}. Hence, in our query language the aggregation operators will work only with told-values similar to the epistemic approach of \cite{29}. So, for all knowledge bases \(KB_i\) in the ABox sequence we demand that they entail hasVal\((s_0, v_0)\) for some value constant \(v_0\). This is the case when hasVal\((s_0, v_0)\) is directly contained in the ABox or implied with some other ABox axiom and TBox axiom. The latter is, e.g., the case because of the existence of a role inclusion hasTempVal \sqsubseteq hasVal and of the ABox assertion hasTempVal\((s_0, v_0)\).
4.4 Query Orthogonality and the Scope of Assertions

The STARQL stream query language fulfills the desirable orthogonality property as it takes streams of timestamped assertions as input and produces again streams of timestamped assertions. This approach is motivated by the idea that getting answers to queries is not only an activity of GUI client programs (it is not only variable bindings that one wants to determine by queries), but query outcomes are going to be used as input to other queries as well as the generation of (temporal) ABox assertions in the application scenario itself. The expressions following the SELECT clause are templates for generating further timestamped ABox assertions (based on the intended interpretations within the query). The produced ABox assertions hold only within the output stream in which they are generated—and not universally. Otherwise we would have to handle recursion in queries—which, though possible using some kind of fixpoint operator, might lead to bad performance due to theoretically high complexity of the query answering problem. Hence the stream of ABox assertions generated by a query is in a different “category” than the assertions in the static ABox and the historical ABox.

Though the ABox assertions are limited to hold in the output streams, they may interact with the TBox, leading to entailed assertions. Assume that the TBox contains the following axiom stating that a sensor with a recent monotonic increase is a sensor in a critical mode:

\[ \text{RecMonInc} \sqsubseteq \text{Critical} \]

The engineer could ask for those temperature sensors in a critical mode at every second on the stream \( S_{out3} \) generated by one of our queries above. The components at which the sensors have to be removed in at most \( 10^4 \) seconds are declared as those ones that have been removed in the next service maintenance (see Listing 4.5).

```
CREATE STREAM S_out_5 AS

SELECT { ?component removeDueToSensor } ?sens < NOW + 10^4s>
FROM S_out_3 0s <= [NOW, NOW] -> 1s
USING STATIC ABOX <http://optique.project.ifi.uio.no/Astatic>,
TBOX <http://optique.project.ifi.uio.no/TBox>
WHERE { ?sens rdf:type TempSensor . ?sens partOf ?component }
SEQUENCE BY StdSeq AS stateSequence
HAVING EXISTS i IN stateSequence { ?sens rdf:type Critical }<i>
```

Listing 4.5: STARQL example for scope locality

The query is evaluated on the stream \( S_{out3} \) which contains assertions of the form \( \text{RecMonInc}(\text{sens})(t) \). At every second only the actual assertion is put into the temporal ABox (window range = 0s) so that the sequence contains only a trivial sequence of length 1 (at most). The EXISTS operator just tests whether the condition \( \text{Critical}(\text{sens}) \) holds in some ABox in the sequence (which is of length 1 here). Now, the stream \( S_{out3} \) does not contain any ABox assertions with the concept symbol \( \text{Critical} \), but such assertions are entailed by assertions of the form \( \text{RecMonInc}(\text{sens})(t) \) in \( S_{out3} \) and the TBox. Hence, the query above will indeed find those components that have to be removed due to a critical sensor. The example might use oversimplification but the reader should be able to understand the main idea.

Changing the stream from \( S_{out3} \) to \( S_{Msmt} \) and thereby keeping everything else the same, will lead to a query which does not find any components—as the TBox and the assertions in \( S_{Msmt} \) do not entail assertions of the form \( \text{Critical}(\text{sens}) \). Hence, the answers of a query really depend on the streams to which the queries are issued.

The general motivation of this approach is similar to the CONSTRUCT operator in the SPARQL query language, which provides the means to prescribe the format in which the bindings of the variables should
be outputted. But in contrast, our approach also considers the outputs as ABox assertions that are part of an ontology. The idea of viewing the query head as a template for general ABox assertions was described already in the query language nRQL \cite{101} coming with the Racer system.

4.5 Multiple Streams

Many interesting time series features have to combine values from different streams. In the SIEMENS scenario, this could be the combination of values from different measurement streams, or the combination of the measurement streams with the event streams in order to detect correlations. The simplest combination is that of a union of streams which is directly provided in the same comma-separated notation as in pure SQL. So if one has two different measurement streams and one is interested in the recent monotonic increase, then the query would just be formulated as in Listing 4.6.

```
CREATE STREAM S_out_6 AS

SELECT { ?sens rdf:type RecentMonInc }<NOW>
FROM S_Msmt_1 0s<-[NOW-2s, NOW]->1s,
S_Msmt_2 0s<-[NOW-2s, NOW]->1s
WHERE { ?sens rdf:type TempSensor }
SEQUENCE BY StdSeq AS stateSequence
HAVING monInc(stateSequence, ?sens hasVal *)
```

Listing 4.6: STARQL example for combining multiple streams

The semantics of the union is given by pure set union of the temporal ABoxes generated by the window applications. But in many cases, the simple union of the ABoxes is not an appropriate means, because the timestamps of the streams might not be synchronized, so the simple idea of joining ABoxes with the same timestamp may lead to an ABox sequence that is well defined but pragmatically irrelevant.

For example, if some temperature sensor is recorded to show some value at time 2005-01-01 8:35h 12s whereas a different (pressure) sensor shows some value at a slightly different time 2005-01-01 8:35h 33s, then the engineer may have an interest in converting these assertions into the same ABox: Because, only if the values of both sensors fulfil some conditions “at the same time” (read as “in the same ABox”) will the engineer be able to infer additional knowledge. For example, the engineer could formulate a rule (formally to be represented either by a TBox axiom or by a query) saying that if the pressure value is bigger than the temperature value at the same time, then the turbine is considered to be in a critical mode.

\[
\text{hasVal}(s_T, v_1) \land \text{attachedAt}(s_T, \text{turb}) \land \\
\text{hasVal}(s_P, v_2) \land \text{attachedAt}(s_P, \text{turb}) \land \\
v_2 > v_1 \rightarrow \text{crit(turb)}
\]

Hence the query language envisioned to be used for such scenarios should provide means to define a merging operator based on some granularity parameter, on an equivalence relation, or, in its most general form, based on a similarity relation on the timestamps. The similarity relation then makes it possible to find consequences for conditions regarding the long run (rough granularity) or conditions on the short term (fine granularity) etc. So, in the example above, the idea could be to use a floor operator (granularity: only up to minutes), so that the timestamps 2005-01-01 8:35h 12s and 2005-01-01 8:35h 13s are considered to denote the same time—which may be the (rougher) timestamp 2005-01-01 8:35h.

Clearly, it may be possible to formulate the rules directly by reference to time intervals and use a tolerance threshold within the rules. But we decided to stick to “timeless” representation of rules and axioms in order to fruitfully use existing rule, reasoning, and/or rewriting systems.
In many cases, the elements in a stream could be considered to be homogeneous in the sense that the elements have a special form. In DL language, it could be the case, that all elements in a given stream are ABox assertions of the form R(x, y) (with perhaps a timestamp), where x, y are instantiated by constants, whereas in another stream all elements are of the form A(x) etc. But this may not necessarily be the case, hence we chose to work directly with streams of ABoxes.

Why don’t we just put everything into one single ABox stream and declare the semantics w.r.t. to this stream alone? The reason is that we want to be flexible in distinguishing between the things we get from different sources—following the scope locality principle discussed above.

Our query language gives the flexibility to implement different merges. This can be done due to the fact that STARQL offers the possibility to arbitrarily assign timestamps to the assertions templates after the select clause. For example, assume that the time evolvement is given by steps of 0.01s duration. A stream of measurements with granularity up to 0.01 seconds is coarsened to a stream of measurements with granularity up to 0.1 seconds as demonstrated in Listing 4.7. Here we use the floor operator \lfloor \cdot \rfloor written as |_ _|.

```
CREATE STREAM S_out_7 AS
SELECT { ?sens hasVal ?v } |_ NOW * 10 _| * 0.1 >
FROM S_Msmt 0s < -[NOW, NOW] - >0.01 s
WHERE { ?sens hasVal ?v }
SEQUENCE BY StdSeq
```

Listing 4.7: Example for coarsening

This type of coarsening can also be applied to another stream and then one may use the standard sequencing to realize a merge into a sequence of ABoxes.

Last but not least we would like to mention that STARQL also provides means for directly defining a non-standard sequencing operation, which can be referred to within the `SEQUENCE BY` clause of a query. Next to the standard sequencing, the query language also provides the predefined sequencing operation based on typical similarity/equivalence relations on timestamps. The similarity relations have to fulfill the additional property of obeying the time ordering, so that the similarity classes can be considered as focussing-out operations on the timestamps.

### 4.6 Temporal Reasoning in STARQL

The challenges of streaming scenarios demands a general framework structure focussing mainly on streaming aspects rather than on temporal aspects. Nonetheless, there are at least two links between STARQL and temporal reasoning.

The first link is the fact that STARQL reduces to a simple form of temporal reasoning under some restrictions on the window operator. The interval notation for the window operator on streams already hinted in a relevant connection to valid time reasoning on intervals in the spirit of \[75\]. Namely, if the slack and the slide parameter of the window operators are dropped (meaning: setting slack onto 0 and setting slide to the pulse granularity), then the window operator reduces to a simple method of interval reasoning. More concretely, with such a window one can ask whether there exists a time point in the given window (interval) for which such-and-such condition (specified somewhere else in the query) holds.

The second connection to temporal reasoning is the fact that STARQL embeds it within the ABox sequence data structure. The sequence of ABoxes can be considered to be a sequence of states which are described by ABoxes; hence typical temporal operators such as “at the next state in the future” or “at some state in the past”, which are included in temporal logics such as LTL, can be used for doing filtering, querying, or testing properties. Our sorted first order language used in the HAVING clause of STARQL queries allows temporal reasoning in the sense that it does filtering, querying and property testing on the basis of indexed
query patterns, thereby allowing to refer to the indexes and comparing them. So, one can for example say that for all indexes at which some property hold there is a later index at which another property holds.
Chapter 5

Formal Syntax and Semantics

Following the informal introduction of STARQL in the previous chapter, this chapter gives a detailed formal treatment of its syntax and semantics.

5.1 Syntax

A STARQL query set is composed of create expressions \((createExp)\), followed by declarations. Because declarations implement an extended feature, we describe their syntax at the end.

STARQL contains different CREATE expressions, one for streams, another for aggregation operators, and a third for a default pulse. The create expression for streams have an optional pulse declaration expression which overrides the global pulse declaration.

\[
createExp \rightarrow \text{CREATE STREAM } \text{streamName} \text{ AS} [\text{pulseExp} | \text{selectExp}]
\]

\[
\text{CREATE AGGREGATE OPERATOR } \text{unaryMacroName(seqName, starPattern)} \text{ AS } \text{UnaryAggDef} | \text{CREATE AGGREGATE OPERATOR } \text{binaryMacroName(seqName, starPattern, starPattern)} \text{ AS } \text{BinaryAggDef} | \text{pulseExp}
\]

\[
pulseExp \rightarrow \text{CREATE PULSE AS START = } \text{startTime}, \text{ FREQUENCE = } \text{frequency}
\]

In later versions of STARQL we are also going to introduce creation expressions for user defined sequencing methods. Until then we will only provide a standard sequencing method and a parameterized family of predefined sequencing methods.

The main outer structure of STARQL queries, which are described by the select expressions \(selectExp\), resembles that of SQL and is given below.

\[
selectExp \rightarrow \text{SELECT } \text{selectClause}\(\vec{x}, \vec{y}\) \text{ FROM } \text{listOfWindowedStreamExp} \text{[USING } \text{listOfRessources]} \text{ WHERE } \text{whereClause}\(\vec{x}\) \text{ SEQUENCE BY } \text{seqMethod} \text{[HAVING } \text{havingClause}\(\vec{x}, \vec{y}\)]}
\]

We will discuss the keywords in this query structure—starting with the WHERE and HAVING clauses which use embedded filter queries on DL (SPARQL) knowledge bases. We consider two instances of the filter queries, first with full unions of conjunctive queries (UCQs) in combination with the corresponding TBox
The syntax of STARQL assumes that a DL signature $\text{Sign}_{DL}$, on which the embedded queries and the ABoxes/TBoxes rely, is given. More concretely, we assume that there is a signature containing concept symbols, role symbols, attribute symbols, individual constants and value constants. The query language will add to this signature names for streams and higher-order predicates for the macro definitions.

Let $\Phi(\vec{x})$ be a UCQ resp. a GCQ formula over $\text{Sign}_{DL}$ with free (distinguished) variable $\vec{x}$. Using the same notational convention as in Chapter 4 on STARQL, these queries are given by triple templates in SPARQL notation. For the details we refer the reader to the W3C syntax definitions [43x600]http://www.w3.org/TR/rdf-sparql-query/.

$$\text{whereClause}(\vec{x}) \rightarrow \Psi(\vec{x})$$

The query body formulas in the WHERE clause are allowed to have exactly those distinguished variables that are mentioned in the SELECT clause. This is a safety condition without which one would have to define how to generate new objects—a feature that we are going to introduce and study in later phases of the OPTIQUE project. As the rule above demonstrates, we use the extended grammar facility of denoting with the same meta variable names the same variables in the whole rule, i.e., in the example above, the $\text{whereClause}(\vec{x})$ denotes an expression having exactly the variables $\vec{x}$ from $\Psi(\vec{x})$ (as open variables). Though not mentioned here, we can safely assume that the WHERE clause may have an optional FILTER clause as in SPARQL.

The HAVING clause opens a context on which aggregation and filter conditions on a sequence of ABoxes are formulated. The ABox sequence $seq$ is always finite (as the streams are isomorphic to the natural numbers), hence there is a finite interval $I(seq)$ of natural numbers with the usual ordering, which represents the order of timestamps within the window. We loosely use $i \in seq$ instead of $i \in I(seq)$. The having clauses now are constructed as a three-sorted FOL with embedded UCQs resp. GCQs. The three sorts are first the interval $I$, the concrete domain (for values), and the domain of abstract objects.

Let $\Psi(\vec{x}, \vec{y})$ denote a UCQ resp. a GCQ formula with free (distinguished) variable sets $\vec{x}, \vec{y}$. The set of variables $\vec{x}$ are those variables directly used after the SELECT keyword and bounded to constants within the WHERE clause. The set of variables $\vec{y}$ are variables to be used in the HAVING clause as distinguished variables.

There are different types of atoms allowed in the HAVING clause. The first group is made up by embedded filter queries and an index of the sequence. The second group contains atoms over the concrete domain values. The third group are arithmetic atoms over the indexes. Moreover, there are atoms using macro predicates. We deviate a little bit from the context free representation by allowing for co-occurrences of the same variables on the lefthand side and the righthand side of a grammar rule. An atom $\text{havingAtom}(\vec{x}, \vec{y})$ is one containing the variables in $\vec{x}, \vec{y}$. The reason for using different variable lists is due to their different roles within a SELECT expression.

$$\text{havingIndexedAtom}(\vec{x}, \vec{y}, i) \rightarrow \Psi(\vec{x}, \vec{y}) < i>$$

$$\text{havingValueAtom}(\vec{x}, \vec{y}) \rightarrow \text{valTerm}_1 < \text{valTerm}_2 | \text{valTerm}_1 <= \text{valTerm}_2 |$$

$$\text{valTerm}_1 = \text{valTerm}_2$$

$$\text{havingIndexArithAtom}(i_1, i_2) \rightarrow i_1 = i_2 | i_1 < i_2 | i_1 <= i_2$$

$$\text{havingMacroAtom}(\vec{x}) \rightarrow \text{unaryMacroName} ( \text{seqName}, \text{starPattern}(\vec{x}) ) |$$

$$\text{havingMacroAtom}(\vec{x}_1, \vec{x}_2) \rightarrow \text{binaryMacroName} ( \text{seqName}, \text{starPattern}(\vec{x}_1), \text{starPattern}(\vec{x}_2) )$$

$$\text{arithAggr} \rightarrow \text{COUNT} | \text{AVG} | \text{SUM} | \text{MIN} | \text{MAX}$$
Value terms are variables in the sort of the domain and contained in one of the list of variables $\vec{x}, \vec{y}$, or they are value constants. The variables $i_1$ and the $i_2$ stand for indexes in the sequence. The expression $\text{unaryStarPattern}(\vec{x})$ stands for a SPARQL triple pattern where at least in one triple the first argument or the third argument is replaced by $\ast$; the substitution must be type coherent. The subclass of unary star patterns $\text{unaryValueStarPattern}$ are constructed by replacing value positions (of the same data type) by $\ast$.

The $\text{havingClause}$ filter conditions are built on the basis of the $\text{havingAtoms}$ as FOL formulas using them as subformulae. Here we use the abbreviation $\text{havingCl}$ for $\text{havingClause}$, and $\text{havingIndAt}$ for $\text{havingIndexedAtom}$.

\[
\begin{align*}
\text{havingCl}(\vec{x}, \vec{y}, i) & \rightarrow \text{havingIndAt}(\vec{x}, \vec{y}, i) \\
\text{havingCl}(\vec{x}, \vec{y}) & \rightarrow \text{havingValueAtom}(\vec{x}, \vec{y}) \\
\text{havingCl}(\vec{x}) & \rightarrow \text{havingMacroAtom}(\vec{x}) \\
\text{havingCl}(i_1, i_2) & \rightarrow \text{havingIndexArithAtom}(i_1, i_2) \\
\text{havingCl}(\vec{x}, \vec{y}, i, \vec{i}) & \rightarrow \text{FORALL } i' \text{ IN seq havingCl}(\vec{x}, \vec{y}, i', \vec{i}) | \\
& \text{EXISTS } i' \text{ IN seq havingCl}(\vec{x}, \vec{y}, i', \vec{i}) | \\
& \text{FORALL } y' ( \text{ IF havingIndAt}(\vec{x}, y', \vec{y}, i) \text{ THEN havingCl}(\vec{x}, y', \vec{y}, \vec{i}) ) | \\
& \text{EXISTS } y' \text{ havingIndAt}(\vec{x}, y', \vec{y}, i) \text{ AND havingCl}(\vec{x}, y', \vec{y}, \vec{i}) | \\
& \text{NOT havingCl}(\vec{x}, \vec{y}, \vec{i}) \\
\text{havingCl}(\vec{x}, \vec{x'}, \vec{y}, \vec{y'}, i, \vec{i}, \vec{i'}) & \rightarrow \text{havingCl}(\vec{x}, \vec{y}, \vec{i}) \text{ OR havingCl}(\vec{x'}, \vec{y'}, \vec{i'}) | \\
& \text{havingCl}(\vec{x}, \vec{y}, \vec{i}) \text{ AND havingCl}(\vec{x'}, \vec{y'}, \vec{i'}) | \\
& \text{IF havingCl}(\vec{x}, \vec{y}, \vec{i}) \text{ THEN havingCl}(\vec{x'}, \vec{y'}, \vec{i'})
\end{align*}
\]

The clauses are constructed as FOL formulas where all quantifications over abstract objects and values are guarded. This ensures that with the quantifiers $\text{FORALL}$ and $\text{EXISTS}$ one quantifies only on the told values of the SPARQL queries. Moreover, as an additional safety condition we demand that for all value variables there is an indexed atom and an embedded filter query in it, in which the variable occurs as distinguished variable. Last but not least, we demand that the $\text{HAVING}$ clause does not contain open index variables.

The macro atoms are defined by a $\text{CREATE AGGREGATE OPERATOR}$ clause. The outer structure of expressions $\text{UnaryAggDef}$ and $\text{BinaryAggDef}$ is similar to the one of having clauses, the exception being that they may contain higher order variables holding the positions at which the star patterns are going to be substituted for the macro expansion. We realize the definition by introducing an additional atom expression $\text{HigherOrdIndAtom}$ which replaces the indexed atoms. To avoid circular macro definitions, we disallow nesting of macro atoms.

\[
\begin{align*}
\text{HigherOrdIndAtom}(y, i) & \rightarrow \text{funcVar}(y) <i> \\
\text{aggDef}(\vec{x}, \vec{y}) & \rightarrow \text{havingValueAtom}(\vec{x}, \vec{y}) \\
\text{aggDef}(x, i) & \rightarrow \text{HigherOrdIndAtom}(x, i) \\
\text{aggDef}(i_1, i_2) & \rightarrow \text{havingIndexArithAtom}(i_1, i_2) \\
\text{aggDef}(\vec{x}, \vec{y}, i, \vec{i}) & \rightarrow \text{FORALL } i' \text{ IN seq aggDef}(\vec{x}, \vec{y}, i', \vec{i}) | \\
& \text{EXISTS } i' \text{ IN seq aggDef}(\vec{x}, \vec{y}, i', \vec{i}) | \\
& \text{FORALL } y' ( \text{ IF havingIndAt}(\vec{x}, y', \vec{y}, i) \text{ THEN aggDef}(\vec{x}, y', \vec{y}, \vec{i}) ) | \\
& \text{EXISTS } y' \text{ havingIndAt}(\vec{x}, y', \vec{y}, i) \text{ AND aggDef}(\vec{x}, y', \vec{y}, \vec{i}) | \\
& \text{NOT aggDef}(\vec{x}, \vec{y}, \vec{i}) \\
\text{aggDef}(\vec{x}, \vec{x'}, \vec{y}, \vec{y'}, i, \vec{i}, \vec{i'}) & \rightarrow \text{aggDef}(\vec{x}, \vec{y}, \vec{i}) \text{ OR aggDef}(\vec{x'}, \vec{y'}, \vec{i'}) | \\
& \text{aggDef}(\vec{x}, \vec{y}, \vec{i}) \text{ AND aggDef}(\vec{x'}, \vec{y'}, \vec{i'}) | \\
& \text{IF aggDef}(\vec{x}, \vec{y}, \vec{i}) \text{ THEN aggDef}(\vec{x'}, \vec{y'}, \vec{i'})
\end{align*}
\]
The set of aggregate definitions, which contains occurrences of exactly one functional variable \( f_1 \) is the set of expressions \( \text{unaryAggDef} \). The set of aggregate definitions, which contains occurrences of exactly two functional variables \( \text{funcVar}_1, \text{funcVar}_2 \) is the set of expressions \( \text{binaryAggDef} \). We assume that the functional variables are ordered. Actually, as we do handle only macros up to arity two we only need two concrete variables \( f_1 \) and \( f_2 \) which we refer to as the first and second variable resp.

The expansion of a having clause containing macro definitions is an interwined substitution. In case of a definition \( \text{unaryAggDef} \) for a unary macro atom \( \text{unaryMacroName}(\text{seqName}, \text{starPattern}(\vec{x})) \) the substitution is as follows: All higher order atoms \( f_1(x)<i> \) occurring in \( \text{unaryAggDef} \) are substituted by the expression which results from the unary star pattern by substituting all \( * \) occurrences with the domain variable \( x \), and then attaching to this expression the timestamp \( <i> \). In case of binary aggregation definitions the procedure is similar, the difference being that we now use the second star pattern for the second variable \( f_2 \).

The select clauses are conjunctions of SPARQL triples and aggregation atoms, both with a timestamp.

\[
\begin{align*}
\text{selectClause}(\vec{x}, \vec{y}) & \rightarrow \text{sparqlTriple}(\vec{x}, \vec{y}) <\text{timeExp}_1> . \{ \text{selectClause}(\vec{x}, \vec{y}) <\text{timeExp}_2> \} | \\
\text{aggAtom}(\vec{x}) & \rightarrow \text{attributeSymb}(x, \text{arithAggrOp}(\text{seqName}, \text{unaryValStarPattern}(\vec{x}))) \\
\text{arithAggrOp} & \rightarrow \text{COUNT} | \text{AVG} | \text{SUM} | \text{MIN} | \text{MAX}
\end{align*}
\]

Here are a few remarks on the rules above. The expression \( \text{timeExp} \) is an arithmetical expression using the keyword \texttt{NOW}. In most of the relevant examples, \( \text{timeExp} \) stands for \texttt{NOW}. Aggregation atoms are built by an attribute symbol, a term \( x \) that is either a constant or a variable contained in \( \vec{x} \), and an arithmetical aggregation operator based on some unary star filter pattern. As predefined arithmetical aggregations we allow the ones also provided by SQL.

The list of stream expressions refers to a list of named streams or select expressions again.

\[
\begin{align*}
\text{listOfWindowedStreamExp} & \rightarrow (\text{StreamName} | \text{selectExp})\text{windowExp} \\
\text{windowExp} & \rightarrow \text{slackConst}<\text{timeExp}_1<\text{timeExp}_2>\rightarrow\text{slideConst}
\end{align*}
\]

Both time expression parameters in the window must contain the keyword \texttt{NOW}. Moreover, they should be such that the intervals are well defined. In most cases, \( \text{timeExp}_1 \) will have the form \( \text{NOW} - \text{timeConst} \). The time expressions in the window could lead to undefined intervals. Hence we demand that \( \text{timeExp}_1 \) is smaller than \( \text{timeExp}_2 \) for all positive value instantiations from the time domain for \texttt{NOW}.

The expression \( \text{listOfRessources} \) just describes the list of ABoxes and TBoxes which have to be used for answering the embedded queries.

\[
\begin{align*}
\text{listOfRessources} & \rightarrow \text{typedRessourceList}[ , \text{listOfRessources}] \\
\text{typedRessourceList} & \rightarrow \text{STATIC ABOX listofURIstoStaticABoxes} | \\
& \text{TEMPORAL ABOX listofURIstoTemporalABoxes} | \\
& \text{TBOX listofURIstoTBoxes}
\end{align*}
\]

There may be different methods to sequence ABoxes. We provide the standard method \texttt{StdSeq} as a built-in method. A second built-in method expects an equivalence relation \( \sim \) on the time domain. (In later versions of STARQAL we are going to allow for other relations \( \sim \), which may not fulfill transitivity.)

\[
\begin{align*}
\text{seqMethod} & \rightarrow \text{StdSeq} | \text{SeqMethod}(\sim)
\end{align*}
\]

Another feature of our language, which was not discussed on the example section, concerns rigidity. As discussed above we assume that the TBox is a global atemporal resource in which all axioms hold for all
time points. Hence, within such TBoxes one cannot declare persistency conditions saying, e.g., that if a sensor is broken, then it will stay broken. To mitigate this weak expressiveness of the TBoxes, we introduce persistency declarations into our query language, with which we can formulate that a concept or a role is (future-) persistent.

\[
\text{declareExp} \rightarrow \text{DECLARE PERSISTENT (roleSymbol | conceptSymbol)}
\]

5.2 Semantics

STARQL queries have streams of ABox assertions as input and output. That means that the definition of the semantics for STARQL should explicate how the output stream of ABox assertions is computed from the input streams. Using the compositionality principle, the semantics definition for SPARQL can be accomplished by defining the semantic denotations for the substructures of the query (be it subqueries or other data structures) and then by composing them to the denotation of the whole query.

Regarding the query components of STARQL, one has to explain the different keywords, the semantics of all involved operators, i.e., the sequencing operator, the window operator, the all and exists aggregators. Moreover, as STARQL relies on embedded DL/SPARQL query filters (in the \text{WHERE} and \text{HAVING} clauses), one has also to declare their semantics, which will be done here by short recaps of the standard certain answer semantics.

The syntax definition for STARQL contained expressions that are time related such as \text{startTime}, \text{frequency} in the create pulse expression or \text{slackConst}, \text{slideConst} in the window expression. All these will be evaluated to time points in a given time domain \((T, \leq)\), a linearly ordered set of timestamps. We do not necessarily restrict \(T\) to be the set of natural numbers, but also allow for the rational and the real numbers. In all our examples in Section 4 we used seconds as unit. Other time units would have to be considered as syntactic sugar and would have to be converted by the query answering system that implements STARQL.

STARQL queries can contain aggregation macros. We explained how these are expanded in Section 4 to queries containing no macros. Hence, for the definition of the semantics we may safely assume macro free STARQL queries.

5.2.1 Window semantics

We start here with defining the semantics of the window operator. We will deal with the slack parameter in later phases of the project (hence, set it in this report directly to 0), and so we define here only the window w.r.t. to the sliding aspect. The denotation \(\llbracket wS \rrbracket\) of a windowed stream expression \(wS\)

\[
wS = S \ \text{winExp} = S \ 0 \leftarrow [\text{timeExpNow}_1, \text{timeExpNow}_2] \rightarrow \text{sl}
\]

is a stream of temporal ABoxes. Let \(\llbracket S \rrbracket\) be the denotation of \(S\), which may be an input stream name or a complex stream expression for a stream of ABox assertions. Let further \(\lambda t.g_1(t) = \llbracket \text{tempExpNow}_1 \rrbracket\) and \(\lambda t.g_2(t) = \llbracket \text{tempExpNow}_2 \rrbracket\) be the functions corresponding to the time expressions.

The pulse declaration (that is: the local pulse declaration within the create stream expression, if given; or if not given, then the default pulse definition) defines a subset \(T' \subseteq T\) of the time domain. This set \(T'\) is going to be declared to be the domain of the stream of temporal ABoxes. Let \(T'\) be represented by the increasing sequence of timestamps \((t_i)_{i \in N}\), where \(t_0\) is the starting point fixed in the pulse declaration.

Now, one defines for every \(t_i\) the temporal ABox \(\mathcal{A}_{t_i}\) such that \((\mathcal{A}_{t_i}, t_i) \in \llbracket wS \rrbracket\). If \(t_i < \text{sl} - 1\), then \(\mathcal{A}_{t_i} = \emptyset\). Else set first

\[
tstart = \lfloor t_i/\text{sl} \rfloor \times \text{sl}
\]

\[
tend = \max\{tstart - (g_2(t_i) - g_1(t_i)), 0\}
\]
and define on that basis

$$\tilde{\mathcal{A}}_i = \{(ass, t) \mid (ass, t) \in [S] \text{ and } t_{end} \leq t \leq t_{start}\}$$

We describe the function, which gets as input a stream $[S]$ and outputs a stream of temporal ABoxes as $F_{\text{winExp}}^{\text{winExp}} : \text{StreamsOfAssertions} \rightarrow \text{StreamOfABoxes}$. Using this function, we can write the denotation of the windowed stream more compactly as $[wS] = F_{\text{winExp}}^{\text{winExp}}([S])$.

### 5.2.2 Merge for Multiple Streams

The SELECT expressions allow for a list of windowed streams; the semantics of the list is a simple temporal join defined as follows. Let $[\text{StA}_1] = \{(\tilde{\mathcal{A}}_1, t) \mid t \in T_1$, $[\text{StA}_2] = \{(\tilde{\mathcal{A}}_2, t) \mid t \in T_2\}$ be two streams of temporal ABoxes with supports $T_1, T_2$. Then $[\text{StA}_1] \bowtie \text{temp} [\text{StA}_2]$ is defined as follows:

$$[\text{StA}_1] \bowtie \text{temp} [\text{StA}_2] = \{(\tilde{\mathcal{A}}_1, t) \mid t \in T_1 \setminus T_2\} \cup \{(\tilde{\mathcal{A}}_2, t) \mid t \in T_2 \setminus T_1\} \cup \{(\tilde{\mathcal{A}}_1 \cup \tilde{\mathcal{A}}_2, t) \mid t \in T_1 \cap T_2\}$$

This stream of temporal ABoxes is the input for the sequencing method.

### 5.2.3 The Sequencing Method

Given a stream of temporal ABoxes $\text{StA}$, the standard sequencing method just gathers together all ABox assertions with the same timestamp into the ABox. That is, let $(\tilde{\mathcal{A}}_1, t) \in \text{StA}$ and let $t_1, \ldots, t_k$ be the timestamps occurring in $\tilde{\mathcal{A}}_i$. Then the sequence of ABoxes $\text{seqA}$ at time point $t$ is defined as $(\mathcal{A}_1, \ldots, \mathcal{A}_k)$ where for every $i \in \{1, \ldots, k\}$ the pure ABox $\mathcal{A}_i$ is defined by $\mathcal{A}_i = \{(ass, t_i) \mid (ass, t_i) \in \tilde{\mathcal{A}}_i\}$. This is the definition in case of an empty temporal ABox (specified in the USING clause).

If there is a temporal ABox $\mathcal{A}$ specified, then it is incorporated in the sequencing at the right time point. That means: For $\mathcal{A}_i = \{(ass, t_i) \in \tilde{\mathcal{A}}_i\}$ denoting again the $i$th ABox in the sequence, one now defines $\mathcal{A}_i = \{(ass, t_i) \mid (ass, t_i) \in \tilde{\mathcal{A}}_i\} \cup \{(ass, t_i) \mid (ass, t_i) \in \mathcal{A}\}$. We denote the function which gets a stream of temporal ABoxes and outputs a stream of sequences of ABoxes (according to the standard sequencing method) as $F_{\text{seqStdSeq}}$.

Besides the standard method we also allow for a second predefined family of sequencing methods. (In later extensions of STARQL we will allow even the user to define his own sequencing methods.) The members of the family are specified by an equivalence relation $\sim$ on the time domain. (In later extensions of STARQL we will also consider similarity relations which do not have to be transitive.) The equivalence classes $[x]_\sim$ for $x \in T$ form a partition of the domain. We restrict the class of admissible equivalence relations to those $\sim$ that respect the time ordering, i.e., the partition of $\sim$ equivalence classes should be convex intervals on the time domain.

Now, we define the sequence of ABoxes generated by $\text{seqMethod}(\sim)$ on the stream of temporal ABoxes $\text{StA}$ as follows: Let $(\tilde{\mathcal{A}}_t, t)$ be the temporal ABox at time point $t$. Let $T' = \{t_1, \ldots, t_k\}$ be the time points occurring in $\tilde{\mathcal{A}}_t$ and let $k$ be the number of different equivalence classes generated by the time points in $T'$. Then define the sequence at $t$ as the $(\mathcal{A}_1, \ldots, \mathcal{A}_k)$ and for every $i \in \{1, \ldots, k\}$ the pure ABox $\mathcal{A}_i$ is defined by

$$\mathcal{A}_i = \{(ass, t') \mid (ass, t') \in \tilde{\mathcal{A}}_t \text{ and } t' \text{ in } i\text{th equivalence class}\}$$

We denote the function which gets a stream of temporal ABoxes and outputs a stream of sequences of ABoxes according to this method as $F_{\text{seqMethod}(\sim)}$.

### 5.2.4 The Semantics of the Where and Having Clauses

STARQL relies on the certain answer semantics for the embedded queries. We recapitulate its definition and then describe the semantics of the where and having clauses.
If $\Psi(\vec{x})$ is any FOL formula (not necessarily a UCQ or a GCQ) with open variables in $\vec{x}$ and symbols in a signature $\text{Sign}$, and $\mathcal{T} \cup \mathcal{A}$ is an ontology on $\text{Sign}$, then the set of certain answers is defined as those tuples $\vec{a}$ constants in $\mathcal{T} \cup \mathcal{A}$, that can be shown to fulfill $\Psi$ w.r.t. $\mathcal{T} \cup \mathcal{A}$.

$$\text{cert}(\Psi(\vec{x}), \mathcal{T} \cup \mathcal{A}) = \{ \vec{a} \mid \mathcal{T} \cup \mathcal{A} \models \Psi[\vec{x}/\vec{a}] \}$$

The WHERE clauses allow for embedded queries—in our case UCQ resp. GCQ—and they are evaluated against the static ABox $\mathcal{A}_\text{st}^1, \ldots, \mathcal{A}_\text{st}^k$ and the TBoxes $\mathcal{T}_1, \ldots, \mathcal{T}_l$ listed in the USING clause. Hence the constant vectors $\vec{a} \in \text{cert}(\Phi(\vec{x}), \bigcup_{1 \leq i \leq l} \mathcal{T}_i \cup \bigcup_{1 \leq i \leq k} \mathcal{A}_\text{st}^i)$ will be passed to the having clauses.

The semantics of the having clauses rests on the meaning of the indexed atoms, whose meaning again are given by the certain answer semantics for the embedded filter queries. Let be given the sequence of ABoxes $\text{seq} = (\mathcal{A}_1, \ldots, \mathcal{A}_k)$ and let $I = \{1, \ldots, k\}$ be the index set of the sequence. The set of individual constants contained in the static ABoxes, the temporal ABoxes, and in the streams is denoted $\mathcal{CN}$, the set of value constants therein is denoted $\mathcal{VN}$. Let $\Phi(\vec{y})$ be a SPARQL query, either a UCQ or a GCQ, with open variables in $\vec{x}_i = (y_1, \ldots, y_n)$. The guarded fragment of FOL generated by the grammar for havingClause contains only predefined predicates, those for the arithmetics on the values and the indices on the one hand, and the indexed atoms $\Phi(\vec{y})(i)$, which can be thought of $n+1$-ary atoms with predefined meanings (namely predefined by the certain answer semantics.) So, in essence we have to declare the meaning of the indexed atoms. We first define its meaning without any persistency declaration axioms.

$$\text{seq} \models \Phi(\vec{b})(\vec{i}) \iff \vec{b} \in \text{cert}(\Phi(\vec{x}), \mathcal{A}_I \cup \bigcup_{1 \leq i \leq l} \mathcal{T}^i \cup \bigcup_{1 \leq i \leq k} \mathcal{A}^i_{\text{st}})$$

The semantics of more complex having clauses is defined recursively in the usual manner. To handle variables correctly, the modeling relation $\models$ is defined not only for a sequence $\text{seq}$ but also an variable assignment $\text{va}$ of constants to all variables. The domains of the quantifiers over individual variables are $\mathcal{CN}$, the quantifiers over indexes is the index set $I$ of the sequence, and the quantifiers over value variables are value constants $\mathcal{VN}$.

If there are persistency declarations, then the semantics are declared with respect to an extended sequence of ABoxes, in which the persistency assumptions are materialized. For example suppose that the concept symbol $A$ is declared to be persistent and that it occurs in an ABox assertion $A(a)$ in $\mathcal{A}_i$. Then the sequence is changed such that for all ABoxes $\mathcal{A}_j$ later in the sequence $(i < j)$ the atom $A(a)$ is added.

A HAVING clause $\Phi$ either evaluates to a truth value or to a set of bindings for the open variables $\vec{x}$ in it. So, if there is a stream of ABox sequences $\text{stseq}$, then we can apply the above evaluation of $\Phi$ for all sequences of ABoxes in $\text{stseq}$. The resulting output stream is defined as follows: If $\Phi$ is a boolean query evaluating to true in the ABox sequence $(\text{seq}_t, t) \in \text{stseq}$, then the stream contains $\{\vec{a}\}$, $t$. If $\Phi$ evaluates to false, then there is no stream element for time point $t$. If $\Phi$ contains a non-empty list of variables $\vec{x}$ and it is evaluated to $\vec{a}$ in the ABox sequence $(\text{seq}_t, t) \in \text{stseq}$, then $(\vec{a}, t)$ is in the output stream. If there is no binding, then there is no stream element at time point $t$. We denote the function, which maps the stream of sequences under the having clause $\Phi$ to a stream of bindings as $F^{\Phi(\vec{x})}_{\text{stseq}2\text{ob}}$.

### 5.2.5 Arithmetic Aggregation Operators

The arithmetic operators are defined on finite sets $\mathcal{SV}$ of some value domain. We assume that the value domain has a linear ordering and an addition operator (so the natural, natural, or real numbers are admissible domains.) Then, $[[\text{MAX} (\mathcal{SV})]]$ denotes the maximum in $\mathcal{SV}$, $[[\text{MIN} (\mathcal{SV})]]$ the minimum, $[[\text{COUNT} (\mathcal{SV})]]$ the cardinality of $\mathcal{SV}$, $[[\text{SUM} (\mathcal{SV})]]$ the sum and $[[\text{AVG} (\mathcal{SV})]]$ the average on $\mathcal{SV}$. 

32
5.2.6 Semantics of Select Expressions

Assume that we have the following select expression \( se \).

\[
se = \text{SELECT } \Theta_1(\vec{x}, \vec{y})(\text{timeExp}_1), \ldots, \Theta_r(\vec{x}, \vec{y})(\text{timeExp}_r)
\]

\[
\text{FROM } S_1 \text{ winExp}_1, \ldots, S_m \text{ winExp}_m
\]

\[
\text{USING } \text{STATIC ABOX } A_1^{\text{st}}, \ldots, A_k^{\text{st}}, \text{ TEMPORAL ABOX } \tilde{A}_1, \ldots, \tilde{A}^h,
\]

\[
\text{TBOX } T^1, \ldots, T^l
\]

\[
\text{WHERE } \Psi(\vec{x})
\]

\[
\text{SEQUENCE BY } seq\text{Meth}
\]

\[
\text{HAVING } \Phi(\vec{x}, \vec{y})
\]

The denotation \([se]\) is a stream defined on the basis of the denotations \([S_1]\), \ldots, \([S_m]\) of the input streams and all the functions \( F \) defined in the previous subsections. We introduce the following short cut \( svb_{se} \) for the expression denoting the stream of variable bindings produced by the select expression \( se \). This expression is used twice, once in the having clause, and once in the select head clause if one of the \( \Theta_i \) is an aggregation atom.

\[
[svb_{se}] = F_{seq\text{Meth}}(F_{\text{winExp}}(\[S_1]\)) \times_{\text{temp}} F_{\text{winExp}}(\[S_2]\)) \cdots \times_{\text{temp}} F_{\text{winExp}}(\[S_m]\))
\]

With this the denotation of \( se \) is defined as follows:

\[
[se] = \{[\Theta_1(\vec{a}, \vec{b})(\text{timeExp}_1(t))], \ldots, \Theta_r(\vec{a}, \vec{b})(\text{timeExp}_r(t))] \mid
\]

\[
\tilde{a} \in \text{cert}(\Psi(\vec{x}), \bigcup_{1 \leq i \leq l} T^i \cup \bigcup_{1 \leq i \leq k} A_i^{\text{st}}) \text{ and }
\]

\[
(\vec{b}, t) \in F_{\Phi(\vec{a}, \vec{y})}(svb_{se})
\]

To complete the the semantics for select expressions we have to define \([\Theta_i(\vec{a}, \vec{b})(\text{timeExp}_i(t))]\), the denotations of the the select (head) clause expression. \([\text{timeExp}_i(t)]\) evaluates to a timestamp constant \( tsc_i \) by applying the function \([\text{timeExp}]\) (in the variable \( t \)) to \( t \). If \( \Theta_i(\vec{a}, \vec{b}) \) is a SPARQL triple \( \text{sparqlTriple}(a, b) \), then

\[
[\Theta_i(\vec{a}, \vec{b})(\text{timeExp}_i(t))] = \text{sparqlTriple}(\vec{a}, \vec{b})(tsc_i)
\]

If \( \Theta_i(\vec{a}, \vec{b}) \) is an aggregation atom expression \( \text{aggAtom} \) of the form

\[
\text{attributeSymb}(a, \text{arithAggrOp}(\text{unaryValStarPattern}(\vec{a})))
\]

then the denotation depends on the used aggregation operator. Let the unary value star pattern be denoted by \( \chi(\vec{a}, \ast) \) and consider \( \chi(\vec{a}, z) \) for a new variable \( z \) not appearing in the expression \( se \). The set of value constants

\[
SV(t) = \{vc \mid (vc, t) \in F_{\chi(\vec{a}, z)}(svb_{se})\}
\]

denotes a set value constants resulting as variable bindings for \( z \) at time point \( t \). Then we define

\[
[\Theta_i(\vec{a}, \vec{b})(\text{timeExp}_i(t))] = \text{attributeName}(a, [\text{arithAggrOp}(SV(t))])(tsc_i)
\]

which completes the definition of \( se \).
Chapter 6

Summary

We have proposed an orthogonal query language which can refer to streams in order to produce new streams etc. In particular, we have investigated stream querying in an OBDA/ABDEO setting.

In the context of a multiple query setting one may also add as a distinguishing feature a local scope principle. Normally, in temporal database systems there is just one big central database that is queried. In contrast, the multiple stream concept lends itself to the idea that ABox assertions hold within the streams which are queried, and more interestingly, in the extended framework of queries, the output assertions hold only on the output stream associated with the query. The benefits of this locality is its flexibility. For instance, it may be the case that different stream sources talk about the same things (e.g., about the temperature within nearby areas in a turbine), but we are only interested on the ones from some specific sensor, as it is either more reliable or it is known to be faulty and we want to test it.

The main idea of the query language STARQL is to capture the data within a sliding window operator, that at every moment in the evolving time makes a snapshot of the data within a period specified in the window size, filters/calculates the queried information in the window, and then slides to the next evaluation time point specified by the slide parameter. In practice, it may be the case that, after a sliding step, data arrive that are not within the actual window. So, in addition to the window size one can provide a slack size that allows to temporarily widen the window.

One of the reasons of the success in applying the sliding window concept to DSMS can be found in the closed world assumption inherited from relational DBMS: every fact recorded in the database is considered to hold/to be true, whereas everything not recorded in the databases is assumed not to hold. In formal logic speak, a database is a predicate logical interpretation/structure that for every relation (schema) determines its extension. Clearly, this picture and its underlying closed world assumption is an idealization which does not do justice to the practical DB applications. And indeed, the relation DB paradigm has also found a way to deal with indefinite knowledge by NULL values: Sometimes one just does not know the entry of a column in the relational scheme, hence rather than declaring that all facts resulting by instantiating the column with one of the allowed values to be false, one relaxes the closed world assumption and just mentions that one does not know the value by marking the entry with the NULL value. Moreover, relational DBMS come also with the possibility to formulate constraints (primary key, foreign key etc.) on the data. All these additional expressive means are well investigated and part of the folklore in the DB community (see the standard foundational book on databases [1]).

For handling indefiniteness, incompleteness in streams of data one can rely on the general solutions of conventional relational DBMS. And many of the mentioned approaches follow this strategy by inheriting most of the conventional relations DBMS operators into the DSMS query language. We describe an alternative approach for dealing with temporal data and streams. The approach relies mainly on the ideas of ontology based data access (OBDA). In contrast to the approaches extending SQL, the OBDA approach is of a more logic oriented style, and breaks radically with the closed world assumption in order to deal with the indefiniteness and incompleteness in data; moreover, it provides a more declarative way to formulate and apply reasoning on complex constraints represented via axioms in the TBox. The other part of the ontology
is made up by the ABox, containing assertional axioms which are produced by mappings from the real data stored in a database.

As we said above, the approach described is centered around OBDA, but it addresses additional aspects that are not covered by the classical literature on OBDA. OBDA in the classical (narrow) sense uses logics in the DL-Lite family that allow for the generation of objects (true existentials on the right hand sides of general inclusion axioms) but still allow for a reduction of reasoning w.r.t. the TBox to answering of rewritten first order logic queries on the original data. For some applications the expressiveness of the classical OBDA paradigm is not sufficient, and hence there is a need for the challenging paradigm of accessing big data w.r.t. to expressive ontologies (ABDEO). The idea of this paradigm is to make the access on big data feasible by first partitioning the data into small chunks (modules).

Having these distinctions in mind, we can position our approach as a semantical framework in the intersection of classical OBDA, ABDEO and stream processing [96]. We propose a new query language and semantics with a necessary extension on the window concepts, fitting better into the OBDA/ABDEO paradigms. A central idea is to provide ABox sequence constructors that group timestamped ABox assertions into ABoxes. The sequence sets up a (nearly) standard context in which OBDA/ABDEO reasoning services can be applied.
Bibliography


36


[34] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Vijayshankar Raman, Frederick Reiss, and Mehul A. Shah. Telegraphcq: Continuous dataflow processing for an uncertain world. In CIDR, 2003.


Appendix A

Outline of Time and Streams Architecture

This appendix reports the paper:

Addressing Streaming and Historical Data in OBDA Systems: Optique’s Approach
(Statement of Interest)

Ian Horrocks\textsuperscript{1}, Thomas Hubauer\textsuperscript{4}, Ernesto Jimenez-Ruiz\textsuperscript{2}, Evgeny Kharlamov\textsuperscript{2},
Manolis Koubarakis\textsuperscript{4}, Ralf Möller\textsuperscript{1}, Konstantina Bereta\textsuperscript{4}, Christian Neuenstadt\textsuperscript{1},
Özgür Özcep\textsuperscript{1}, Mikhail Roshchin\textsuperscript{3}, Panayiotis Smeros\textsuperscript{4}, Dmitriy Zheleznyakov\textsuperscript{2}

\textsuperscript{1} Hamburg University of Technology, Germany
\textsuperscript{2} Oxford University, UK
\textsuperscript{3} Siemens Corporate Technology, Germany
\textsuperscript{4} University of Athens, Greece

\textbf{Abstract.} In large companies such as Siemens and Statoil monitoring tasks are of great importance, e.g., Siemens does monitoring of turbines and Statoil of oil behaviour in wells. This tasks bring up importance of both streaming and historical (temporal) data in the Big Data challenge for industries. We present the Optique project that addresses this problem by developing an Ontology Based Data Access (OBDA) system that incorporates novel tools and methodologies for processing and analyses of temporal and streaming data. In particular, we advocate for modelling time aware data by temporal RDF and reduce monitoring tasks to knowledge discovery and data mining.

\section{Introduction}

A typical problem that end-users face when dealing with Big Data is of data access, which arises due to the three dimensions (the so-called “3V”) of Big Data: \textit{volume}, since massive amounts of data have been accumulated over the decades, \textit{velocity}, since the amounts may be rapidly increasing, and \textit{variety}, since the data are spread over different formats. In this context accessing the \textit{relevant} information is an increasingly difficult task and the Optique project\textsuperscript{8} aims at providing solutions for it.

The project is focused around two demanding use cases that provide it with motivation, guidance, and realistic evaluation settings. The first use case is provided by Siemens\textsuperscript{5} and encompasses several terabytes of temporal data coming from sensors, with a growth rate of about 30 gigabytes per day. Users need to query this data in combination with many gigabytes of other relational data that describe events. The second use case is provided by Statoil\textsuperscript{6} and concerns more than one petabyte of geological data. The data are stored in multiple databases which have different schemata, and the users have to manually combine information from many databases, including temporal, in order to get the results for a single query. In general, in the oil and gas industry, IT-experts spend 30–70\% of their time gathering and assessing the quality of data\textsuperscript{7}.

\begin{footnotesize}
\begin{itemize}
\item \textsuperscript{5} http://www.siemens.com
\item \textsuperscript{6} http://www.statoil.com
\end{itemize}
\end{footnotesize}
is clearly very expensive in terms of both time and money. The Optique project aims at solutions that reduce the cost of data access dramatically. More precisely, Optique aims at automating the process of going from an information requirement to the retrieval of the relevant data, and to reduce the time needed for this process from days to hours, or even to minutes. A bigger goal of the project is to provide a platform with a generic architecture that can be adapted to any domain that requires scalable data access and efficient query execution.

The main bottleneck in the Optique’s use cases is that data access is limited to a restricted set of predefined queries (cf. Figure 1, left, top). Thus, if an end-user needs data that current applications cannot provide, the help of an IT-expert is required to translate the information need of end-users to specialised queries and optimise them for efficient execution (cf. Figure 1, left, bottom). This process can take several days, and given the fact that in data-intensive industries engineers spend up to 80% of their time on data access problems [7] this incurs considerable cost.

The Semantic approach known as “Ontology-Based Data Access” (OBDA) [16, 6] has the potential to address the data access problem by automating the translation process from the information needs of users (cf. Figure 1, right) to data queries. The key idea is to use an ontology, that presents to users a semantically rich conceptual model of the problem domain. The user formulates their information requirements (that is, queries) in terms of the ontology, and then receives the answers in the same intelligible form. These requests should be executed over the data automatically, without an IT-expert’s intervention. To this end, a set of mappings is maintained which describes the relationship between the terms in the ontology and the corresponding data source fields.

As discussed above, in the Siemens use case one has to deal with large amounts of streaming data, e.g., sensor and event data from many turbines and diagnostic centres in many different streams with the size of up to two kilobytes, in combination with historical, that is, temporal relational data sources. Thus, one has to provide Semantic technologies that enable modelling, e.g., with temporal RDF, and processing of both historical and streaming data which includes data mining and complex event processing. The data mining (and time series analysis) aspect is crucial for the Siemens use case as it sets the foundation for preventive diagnostics. More concretely, one of the diagnostic
Fig. 2. Left: classical OBDA approach. Right: the Optique OBDA system

engineers’ requirements is a means to find correlations between events like that of a can flame failure in a turbine and patterns in turbine’s relevant timed stamped sensor data measuring temperature (or pressure etc.) If some correlation between an event and sensor data is detected in the historical data by some data mining procedure, then this correlation should be expressible by a continuous query that can be used on real-time data for preventive diagnostics. For example an error event in a turbine $T$ may be identified within the sensor data by at least five percent decrease of measured value TC255-Measurement in sensor TC255 w.r.t. the average in one hour followed by a statistically significant increase (here: more than the two times of the measured average in hour) of measured value TC256-Measurements. The continuous query may refer to a measurement ontology and a (rough) model of the turbine structure within the diagnostic engineer’s ontology, that he has to set up in order to localize failures (up to some precision).

The classical OBDA systems (cf. Figure 2, left, shows a conceptual architecture of a classical OBDA system) fail to provide support for these tasks. In the Optique project, we aim at developing a next generation OBDA system (cf. Figure 2, right) that overcomes this limitations. More precisely, the project aims at a cost-effective approach that includes the development of tools and methodologies for processing and analytics of streaming and temporal data. These require to revise existing and develop new OBDA components, in particular, to support novel: (i) ontology and mapping management, (ii) user-friendly query formulation interface(s), (iii) automated query translation, and (iv) distributed query optimisation and execution in the Cloud. In this paper we will give a short overview of challenges that we encompass on the way to this goal.

The remainder of the paper is organised as follows. We discuss Optique’s challenges in handling of streaming and historical data and present the general architecture of the Optique’s OBDA solution in Section 2. Finally, we discuss related work (Section 3) and conclude (Section 4).
2 Stream Processing and Analytics in Optique’s OBDA

A general requirement, especially motivated by the Siemens use case, is to support such a combination of the data, ontology, mapping, and query languages that is expressive enough for modelling machines, symptoms, and diagnoses, and guarantees a complete, correct, and feasible query answering over temporal and streaming data. We now overview challenges to be solved in achieving this goal: we discuss ontologies, query languages, query processing, and visualisation of answers.

We plan to model temporal data with some extension of RDF. This could be achieved, for example, by adding to RDF triples an extra fourth component: validity time. Thus, the first challenge to address is an understanding of the right temporal RDF data model.

The key component in the Optique OBDA solution is the domain ontology, since it enables users to understand the data at hand and formulate queries. Thus, the next challenge to address is how to model both data streams and temporal data via ontologies. In particular this will require to model time (at least in the query language). On the level of mappings, a homogeneous mapping language for static and streaming data has to be provided.

The query language that the system should provide to end users should combine

(i) *temporal operators*, that address the time dimension of data and allow to retrieve data which was true “always” in the past or “sometimes” in the last X months, etc.,

(ii) *time series analysis operators*, such as mean, variance, confidence intervals, standard deviation, as well as trends, regression, correlation, etc., and

(iii) *stream oriented operators*, such as sliding windows.

Besides, the query language should provide some means for intelligent query answering of queries on complex patterns in the data by, e.g., telling in the negative case what similar patterns exist (query relaxation) and in the positive case how the multitude of patterns can be further restricted (query refinement). Finally, the query language should support formulating queries based on results of explorative data analysis.

Given the query, mapping languages, and ontology, the Optique system should be able to translate queries into highly optimised executable code over the underlying temporal and streaming data. This requires techniques for automated query translation of one-time, continuous, temporal queries, and their combinations. Existing translation techniques are limited and they do not address query optimisation and distributed query processing. Thus, novel approaches should be developed.

Another set of challenges in Optique is related to handling answers to queries. One issue is visualisation of massive volumes of data formatted according to the domain ontology. To address this issue, in particular, Data Mining and Pattern Learning, techniques over ontological data should be developed to enable automatic identification of interesting patterns in the data. Another challenge is how to manage “dirty” data. The Optique system must provide basic means for data cleaning such as: automated identification, mapping and alignment of data types, including dates, time synchronisation, and data quality issues (outlier detection, noise, missing values, etc.).

To sum up this section, we will provide the general architecture of the Optique OBDA system.
The general architecture of the Optique OBDA system

**General architecture of Optique’s OBDA solution.** Figure 3 gives an overview of the Optique’s OBDA solution architecture and its components. The architecture is developed using the three-tier approach and has three layers:

- The **presentation layer** consists of three main user interfaces: to compose queries, to visualise answers to queries, and to maintain the system by managing ontologies and mappings. The first two interfaces are for both end-users and IT-experts, while the third one is meant for IT-experts only.
- The **application layer** consists of several (main) components of the Optique’s system, supports its machinery, and provides the following functionality:
  - query formulation,
  - ontology and mapping management,
  - query answering, and
  - processing and analytics of streaming and temporal data.
- The **data and resource layer** consists of the data sources that the system provides access to, that is, relational, semistructured, temporal databases and data streams. It also includes a cloud that provides a virtual resource pool.
The entire Optique system will be integrated via the Information Workbench platform [11]7.

3 Related Work

Each of the approaches mentioned in the following deals with only one of the aspects that are relevant for the envisioned software component of the Optique query answering system: either the aspect of query answering over temporal data that can be described as historical; or the aspect of query answering over streamed data. Adhering to the requirements of the (Siemens) use case, the Optique approach favors a more integrative approach, that combines query answering over historical data and query answering over regularly updated temporal data stemming from many different streams.

There exist several approaches that address the problem of representing, inferring with, and querying temporal data within the general context of ontologies. As the Optique project will follow a weak temporalization of the OBDA paradigm, which will guarantee the conservation of so-called FOL rewritability (which essentially means a possibility to translate ontological queries into SQL queries over data sources), work on modal-style temporal ontology languages formalised via Description Logics [13] is of minor relevance; because of the bad complexity properties, this is even true for temporalized lightweight logics [2].

The approach in [10] introduces temporal RDF graphs, details out a sound and complete inference system, and gives a sketchy introduction to a possible temporal query language. A similar representation of temporal RDF graphs is adopted within the spatio-temporal RDF engine STRABON [12, 4]8, which also defines the spatio-temporal extension stSPARQL of the W3C recommendation query language SPARQL 1.1. Strabon is currently the only fully implemented spatio-temporal RDF store with rich functionality and very good performance as seen by the comparison in [12, 4]. For a similar temporal version of RDF, which is oriented at the temporal database language TSQL2, compare [9]. The authors of [17] favor a more conservative strategy by modeling time directly with language constructs within RDF and SPARQL—the resulting extensions of RDF and SPARQL being mere syntactic sugar. The logical approach of [14] follows ideas of [10] but shifts the discussion to the level of genuine ontology languages such as OWL; the semantics of the temporalized RDF and OWL languages are given by a translation to (a fragment of) first order logic. The temporalized SPARQL query language uses a careful separation of the time component and the thematic component that guarantees feasibility of query answering.

The concept of streaming relational data as well as the concepts underlying complex event processing are well understood and are the theoretical underpinnings for highly developed streaming engines used in industrial applications. The picture for stream processing within the OBDA paradigm is quite different; the few implemented streaming engines [5, 3, 15] are still under development and have been shown to lack one or other basic functionality [18]. Though all of the systems are intended to be used within the

7 www.fluidops.com/information-workbench/
8 www.strabon.di.uoa.gr
OBDA paradigm, only C-SPARQL [3] seems to have (minimal) capabilities for reasoning/query answering over ontologies. There is no agreement yet on how to extend SPARQL to work over streams; and so all of the mentioned systems have their own streamified version of SPARQL. However, the core of all extensions seems to be the addition of (sliding) window operators over streams, which are adapted from query languages over relational streams [1].

4 Conclusions

We presented motivations and challenges for the use of Ontology Based Data Access systems as a solution for Big Data access problem in industries. The important challenge in industries is knowledge discovery and data mining of temporal and streaming data, while the state of the art Semantic technologies that the OBDA systems rely on fail to address it adequately. The Optique project aims at providing this demanding technology which will be validated in the two industrial use case: Statoil and Siemens. In particular we plan to (i) explore the possibility of modelling streaming data with temporal RDF and (ii) understand how knowledge discovery and data mining techniques developed for Linked Open Data could be adapted and extended in our setting.

Acknowledgements.

The research presented in this paper was financed by the Seventh Framework Program (FP7) of the European Commission under Grant Agreement 318338, the Optique project.

References

Appendix B

Modularization Approach

This appendix reports the paper:

Advances in Accessing Big Data with Expressive Ontologies

Ralf Möller¹, Christian Neuenstadt¹, Özgür L. Özcep¹, and Sebastian Wandelt²

¹ Hamburg University of Technology, 21073 Hamburg, Germany
² Humboldt-Universität zu Berlin, 10099 Berlin, Germany

Abstract. Ontology-based query answering has to be supported w.r.t. secondary memory and very expressive ontologies to meet practical requirements in some applications. Recently, advances for the expressive DL $\mathcal{SHI}$ have been made in the dissertation of S. Wandelt for concept-based instance retrieval on Big Data descriptions stored in secondary memory. In this paper we extend this approach by investigating optimization algorithms for answering grounded conjunctive queries.

1 Introduction

Triplestores, originally designed to store Big Data in RDF format on secondary memory with SPARQL as a query language, are currently more and more used in settings where query answering (QA) w.r.t. ontologies is beneficial. However, reasoning w.r.t. ontologies in secondary memory is provided for weakly expressive languages only (e.g., RDFS), if even, and in some cases, query answering algorithms are known to be incomplete. For moderately expressive DL languages, such as DL-Lite, good results for sound and complete query answering w.r.t. large (virtual) Aboxes have already been achieved with OBDA based query rewriting techniques and schema specific mapping rules [1]. However, for expressive, more powerful DLs such as $\mathcal{ALC}$ and beyond only first steps have been made. Solving the problem of Accessing Big Data with Expressive Ontologies (ABDEO) is an important research goal.

A strategy to solve the ABDEO problem is to “summarize” Big Aboxes by melting individuals such that Aboxes fit into main memory [2]. In some situations inconsistencies occur, and summarization individuals must be “refined” (or unfolded) at query answering time in order to guarantee soundness and completeness, a rather expensive operation [3]. Other approaches make use of Abox modularization techniques and try to extract independent modules such that query answering is sound and complete. A first investigation of Abox modularization for answering instance queries w.r.t. the DL $\mathcal{SHIF}$ is presented in [5].

³ This work has been partially supported by the European Commission as part of the FP7 project Optique (http://www.optique-project.eu/).
However, modularization with iterative instance checks over all individuals and modules of an Abox is not sufficient to ensure fast performance [5].

The ABDEO approach presented here is based on a modularization approach developed by Wandelt [15,12,11,14] for really large Aboxes (containing data descriptions for > 1000 universities in terms of LUBM scale measures [5], i.e., datasets in the range of billions of triples). Modules (islands) derived by Wandelt’s techniques are usually small in practical applications and can be loaded into main memory such that a standard tableau prover can be used for instance checks. Iteration over all individuals gives sound and complete answers, in principle. Compared to [5], Wandelt (i) proposed extended modularization rules, (ii) implemented incremental ways of computing Abox modularizations, and (iii) investigated new ways to optimize sound and complete concept-based query answering (instance queries) with tableau-based reasoning systems for the logic $SHI$. In particular, “similarities” between modules are detected such that a single instance query on a representative data structure (a so-called one-step node) yields multiple results at a time. Due to modularization rules, one-step node query answering is sound [11] and in many (well-defined) cases complete. Therefore, since overall completeness is obtained by iterative instance checking for the remaining candidates, reducing the number of instance checks with one-step nodes drastically reduces query answering times (the approach is reminiscent of but different from [3], see below or cf. [11] for details). In addition, to eliminate candidates, Wandelt and colleagues also investigate complete approximation techniques (see [15] for details).

In this paper we extend Wandelt’s modularization based approach for query answering by investigating optimization techniques for answering grounded conjunctive queries w.r.t. $SHI$ ontologies. Grounded conjunctive queries are more expressive from a user’s point of view than instance queries. We argue that grounded conjunctive queries substantially narrow down the set of instance checking candidates if selective role atoms mentioned in queries are exploited for generating candidates for concept atoms, such that approximation techniques (to, e.g., DL-Lite) are not required in many cases. We demonstrate our findings using the LUBM benchmark as done, e.g., in [8] and [5]. As an additional extension to Wandelt’s work, which uses specific storage layouts for storing Abox data descriptions and internal information in SQL databases, we investigate ontology-based access to existing data stores, namely triplestores, while providing query answering w.r.t. expressive ontologies.

2 Preliminaries

We assume the reader is familiar with description logic languages, ontologies (knowledge bases), inference problems, and optimized tableau-based reasoning algorithms (see, e.g., [10,6]). For the reader’s convenience we define conjunctive queries in general, and grounded conjunctive queries in particular (adapted from

\footnote{Note that Abox modularization is different from Tbox modularization, as for instance investigated in [4].}
[9]). In the following we use \textbf{AtCon}, \textbf{Con}, and \textbf{Rol} for the sets of atomic concept descriptions, concept descriptions, and role descriptions, respectively, in the ontology.

A conjunctive query (CQ) is a first-order query \( q \) of the form \( \exists u.\psi(u, v) \) where \( \psi \) is a conjunction of concept atoms \( A(t) \) and role atoms \( R(t, t') \), with \( A \) and \( R \) being concept and role names, respectively. The parameters \( t, t' \) are variables from \( u \) or \( v \) or constants (individual names). The variables in \( u \) are the existentially quantified variables of \( q \) and \( v \) are the free variables, also called distinguished variables or answer variables of \( q \). The query \( q \) is called a \( k \)-ary query iff \(|v| = k \). In a grounded conjunctive query (GCQ), \( u \) is empty. We only consider grounded conjunctive queries in this paper. We define an operator \textit{skel} that can be applied to a CQ to compute a new CQ in which all concept atoms are dropped.

The query answering problem is defined w.r.t. an ontology \( O = (T, R, A) \). Let \( \text{Inds}(A) \) denote the individuals in \( A \). For \( I \) an interpretation, \( q = \psi(v) \) a \( k \)-ary grounded conjunctive query, and \( a_1, \ldots, a_k \in \text{Inds}(A) \), we write \( I \models q[a_1, \ldots, a_k] \) if \( I \) satisfies \( q \) (i.e., all atoms of \( q \)) with variables \( v_i \) replaced by \( a_i, 1 \leq i \leq k \). A certain answer for a \( k \)-ary conjunctive query \( q \) and an ontology \( O \) is a tuple \((a_1, \ldots, a_k)\) such that \( I \models q[a_1, \ldots, a_k] \) for each model \( I \) of \( O \). We use \( \text{cert}(q, O) \) to denote the set of all certain answers for \( q \) and \( O \). This defines the query answering problem. Given a \( SHI \) ontology \( O \) and a GCQ \( q \), compute \( \text{cert}(q, O) \). It should be noted that “tree-shaped” conjunctive queries can be transformed into grounded conjunctive queries, possibly with additional axioms in the Tbox [7]. The restriction to grounded conjunctive queries is not too severe in many practical applications.

Grounded conjunctive query answering can be implemented in a naive way by computing the certain answers for each atom and doing a join afterwards. Certain answers for concept atoms can be computed by iterating over \( \text{Inds}(A) \) with separate instance checks for each individual. Rather than performing an instance check on the whole Abox, which is too large to fit in main memory in many application scenarios, the goal is to do an instance check on a module such that results are sound and complete. More formally, given an input individual \( a \), the proposal is to compute a set of Abox assertions \( A_{isl} \) (a subset of the source Abox \( A \)), such that for all atomic (!) concept descriptions \( A \), we have \( (T, R, A) \models A(a) \) iff \( (T, R, A_{isl}) \models A(a) \).

3 Speeding Up Instance Retrieval

In order to define subsets of an Abox relevant for reasoning over an individual \( a \), we define an operation which splits up role assertions in such a way that we can apply graph component-based modularization techniques over the outcome of the split.

\textbf{Definition 1 (Abox Split).} Given

\begin{itemize}
  \item a role description \( R \),
\end{itemize}
– two distinct named individuals a and b,
– two distinct fresh individuals c and d, and,
– an Abox A,

an Abox split is a function \( \downarrow^{R(a,b)}_{c,d} : \text{SA} \to \text{SA} \), defined as follows (\( \text{SA} \) is the set of of Aboxes and \( A \in \text{SA} \)):

– If \( R(a,b) \in A \) and \( \{c,d\} \not\subseteq \text{Ind}(A) \), then

\[
\downarrow^{R(a,b)}_{c,d}(A) = A \setminus \{R(a,b)\} \cup \{R(a,d), R(c,b)\} \cup \{C(c) \mid C(a) \in A\} \cup \{C(d) \mid C(b) \in A\}
\]

– Else

\[
\downarrow^{R(a,b)}_{c,d}(A) = A.
\]

**Definition 2 (Extended ∀-info Structure).** Given a TBox \( T \) in normal form and an Rbox \( R \), an extended ∀-info structure for \( T \) and \( R \) is a function \( \text{extinfo}^T_R : \text{Rol} \to \wp(\text{Con}) \), such that we have \( C \in \text{extinfo}^T_R(R) \) if and only if there exists a role \( R_2 \in \text{Rol} \), such that \( R \models R \sqsubseteq R_2 \) and \( \forall R_2.C \in \text{clos}(T) \), where \( \text{clos}(T) \) denotes the set of all concept descriptions mentioned in \( T \).

For a formal definition of the normal form of a Tbox, see [11]. Here we use an example to illustrate the idea.

**Example 1 (Example for an Extended ∀-info Structure).** Let

\[
\begin{align*}
T_{\text{Ex1}} &= \{\text{Chair} \sqsubseteq \forall \text{headOf}.\text{Department}, \\
&\quad \exists \text{memberOf}.\top \sqsubseteq \text{Person}, \\
&\quad \text{GraduateStudent} \sqsubseteq \text{Student}\}, \\
R_{\text{Ex1}} &= \{\text{headOf} \sqsubseteq \text{memberOf}\},
\end{align*}
\]

then the TBox in normal form is

\[
\begin{align*}
T_{\text{Ex1norm}} &= \{\top \sqsubseteq \neg \text{Chair} \sqcup \forall \text{headOf}.\text{Department}, \\
&\quad \top \sqsubseteq \forall \text{memberOf}.\bot \sqcup \text{Person}, \\
&\quad \top \sqsubseteq \neg \text{GraduateStudent} \sqcup \text{Student}\}
\end{align*}
\]

and the extended ∀-info structure for \( T_{\text{Ex1norm}} \) and \( R_{\text{Ex1}} \) is:

\[
\begin{align*}
\text{extinfo}^T_R(R) &= \begin{cases} \\
\{\text{Department}, \bot\} & \text{if } R = \text{headOf}, \\
\{\bot\} & \text{if } R = \text{memberOf}, \\
\emptyset & \text{otherwise}.
\end{cases}
\end{align*}
\]
The extended ∀-info structure allows us to check which concept descriptions are (worst-case) propagated over role assertions in SHI-ontologies.

**Definition 3 (SHI-splittability of Role Assertions).** Given a SHI-ontology \( \mathcal{O} = (\mathcal{T}, \mathcal{R}, \mathcal{A}) \) and a role assertion \( R(a_1, a_2) \), we say that \( R(a_1, a_2) \) is SHI-splittable with respect to \( \mathcal{O} \) if:

1. there exists no transitive role \( R_2 \) with respect to \( \mathcal{R} \), such that \( R \sqsubseteq R_2 \),
2. for each \( C \in \text{extinfo}^\forall_{T,R}(R) \)
   - \( C = \bot \) or
   - there is a concept description \( C_2 \), s.th. \( C_2(b) \in \mathcal{A} \) and \( T \models C_2 \sqsubseteq C \) or
   - there is a concept description \( C_2 \), s.th. \( C_2(b) \in \mathcal{A} \) and \( T \models C \cap C_2 \sqsubseteq \bot \) and
3. for each \( C \in \text{extinfo}^\forall_{T,R}(R^\bot) \)
   - \( C = \bot \) or
   - there is a concept description \( C_2 \), s.th. \( C_2(b) \in \mathcal{A} \) and \( T \models C_2 \sqsubseteq C \) or
   - there is a concept description \( C_2 \), s.th. \( C_2(a) \in \mathcal{A} \) and \( T \models C \cap C_2 \sqsubseteq \bot \).

So far, we have introduced approaches to modularization of the assertional part of an ontology. In the following, we use these modularization techniques to define structures for efficient reasoning over ontologies.

We formally define a subset of assertions, called an individual island, which is worst-case necessary, i.e. possibly contains more assertions than really necessary for sound and complete instance checking. Informally speaking, we take the graph view of an Abox and, starting from a given individual, follow all role assertions in the graph until we reach a SHI-splittable role assertion. We show that this strategy is sufficient for entailment of atomic concepts. The formal foundations for these subsets of assertions have been set up before, where we show that, under some conditions, role assertions can be broken up while preserving soundness and completeness of instance checking algorithms. First, in Definition 4, we formally define an individual island candidate with an arbitrary subset of the original Abox. The concrete computation of the subset is then further defined below.

**Definition 4 (Individual Island Candidate).**
Given an ontology \( \mathcal{O} = (\mathcal{T}, \mathcal{R}, \mathcal{A}) \) and a named individual \( a \in \text{Ind}(\mathcal{A}) \), an individual island candidate, is a tuple \( \text{ISL}_a = (\mathcal{T}, \mathcal{R}, \mathcal{A}^{isl}, a) \), such that \( \mathcal{A}^{isl} \subseteq \mathcal{A} \).

Given an individual island candidate \( \text{ISL}_a = (\mathcal{T}, \mathcal{R}, \mathcal{A}^{isl}, a) \) and an interpretation \( \mathcal{I} \), we say that \( \mathcal{I} \) is a model of \( \text{ISL}_a \), denoted \( \mathcal{I} \models \text{ISL}_a \), if \( \mathcal{I} \models (\mathcal{T}, \mathcal{R}, \mathcal{A}^{isl}) \).

Given an individual island candidate \( \text{ISL}_a = (\mathcal{T}, \mathcal{R}, \mathcal{A}^{isl}, a) \), we say that \( \text{ISL}_a \) entails a concept assertion \( C(a) \), denoted \( (\mathcal{T}, \mathcal{R}, \mathcal{A}^{isl}, a) \models C(a) \), if for all interpretations \( \mathcal{I} \), we have \( \mathcal{I} \models \text{ISL}_a \implies \mathcal{I} \models C(a) \).

Please note that entailment of concept and role assertions can be directly reformulated as a decision problem over ontologies, i.e., we have \( (\mathcal{T}, \mathcal{R}, \mathcal{A}^{isl}, a) \models C(a) \iff (\mathcal{T}, \mathcal{R}, \mathcal{A}^{isl}) \models C(a) \). In order to evaluate the quality of an individual island candidate, we define soundness and completeness criteria for individual island candidates.
Definition 5 (Soundness and Completeness for Island Candidates).

Given an ontology \( \mathcal{O} = \langle T, R, A \rangle \) and an individual island candidate \( \text{ISL}_a = \langle T, R, A^{atl}, a \rangle \), we say that \( \text{ISL}_a \) is sound for instance checking in ontology \( \mathcal{O} \) if for all atomic concept descriptions \( C \in \text{AtCon} \), \( \text{ISL}_a \models C(a) \Rightarrow \langle T, R, A \rangle \models C(a) \). \( \text{ISL}_a \) is complete for instance checking in ontology \( \mathcal{O} \) if for all atomic concept descriptions \( C \in \text{AtCon} \), \( \langle T, R, A \rangle \models C(a) \Rightarrow \text{ISL}_a \models C(a) \).

We say that \( \text{ISL}_a \) is sound for relation checking in ontology \( \mathcal{O} \) if for all role descriptions \( R \in \text{Rol} \) and all individuals \( a_2 \in \text{Inds}(A) \)

\[
\begin{align*}
- \text{ISL}_a \models R(a, a_2) & \Rightarrow \langle T, R, A \rangle \models R(a, a_2) \quad \text{and} \\
- \text{ISL}_a \models R(a_2, a) & \Rightarrow \langle T, R, A \rangle \models R(a_2, a).
\end{align*}
\]

\( \text{ISL}_a \) is complete for relation checking in ontology \( \mathcal{O} \) if for all role descriptions \( R \in \text{Rol} \) and all individuals \( a_2 \in \text{Inds}(A) \)

\[
\begin{align*}
- \langle T, R, A \rangle \models R(a, a_2) & \Rightarrow \text{ISL}_a \models R(a, a_2) \quad \text{and} \\
- \langle T, R, A \rangle \models R(a_2, a) & \Rightarrow \text{ISL}_a \models R(a_2, a).
\end{align*}
\]

We say that \( \text{ISL}_a \) is sound for reasoning in ontology \( \mathcal{O} \) if \( \text{ISL}_a \) is sound for instance and relation checking in \( \mathcal{O} \). We say that \( \text{ISL}_a \) is complete for reasoning in \( \mathcal{O} \) if \( \text{ISL}_a \) is complete for instance and relation checking in \( \mathcal{O} \).

Definition 6 (Individual Island).

Given an individual island candidate \( \text{ISL}_a = \langle T, R, A^{atl}, a \rangle \) for an ontology \( \mathcal{O} = \langle T, R, A \rangle \), \( \text{ISL}_a \) is called individual island for \( \mathcal{O} \) if \( \text{ISL}_a \) is sound and complete for reasoning in \( \mathcal{O} \).

An individual island candidate becomes an individual island if it can be used for sound and complete reasoning. It is easy to see that each individual island candidate is sound for reasoning since it contains a subset of the original Abox assertions.

In Fig. 1, we define an algorithm which computes an individual island starting from a given named individual \( a \). The set agenda manages the individuals which have to be visited. The set seen collects already visited individuals. Individuals are visited if they are connected by a chain of SHI-unsplittable role assertions to \( a \). We add the role assertions of all visited individuals and all concept assertions for visited individuals and their direct neighbors.

In Lemma 1, we show that the individual island of an individual suffices to decide entailment of atomic concept assertions for an individual.

Lemma 1 (Individual Island Dependencies).

Given an ontology \( \mathcal{O} = \langle T, R, A \rangle \), for all named individuals \( a \in \text{Inds}(A) \) and atomic concept descriptions \( C \), if \( \text{ISL}_a \) is an individual island and \( \text{ISL}_a \not\models C(a) \) then there exists no individual \( \text{diff} \in \text{Inds}(A) \), such that \( \text{ISL}_{\text{diff}} \models C(a) \).

Below, we show in Theorem 1 that the computed set of assertions is indeed sufficient for complete reasoning.
Input: Ontology $\mathcal{O} = \langle T, R, A \rangle$, individual $a \in \text{Inds}(A)$
Output: Individual island $\text{ISL}_a = \langle T, R, A^\text{isl}, \ldots = \{C|C(a) \in A\}, \text{reflset} = \{R|R(a,a) \in A \lor R^-(a,a) \in A\}$, and pnsset is the set of all pseudo node successors of individual $a$.

Algorithm:
1. Let agenda = $\{a\}$
2. Let seen = $\emptyset$
3. Let $A^\text{isl} = \emptyset$
4. While agenda $\neq \emptyset$ do
   1. Remove $a_1$ from agenda
   2. Add $a_1$ to seen
   3. For each $R(a_1, a_2) \in A$
      1. $A^\text{isl} = A^\text{isl} \cup \{R(a_1, a_2) \in A\}$
      2. If $R(a_1, a_2) \in A$ is SHI-splittable with respect to $\mathcal{O}$ then
         $A^\text{isl} = A^\text{isl} \cup \{C(a_2) \mid C(a_2) \in A\}$
      3. else agenda = agenda $\cup \{\{a_2\} \setminus \text{seen}\}$
   4. For each $R(a_2, a_1) \in A$
      1. $A^\text{isl} = A^\text{isl} \cup \{R(a_2, a_1) \in A\}$
      2. If $R(a_2, a_1) \in A$ is SHI-splittable with respect to $\mathcal{O}$ then
         $A^\text{isl} = A^\text{isl} \cup \{C(a_1) \mid C(a_1) \in A\}$
      3. else agenda = agenda $\cup \{\{a_1\} \setminus \text{seen}\}$

Fig. 1. Schematic algorithm for computing an individual island.

Theorem 1 (Island Computation yields Individual Islands for Ontologies). Given an ontology $\mathcal{O} = \langle T, R, A \rangle$ and an individual $a \in \text{Inds}(A)$, the algorithm in Fig. 1 computes an individual island $\text{ISL}_a = \langle T, R, A^\text{isl}, a \rangle$ for $a$.

The proof is given in [11].

For each individual there is an associated individual island, and Abox consistency can be checked by considering each island in turn (islands can be loaded into main memory on the fly). Individual islands can be used for sound and complete instance checks, and iterating over all individuals gives a sound and complete (albeit still inefficient) instance retrieval procedure for very large Aboxes.

Definition 7 (Pseudo Node Successor). Given an Abox $A$, a pseudo node successor of an individual $a \in \text{Inds}(A)$ is a pair $\text{pns}^{a,A} = (\text{rs}, \text{cs})$, such that there is an $a_2 \in \text{Ind}(A)$ with
1. $\forall R \in \text{rs}, (R(a, a_2) \in A \lor R^-(a_2, a) \in A)$,
2. $\forall C \in \text{cs}, C(a_2) \in A$, and
3. rs and cs are maximal.

Definition 8 (One-Step Node).
Given $\mathcal{O} = \langle T, R, A \rangle$ and an individual $a \in \text{Inds}(A)$, the one-step node of $a$ for $A$, denoted $\text{osn}^{a,A}$, is a tuple $\text{osn}^{a,A} = (\text{rootconset}, \text{reflset}, \text{pnsset})$, such that $\text{rootconset} = \{C|C(a) \in A\}$, $\text{reflset} = \{R|R(a, a) \in A \lor R^-(a, a) \in A\}$, and pnsset is the set of all pseudo node successors of individual $a$. 


It should be obvious that for realistic datasets, multiple individuals in an Abox will be mapped to a single one-step node data structure. We associate the corresponding individuals with their one-step node. In addition, it is clear that one-step nodes can be mapped back to Aboxes (for the associated individuals we use a representative individual \( a \)). If \( Abox_a(osn^A) \models C(a) \) for a query concept \( C \) (a named concept), all associated individuals of \( osn^A \) are instances of \( C \). It is also clear that not every one-step node is complete for determining whether \( a \) is not an instance of \( C \). This is the case only if one-step nodes “coincide” with the islands derived for the associated individuals (splittable one-step nodes). Wandelt found that for LUBM in many cases islands are very small, and one-step nodes are indeed complete in the sense that if \( Abox_a(osn^A) \not\models C(a) \) then \( A \not\models C(a) \) (for details see [11]). In the following we assume that for instance retrieval, it is possible to specify a subset of Abox individuals as a set of possible candidates. If the set of candidates is small, with some candidates possibly eliminated by one-step nodes, then iterative instance checks give us a feasible instance retrieval algorithm in practice.

4 Answering Grounded Conjunctive Queries

In this section we will shortly describe an implementation of the introduced techniques with a triplestore database. As other groups we use the Lehigh University Benchmark or LUBM [5] for evaluating algorithms and data structures. This benchmark is an ontology system designed to test large ontologies with respect to OWL applications. With the LUBM generator, the user can generate \( n \) universities each consisting of a random number of departments and individuals. As the number of individuals and the number of assertions increases nearly linear with the number of universities, LUBM is an instrument to test the performance for query answering machines, especially for grounded conjunctive queries in a scalable Abox environment. If a system cannot handle LUBM with, say, a billion triples, it cannot deal with more complex scenarios occurring in future applications, either. The code we used in this paper for evaluating the optimization techniques is written in Java and can be downloaded at http://www.sts.tu-harburg.de/people/c.neuenstadt/. We store data in the triplestore AllegroGraph, which provides access to role instances (triples) w.r.t. RDFS plus transitive roles, i.e., role hierarchies and transitive roles are handled by AllegroGraph. Alternatively one could use materialization or query expansion in the OBDA style for role hierarchies. SPARQL is used as a query language for specifying queries to be executed on a particular triplestore database.

4.1 Setting up an AllegroGraph Triplestore

AllegroGraph is run as a server program. In our setting, data is loaded directly into the server, whereas islands as well as one-step nodes are computed by a remote program run on a client computer (we cannot extend the server program easily). In a first step the whole data system has to be set up before we can
start query answering. During the setup process, the communication between client and server system consists basically of sending SPARQL queries for data access required in the algorithm shown in Fig. 1 as well as sending AllegroGraph statements for adding additional triples (or changing existing ones) for storing islands and one-step nodes. Islands are indicated using the subgraph components of AllegroGraph triples (actually, quintuples).

The creation of one-step nodes is solved using hashvalues with a sufficient bitlength. We first compute a set from each island and immediately build a hashvalue and store it together with the specific island on the triplestore and build the specific one-step nodes. Identical hash values allow one to refer to identical one-step nodes.

Given the concept description \( C \) from the query and the named individual \( a \) from the tuple, we load the specific one-step node for \( a \) from the database and determine whether \( osn_a \) entails \( C(a) \). Depending on the outcome, three states are possible:

- \( osn_a \) entails \( C(a) \), then \( a \) is actually an instance of \( C \).
- \( osn_a \) entails \( \neg C(a) \) or does not entail \( C(a) \) and is splittable, then \( a \) is actually not an instance of \( C \).
- \( osn_a \) is not splittable, then the client has to load and check the entire island associated with \( a \) to find out whether \( a \) actually is an instance of \( C \).

Candidates for concept atoms are determined in our experiments by first doing a retrieval for a query \( q \) by executing \( skel(q) \) (see above for a definition). Bindings for variables in \( skel(q) \) define the candidates for retrieval with concept atoms. By eliminating all skeleton query result tuples that include individuals which do not belong to corresponding concept assertions used in the query, finally all remaining tuples are correct answers to the original conjunctive query.

Wandelt has already investigated the efficiency of Abox modularization techniques for an SQL database server. Here, instead, we work directly on an existing AllegroGraph triplestore, convert the large ontology step by step into small chunks and compare the generated modules with the local modularization of Sebastian Wandelt on the SQL server [11]. The processing time for one university is about 5000 seconds on AllegroGraph server, where it is like one minute for Wandelt’s approach. The modularization of one university takes nearly 200,000 queries. The decrease in performance is based on the huge number of SPARQL mini queries between server and the remote modularization client in the prototype implementation. Thus, only 5 universities are investigated for query answering experiments.

4.2 Evaluating Conjunctive Queries

For evaluating grounded conjunctive queries, LUBM provides 14 predefined test queries, which check several database criteria. We run the tests with an ontology specified in the description logic language \( SHI \). LUBM queries differ in the amount of input, selectivity and reasoning behaviour, for example by relying
on inverse roles, role hierarchy, or transitivity inferences. Selectivity basically means that the grounded conjunctive queries being considered have role atoms with large result sets to be reduced by atom queries, which we call less selective (proportion of the instances involved in the skeleton are high compared to those that actually satisfy the query criteria), or automatically excludes a lot of individuals, which is what we call a highly selective query. Rather than doing a join on the result sets of all atoms in a grounded conjunctive query, role atoms define candidates for concept atoms. Thus for selective queries, candidate sets for concept atoms are smaller. This per se reduces the number of instance checks that remain if, e.g., one-step node optimizations are not applicable (see above).

The result indicates that the less selective a query is w.r.t. role atoms, the more instance checks we need afterwards, and the more time consuming retrieval is (see Figure 2). Nevertheless, most of the LUBM queries are handled fast, even with a simple implementation for concept atoms with repetitive instance checks. Performance for queries 5 and 8 is increased with one-step node retrieval (with multiple individuals returned at a time, see above). Queries 6 and 14 contain only concept atoms and are handled fast only with one-step nodes.

In the LUBM dataset, explicit assertions about subrelations of degreeFrom are made (e.g., doctoralDegreeFrom). The relation degreeFrom is declared as an inverse to hasAlumnus. Thus, although, e.g., Query 13 contains a reference to University0 (asking for fillers of hasAlumnus), adding new universities with degreeFrom tuples with University0 on the right-hand side causes the cardinality of the set of fillers for hasAlumnus w.r.t. University0 to increase, i.e., having a constant in the query does not mean the result set to be independent of the number of universities.
In Figure 3 we see a direct comparison between highly and less selective queries (e.g., LUBM Queries 11, 13, and 8, respectively. Figure 3 shows the relatively high amount of time for answering less selective queries. This gap even increases with an increasing number of universities. The result will be improved in future work by applying Wandelt’s instance retrieval techniques with one-step nodes. To demonstrate that our skeleton query candidate generator is able to significantly improve the results for queries with high selectivity, we compare the approach of skeleton queries with the naive approach without skeleton queries in Figure 3. One can directly see the huge performance gain of the skeleton query. We avoid lots of instance checks and can therefore decrease the answering time by orders of magnitude in many cases.

5 Conclusion

In this work we extended the Abox modularization strategies of Wandelt and colleagues to the efficient use of grounded conjunctive queries on triplestore servers. Our prototype needs linear time to add information to the triplestore in a setup phase. Therefore we were not able to run queries on billions of triples. We conclude that island computation needs to be built into the triplestore software itself and cannot be done from a remote client.

In the average case, the size of the individual island (with respect to the number of assertion in its Abox) is considerably smaller than the original Abox. In our experiments the size is usually orders of magnitudes smaller. Please note that these modularization techniques allow traditional description logic reasoning systems to deal with ontologies which they cannot handle without modularizations (because the data or the computed model abstraction does not fit into main memory).

In addition, the evaluation of the prototype showed how grounded conjunctive queries on triplestore servers w.r.t. expressive ontologies (SHI) can be implemented using only a small size of main memory. The main strategy is to use a skeleton query and try to keep the necessary amount of instance checks in the second step as small as possible. If the number of results for less selective skeleton queries gets larger, the number of instance checks increases rapidly. Results obtained with the techniques discussed in this paper are sound and complete. Note that query engines investigated in [5] are incomplete. While selective queries are handled quite well, query answering times in the case of less selective skeleton queries need cannot be handled by repetitive instance check. In these cases we will use one-step nodes as investigated by Wandelt.

We would like to emphasize that the proposed optimizations can be used for parallel reasoning over ontologies [13]. This will be further investigated in future work such that ABDEO will become possible for practically relevant datasets and ontologies that are more demanding than LUBM.
References


