UNIVERSITÄT ZU LÜBECK

Institute of Information Systems
University of Lübeck

An Engine for Ontology-Based Stream Processing Theory and Implementation

Submitted by Christian Neuenstadt
from Hamburg, Germany.

Lübeck, February 2018

C. Neuenstadt

From the Institute of Information Systems
of the University of Lübeck
Director: Prof. Dr. Ralf Möller

An Engine for Ontology-Based Stream Processing Theory and Implementation

Dissertation
for Fulfillment of
Requirements
for the Doctoral Degree
of the University of Lübeck

from the Department of Computer Sciences

Submitted by

Christian Neuenstadt
from Hamburg, Germany.

Lübeck, February 2018

Chair:	Prof. Dr. Andreas Schrader
First referee:	Prof. Dr. Ralf Möller
Second referee:	Prof. Dr. Martin Leucker
Date of oral examination	06.02.2018

Approved for printing. Lübeck,

Abstract

In recent years various technologies have been developed to access ontology-based temporal and streamified data. This work is a contribution to these efforts by demonstrating the stream-temporal access on relational data with query transformations based on a new query language STARQL and the implementation and evaluation of a prototypical framework for accessing different data sources. As especially designed for industrially sensor network scenarios, STARQL provides access on historic data for reactive diagnostics and streamed data for continuous monitoring to solve requirements of industrial engineers in predictive and real-time scenarios. We explain how to transform STARQL queries w.r.t. mappings into SQL queries on the theoretical and practical side, while evaluating temporal sequences and joins of historical and live streamed data. Finally, we show experiments based on our STARQL prototype and modern database engines such as the PostgreSQL DBMS or distributed Big Data systems such as Spark SQL and Spark streaming to prove the implementability and feasibility of our approach.

Keywords: stream reasoning, ontology-based data access, monitoring, unfolding, safety, temporal reasoning

Zusammenfassung

In den vergangenen Jahren wurden bereits einige Methoden entwickelt, um einen ontologobasierten Zugriff auf zeitliche und strombasierte Daten zu ermöglichen. Diese Arbeit soll einen weiteren Beitrag leisten, um den strom- und zeitbasierten Zugriff auf relationalen Daten mittels Anfragetransformationen und einer neuen optimierten Anfragesprache namens STARQL zu demonstrieren. Die speziell für den industriellen Einsatz entwickelte Sprache STARQL ermöglicht sowohl den Zugriff auf historische Daten für eine reaktive Diagnostik als auch auf geströmte Daten für ein kontinuierliches Monitoring, um Ingenieure optimal im Hinblick auf Analysen und Vorhersagen in Echtzeit-Szenarien zu unterstützen. In diesem Kontext erklären wir wie eine Transformation von STARQL-Anfragen mit Hilfe von deklarativen Abbildungen in SQL-Anfragen umgesetzt werden kann, sodass sowohl Information über zeitliche Sequenzen als auch die Verknüpfungen von historischen und Echtzeit-Daten ermöglicht wird. Schließlich analysieren wir in einer prototypischen Umsetzung das in dieser Arbeit implementierte STARQL Framework und zeigen anhand von Beispielen und Anwendungen in modernen Datenbanksystemen (wie z.B. das verteilte Big Data Framework Spark) die Implementierbarkeit und Machbarkeit unseres Ansatzes im Vergleich zu ähnlichen Systemen.

Contents

1. Introduction **1**
1.1. Challenges for Stream Processing 1
1.2. Research Problems and Scope of Work 2
1.3. Outline 4

2. Preliminaries **5**
2.1. Main Concepts of Stream Processing 5
2.1.1. Continuous Queries on Data Streams 6
2.1.2. Data Stream Model 7
2.1.3. Event Stream Processing Model 8
2.1.4. Window Processing 10
2.1.5. Stream Operator 11
2.1.6. Lambda Architecture 12
2.2. Data Stream Management Systems and Query Languages 13
2.2.1. TelegraphCQ 13
2.2.2. NiagaraCQ 14
2.2.3. OpenCQ 15
2.2.4. Tribeca 15
2.2.5. Aurora/Borealis 15
2.2.6. STREAM 17
2.2.7. Tapestry 17
2.2.8. StreamCloud 18
2.2.9. Exareme 18
2.2.10. PipelineDB - SQL 20
2.2.11. Spark 21
2.2.12. Flink 23
2.2.13. Summary and Overall Comparison 24
2.3. Description Logic and Semantic Representation 28
2.3.1. Description Logics 29
2.3.2. Ontologies for Sensor Networks 36
2.3.3. The SPARQL Query Language 38
2.4. Ontology Based Data Access 43
2.4.1. Classical OBDA 45

2.4.2. Query Transformation for Access on Static Data 46
2.4.3. ABDEO 57
2.4.4. Temporalizing OBDA 58
2.4.5. Streamifying OBDA 60
2.5. Stream Based SPARQL - Extensions 61
2.5.1. Streaming SPARQL 61
2.5.2. C-SPARQL 62
2.5.3. CQELS 62
2.5.4. SPARQLStream 64
2.5.5. EP-SPARQL 65
2.5.6. TEF-SPARQL 65
2.5.7. RSP-QL 66
2.6. Comparison of Semantic Streaming Languages 67
2.6.1. General Semantic Models for Streams 68
2.6.2. Benchmarks for Linked Data 71
2.7. Concluding Remarks 77

3. A New High Level Stream Query Language: STARQL 79
3.1. OBDA Challenges in Sensor Measurement Scenarios 80
3.1.1. Optique - Use Case 80
3.1.2. Natural Query Examples 85
3.1.3. A New Query Language for Streams? 88
3.1.4. Resulting Problems and Hypotheses of this Work 89
3.2. Introduction to STARQL 91
3.2.1. Introduction of STARQL by Example 91
3.2.2. STARQL Stream Operators 96
3.3. Formal Syntax and Semantics 109
3.3.1. General STARQL Syntax 109
3.3.2. STARQL HAVING Clause Syntax and Safety Criteria 112
3.3.3. STARQL Semantics 119
3.3.4. Comparison of STARQL to SPARQL Syntax and Semantics . 123
3.3.5. Expressing Temporal States with STARQL HAVING Clauses 125
3.4. Concluding Remarks 128

4. Transformation of STARQL into Queries for Relational Systems 129
4.1. Transformation of Window and Sequencing Operators 131
4.1.1. Window Transformation for Historical Queries 132
4.1.2. Window Transformation for Continuous / Real Time Queries 134
4.2. Rewriting and Unfolding of STARQL HAVING Clauses 135
4.2.1. An Example Transformation for STARQL Having Clauses . . 137
4.3. Additional Transformation of STARQL Operators 143

4.4. Concluding Remarks 146

5. Querying Relational Streaming Engines with STARQL 147
5.1. Implementation of a STARQL Streaming Engine 147
5.1.1. Transformation Module 149
5.1.2. Query Processing 149
5.1.3. Serialization 152
5.2. Test Dataset 153
5.2.1. Data Schema and Example Data 153
5.2.2. Ontology 155
5.2.3. Mappings 156
5.2.4. Queries 157
5.3. Implementation of the Ontology Based Streaming Back End Adapter 160
5.3.1. Experiments on PostgreSQL Back End 161
5.3.2. Experiments on Exareme 166
5.3.3. Experiments on Spark 168
5.3.4. Experiments on PipelineDB 170
5.4. Concluding Remarks 172

6. Evaluation of Query Processing with STARQL 173
6.1. Functionality Evaluation - Comparison of RDF Stream Processing Engines 173
6.1.1. Comparing RDF-Stream Query Languages 174
6.1.2. Comparing RDF based Streaming Systems 175
6.1.3. Evaluating Functionalities in a Benchmark 176
6.1.4. Discussion of the Functionality Evaluation 178
6.2. Evaluation of Rewriting and Transformation 180
6.2.1. (Non) Reification of Direct Mapping and Time 180
6.2.2. Evaluation of Transformation and Delays 183
6.2.3. Discussion of the Transformation Process 185
6.3. Evaluation of Query Execution 186
6.3.1. Evaluation of Historical Queries 186
6.3.2. Scalability of Query Execution 188
6.4. Discussion of Evaluation Results 189
6.4.1. Evaluation of Functionalities 189
6.4.2. Feasibility of the OBDA Approach 191
6.4.3. Efficiency and Scalability of the Implemented Approach . . . 192

7. Conclusion 193
7.1. Contributions 194
7.2. Outlook and Future Work 195

Appendices 197

A. Transformation of Example Queries 199

B. Distributed Window execution with pl/pgSQL 205

C. SRBench - Queries expressed in STARQL 215

Bibliography 221

Listings 243

1. Introduction

The tremendous hardware development in recent years has produced smaller, cheaper and more efficient sensor devices than ever before. As a result, sensor measurements have become ubiquitous and allow us to improve our life in many scenarios, where monitoring and complex analytics of sensor data can help to overcome daily problems.

This has lead to an emerging increase in applications for managing sensor inputs of many different kinds. Some of these examples are weather observations [84], health monitoring (e.g. heart rate or blood pressure) [107], financial markets (online analysis of stock prices) [249], network traffic monitoring [79, 218] or sensors in smartphones and mobiles (e.g. GPS, accelerometer and compass) [151]. Besides real physical sensor units, current data streams are often produced by services on the web, e.g., messages on various topics produced by humans or machines on Facebook or Twitter with more than 500 million tweets a day [217].

An increasing amount of live data has revolutionized the way we are looking at database systems. Where earlier databases were just stores that evaluated its data inside, we are now using systems, which are evaluating incoming data in real time just as it is arriving. This brought up the area of general stream processing [98, 105] with respect to relational and non-relational data.

1.1. Challenges for Stream Processing

In general the increasing amount of data is often named by the simple buzzword *BigData* [72]. Typical new challenges arriving together with BigData are are measured in three dimensions [152]: *volume* (accumulation of data over time), *velocity* (rapidly increasing input) and *variety* (data in different formats from different locations). To access on the one hand large amounts of already stored or recorded data and on the other fast changing input streams becomes an increasingly difficult task, especially if the data access has to be formulated in a single query. Although this problem has already been addressed in some approaches [161], it stays a major challenge for industry.

For example, the Optique Project [102] focuses on a use case that is provided by Siemens[1] and encompasses terabytes of temporal data, as well as many gigabytes of incoming turbine data from thousands of sensors per day. The combination of data from different sources leads to complex query formulation problems and requires up to 70% of an IT experts time for assessing the quality of data. A restriction to predefined queries in that scenario, which can only be extended by IT experts, was identified as the major bottleneck for providing the requested data and turns out to be very expensive in terms of time and money.

A possible solution to this problem is the so called *Ontology-Based Data Access* approach (short OBDA) [75, 202]. It allows the user to formulate queries on an abstracted semantic layer in a simpler query language, by using underlying transformation services that translate each query into appropriate representations for the (possibly distributed) database engines. It is based on two steps: a rewriting of the query w.r.t.an ontology and an unfolding into a target algebra. The use of an ontology allows the users to formulate their requirements by an enriched conceptual model, which has been specified directly for the problem domains and to receive answers in the same enriched form, while the queries are translated without any notice of the user or intervention of IT-experts in the backend. The general OBDA approach is well established in the field of static or non-temporal data, has been investigated for some approaches on temporal data [22, 25], and implemented in very few preliminary cases for streaming data [58, 59].

Streams and temporal data are strongly connected and thus, require similar additional temporal (logic) operators in their query language. However, processing of data differs. While temporal data can be evaluated by batch processing, the evaluation of streams is a reactive and continuous activity. Both, the processing and temporal operators in ontology based query languages are still part of ongoing and challenging research.

1.2. Research Problems and Scope of Work

Several approaches and query languages have been developed in the recent years that directly access streams containing ontology based data [30, 189]. On the other hand, there has only been one application that lifts relational stream engines onto an ontological level by query transformation techniques [58], as used in the OBDA approach for static data. Nevertheless, most of these approaches treat assertions in a single temporal window as equal with respect to time or just as another *static* attribute. The coexistence of attributes with unclear temporal relations in a slided

[1]http://www.siemens.com

time window can lead to unintended inconsistencies. For example, say, we have collected several measurements from different time points of a single sensor in one temporal window, then an consistency check would fail, as we require the role, which connects sensors and values, to be functional in measurement scenarios.

Although a lot of research has been accomplished on combining streaming and historic data in classical scenarios [161], there is currently no connection between ontology access on streaming data and temporal data in a single application or query language.

Derived from the previous observations, we have identified the following research problems for ontology based stream access:

> A query language on streams should be *time-aware* and also should provide real time based operators with underlying temporal semantics. It should allow for a formulation of temporal patterns and instruments for data analytics in industrial scenarios.

In order to solve this problem, we present a new query language for accessing streams with respect to ontologies: STARQL[2]. With STARQL we push the idea of traditional window operators further by defining finite sequences of temporal states for each window. Additionally we define appropriate tools that allow us to use predefined patterns on possible sequences, based on linear temporal logics. Moreover, we try to combine the most important operators, known from ontology database access and data analytics, to enable aggregations and temporal reasoning. The sequencing strategy for windows that is required to avoid inconsistencies, as argued above, makes rewriting and, more importantly, unfolding of STARQL queries a challenging task.

> A query language based on ontologies should enable a possible rewriting and unfolding technique, which is executed w.r.ṫontologies and mappings, to guarantee flexible access on different state of the art data processing backend systems.

After having mentioned the definition of temporal sequences and operators in our query language, one may ask whether it is still possible to rewrite and unfold one STARQL query into a single backend query, formulated directly in the query language provided by the backend systems. In fact, we demonstrate that such a end-to-end transformation is possible in general for relational database systems and in particular for different state of the art streaming and non-streaming database systems (demonstrated for Apache Spark).

[2][S]treaming and [T]emporal ontology [A]ccess with a [R]easoning-based [Q]uery [L]anguage

Finally, we also would like to guarantee that our transformed STARQL queries for temporal analytics can be executed efficiently on the backends for temporal (historical) and streaming data.

> A query language for streams and temporal analytics in industrial settings should provide combined access on temporal and streamed data in a distributed setting with large volumes of (historical) data.

Considering reasoning on temporal data for reactive diagnostics in industrial settings, it can be shown that a window based approach leads to desirable solutions as well. While for reactive diagnosis the recognition of patterns over a dataset is relevant, we use windows as potential templates to evaluate small subsets of data in one query. Thus, as the processing of recorded data is not bound to a real-time execution, internally, all window instances could be evaluated in parallel.

Further, we show a distributed execution of this approach for STARQL queries and different backends, which can handle the evaluation of real time and historical data.

1.3. Outline

The further thesis is structured as follows. In Chapter 2 we give an overview on relevant state of the art technologies, also relevant for a better understanding of our work. Those include: processing of streaming data, ontology based access on relational data and a comparison of different approaches for stream access with respect to ontologies, together with available benchmarks in that area. Chapter 3 presents our new query language and framework STARQL with a detailed view on its operators, as well as its syntax and semantics. We proceed in Chapter 4 with an description of the rewriting and unfolding process used to translate the explained temporal operators. Chapter 5 presents a prototypical implementation of a STARQL query transformation and processing engine based on work in previous chapters. We explain how rewritings and unfoldings are arranged in the architecture and how the transformation results are executed on different backend systems. We give a detailed evaluation with experiments in Chapter 6. Finally, in Chapter 7, we show our overall conclusions and give an outlook on possible directions for future research.

The presented work in this thesis has been partially published or presented on international conferences [131, 135, 181, 182] and workshops [168, 175, 183]

2. Preliminaries

The following chapter gives a basic overview on current data stream management systems on the one hand and ontology based streaming systems on the other. Furthermore, we explain the classical approach for translating ontology based queries into relational query languages (e.g. SQL) and give a small survey on current state of the art technologies in the field of RDF stream access.

Therefore, after a short introduction on the general streaming approach in Section 2.1, we proceed in Section 2.2 by giving an overview on DSMSs and a theoretical description of the classical OBDA approach with respect to mappings in Section 2.3. Finally, we conclude with already existing approaches for ontology based stream access (Section 2.5) and some possible benchmarks for an evaluation of these systems in Section 2.5.

2.1. Main Concepts of Stream Processing

In this section we would like to give a brief introduction to the basic use of streams in the literature. Processing streaming data focuses basically on evaluating continuous queries on data streams such as raw sensor data from sensor networks measuring information such as temperature, pressure, network monitoring, heart rates in a medical system or more high level streams connected to stock market tickers. In an even more high level fashion these streams can be abstracted further and developed into complex event streams that are processed by a so-called event processing engine. For example, if a temperature sensor exceeds a specific value for a certain amount of time, the current signal could be registered as an alert event, which is then sent over an additional stream. For all of these mentioned applications above, loading the arriving data into a further data base management system appears no longer useful as current DBMS are not designed for rapid high speed and continuous load or even infinite amounts of data coming over an input stream [27], and therefore a new strategy involving so-called continuous queries has been used in different systems [222].

We would like to provide a general overview on the area of processing streaming data with its related and current work by giving a short background on the notions, structures and query languages that these systems have established.

In our scenario for data streams, the main issue relating the data is that some or all of the input is not stored in main memory at a time and can therefore not be accessed completely. Compared to standard relational databases, there are additional differences concerning data [27].

- All stream elements arrive online and thus, the system has no control on the order or arrival times of the data for any input stream.
- The size of any data stream is unknown and could be even infinite, therefore a storage of the complete stream is impossible.
- As soon as a data stream element has been processed, it can not be retrieved or reproduced again, unless it is explicitly stored in memory or on disk, which is commonly not effective regarding its overall size of data.

A system designed for querying data streams does not mean that static data does not exist. Actually the data stream management system is often combined with a standard relational data base, which makes it possible to join window data and additional static data.

2.1.1. Continuous Queries on Data Streams

The first idea for handling continuous data on append-only databases, already used in 1992 [222], was about using continuous queries, which can be distinct from traditional database queries as one-time queries, which are evaluated only once over a snapshot of the dataset, with a single answer set. Continuous queries, on the other hand, are evaluated regularly over time with regular answers on changing windows and a just arriving dataset [28]. Nevertheless, the arriving dataset from continuous streams may be stored in relational data bases, but even if there is enough storage available on the hard drive, performance issues may require that memory be available for computing answer sets for blocking operators such as aggregates for computing the average of data values from a longer time period. Another option instead of saving the retrieved data to disk is providing the resulting answer as a new continuous stream [28]. An example architecture for continuous queries is shown in Figure 2.1. The schema shows different components of a typical processing engine. A query is registered to the system, which regularly evaluates changing window snapshots of incoming data. Streaming engines often support also caches

for aggregation operators (e.gḟor measuring average values) to aggregate data over a longer time period or include data from other permanent stores.

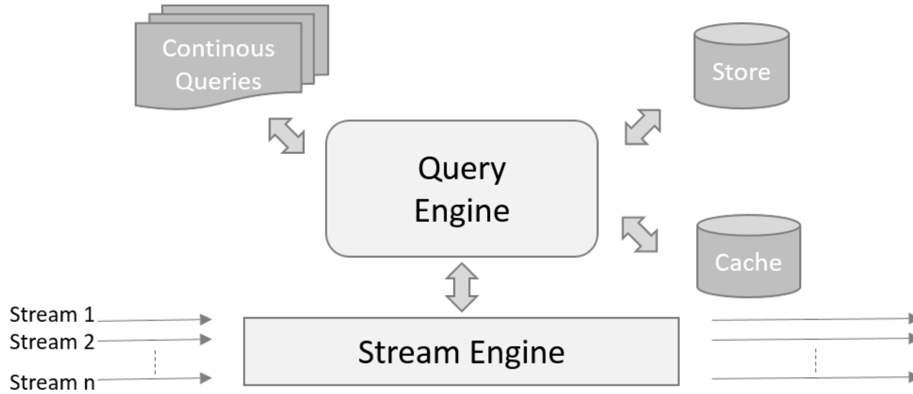

Figure 2.1.: Schema of an architecture for Continuous Queries

2.1.2. Data Stream Model

The use of simple continuous queries was sufficient for non complex cases, where relational query languages without specific streaming capabilities could still be used like in the case of Tapestry in 1992 [222], when references to relations were replaced by references to streams, while tapestry was even limited to append-only databases. However, as the complexity of continuous queries began to grow by the use of additional constructs such as aggregations, subqueries and joins between streams, relations or both, more advanced query languages with specific temporal semantics and operators were required [17][241]. Successor systems introduced continuous streams as unbounded sequences of tuples [105]. Each of these tuples is attached to some time information, which could be a simple index indicating an ordering in time or an explicit timestamp. In the STREAM project [15], assuming a discrete time domain T, a stream is defined as follows.

Definition 1. ***Stream:***. *A stream S is a possibly infinite bag (multiset) of elements* $\langle s, \tau \rangle$*, where s is a tuple belonging to the schema of S, and* $\tau \in \Gamma$ *is the timestamp.*

In the case of an explicit timestamp, several options are possible. While it also induces an implicit order, it could describe tuples at their arrival to the system (transaction time) or an observation time for the actual event (application time), but in most cases a simple production time is chosen, defining the time at which the data has been generated.

Regarding that view, it is not necessarily the case that exactly one tuple arrives per timestamp, each timestamp can be bound to a set of unbound tuples and vice versa. With beginning of the 2000s several new systems appeared, supporting the view on streams in Definition 1, including the mentioned system STREAM [15], TelegraphCQ [71] and Aurora [3].

A typical use case for these models is the sensor measurement scenario, where observations from sensor networks (e.g. temperature, pressure, speed, coordinates) are sent as timestamped tuples to the streaming engine. We illustrate a possible stream model in Figure 2.2. One can see different observations for each sensor stream, while in each stream different sets of unbound tuples are connected to timestamps. The streams can differ in the size of tuples per timestamp or in the rate at which the timestamps arrive, depending on a higher or lower sampling rate.

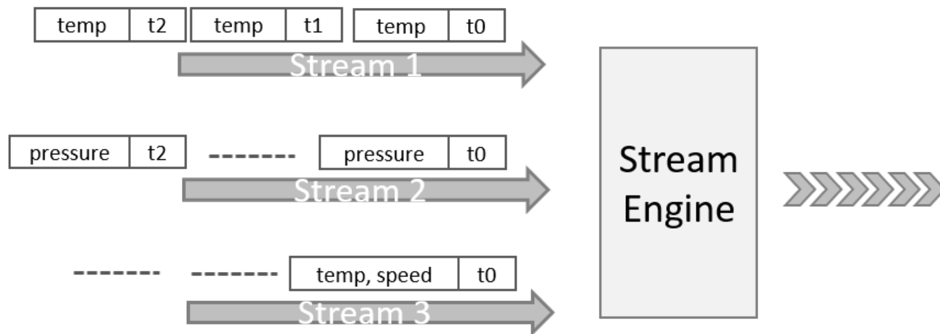

Figure 2.2.: Schema of an architecture for continuous queries

Although the described handling of timestamps and data tuples is a common model for data streams, there are several ways for individual abstractions and handling of streams regarding the data tuples. Instead of just sending unbound tuples like in the mentioned system STREAM [15], they can be sent as abstract objects or datatypes. The representation of data is thus independent from the stream model and sent in some systems as messages in XML notation for each tuple as in NiagaraCQ [73], other systems are more event based with abstract models such as in Esper [92].

2.1.3. Event Stream Processing Model

As mentioned before, an event stream model differs from the raw data stream level by its level of abstraction. Where data stream management systems handle raw data management, event processing system process abstractions of observations. A

more advanced way of processing events is complex event processing or CEP for short. The CEP engine subscribes to sources, also called observers, and manages the dataflow to sinks or consumers. In between it filters and combines information to explain what is happening on a high event level and notify its sinks [83]. CEP systems are highly tailored to detect complex patterns of incoming data involving sequencing and ordering that is a limitation of pure data stream management systems. Additionally, they rely on the ability to specify composite events through patterns and matching incoming notifications using an internal event processing network that includes event processing agents (see Figure 2.3) on the basis of their content and on some ordering relationships between them.

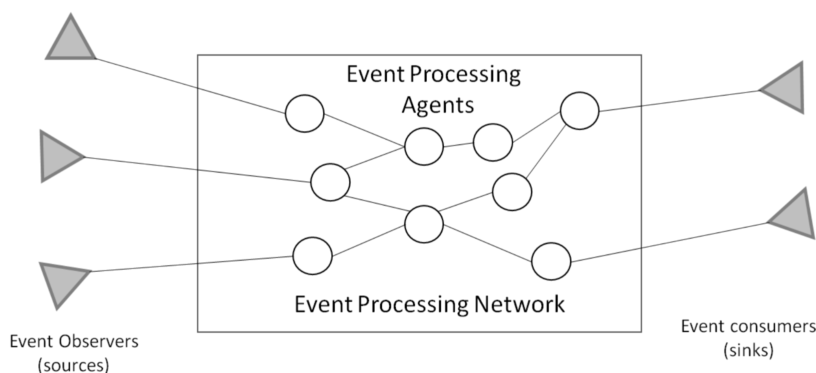

Figure 2.3.: General schema of a CEP network

Other indicators used for encoding complex events can be overlapping events, negation, disjunction and several more [82]. Some of these abstract processors have been formalized in [114]. Its authors describe nine different basic processing units in a formal way, including pass through of variables, emitting of constants, monadic or dyadic functions, filter and window functions. Furthermore, they explain how these processors can be combined to represent *regular expressions*, *finite-state automata* or *linear temporal logic*, which are three major languages that are commonly used to express monitoring specifications in *runtime verification* contexts.

The term runtime verification (or runtime checking) describes the monitoring of invariants or other conditions in software systems to ensure the correctness of software modules. Potential applications in these cases are system tests, debugging, verification and logging of errors. This technique is also applicable in final execution environments and embedded systems as shown in [239]. The paper also discusses cases of monitoring and runtime verification for different kinds of embedded systems, e.g., hardware/software, hybrid and on-chip monitoring.

Beyond that, also new types of events or patterns are defined by event processing

systems for realizing more advanced data mining steps, although the system does not necessarily depend on events only. And thus, different kinds of data input can be combined, naming events, data streams or just stored data in a relational data base.

2.1.4. Window Processing

As we already mentioned above, in comparison to standard relational databases, data stream management systems are unable to provide all data at any given point in time. A common solution to that problem is a computation on the latest arriving tuples only and therefore limit the scope of the data being processed. This is called the *window query model*. We suppose that for a given query q the latest tuples of data arrive at time τ, then the window relation $W(\tau)$ provides all tuples arriving in the interval of $(\tau - \delta, \tau]$, while δ indicates the width of window W. An aggregate function such as AVG can then be applied to $W(\tau)$. There exist several different types of window functions $W(\cdot)$. The following criteria for their classification can be found in [15].

Movement of the Window Endpoints One criterion is described by a fixed or moving window end. Two fixed ends define a *fixed window*, while two sliding endpoints define a *sliding window*. One fixed endpoint and one moving endpoint define a *landmark window*.

Time-Based vs. Tuple-Based *Time-based windows* (also called *physical*) are defined for specific time intervals (e.g., 7 minutes), while *tuple-based* (or *logical*) *windows* end with with a defined number of tuples (e.g., 7 data tuples).

Update Interval Windows can be updated whenever a new tuple arrives (called *eager update*). For any other update interval we talk about a *jumping window*. For update intervals larger or equal to the window size, we call the window a *tumbling window*. Jumping windows are not necessarily updated in a tuple-based way, in many cases a periodic time based update interval is used instead.

These window definitions have been implemented in many systems including STREAM, NiagaraCQ or TelegraphCQ, although no clear and formal unifying semantics has been given. A first approach on defining window operators was introduced in [241], whose authors propose three different window operators transforming streams into relations and vice versa.

2.1.5. Stream Operator

The data streaming system STREAM [241] has introduced stream semantics within its query language CQL[1] to express stream operators that became a standard view in the area for years. CQL [16] uses SQL constructs, which express a transformation from stream into a relational context, and enables evaluation on relations and transforms their result back into an output streams (see Figure 2.4).
The three classes are not explicit operators, but can be seen as a "black box" abstraction with generic properties.

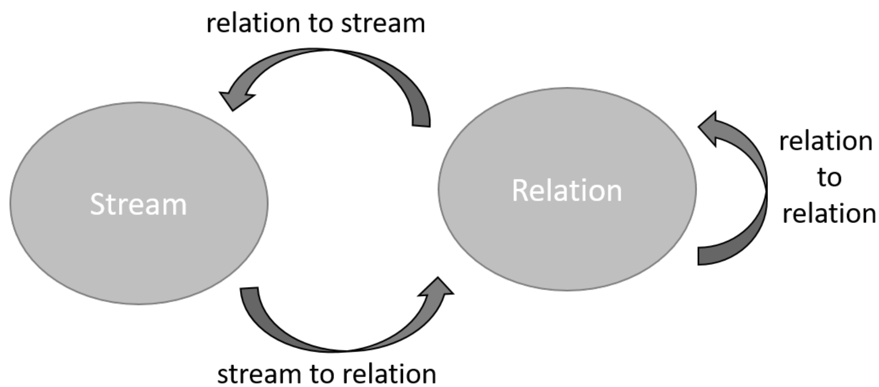

Figure 2.4.: General schema of CQL window and stream operator

Stream to Relation. A stream-to-relation operator takes a stream as input and produces a relation as output, which is an abstraction of window operators using a *sliding window* approach and can be expressed using window specifications, such window *width* and *slide*.

Relation to Relation. A relation-to-relation operator takes one or more relations as input and produces a relation as output. Most of the data manipulation is done using these constructs, which handles for example aggregation operators such as MAX or AVG, another part for data manipulation of this operator is filtering.

Relation to Stream. A relation-to-stream operator takes a relation as input and produces a stream as output. This operator manages the tuple output and is

[1]Continuous Query Language

specified by three different classes of operators:

1. The *Istream* or "insert stream" operator is applied to all tuples s of relation $W(\tau)$, whenever s is in $W(\tau) - W(\tau - 1)$, i.e. all tuples that have been inserted from the input stream at time τ.

2. The *Dstream* or "delete stream" operator is applied to all tuples s of relation $W(\tau)$, whenever s is in $W(\tau-1) - W(\tau)$, i.e. all tuples that have been deleted from the input stream at time τ.

3. The *Rstream* or "relation stream" operator is applied to all tuples s of relation $W(\tau)$, whenever s is in $W(\tau)$, i.e. all current tuples in the relation at time τ.

2.1.6. Lambda Architecture

In the range of BigData scenarios a new architecture was recently presented to combine the described processing of streams with popular batch processing methods for stored data, which is also known as *Lambda Architecture* [161]. It was designed to optimize the balance between latency, throughput and fault tolerance and is generally based on an append-only data model with an immutable data source that allows for different analytical tasks on the arriving data. Moreover, a Lambda architecture is constructed of the following three components:

Batch Layer. This layer precomputes results by using a distributed system that is capable of handling very large volumes of data. A de facto standard for these purposes is *Apache Hadoop* [210]. Its aim is to process the stored dataset and to generate batch processed views, which allow for faster query answering. A possible recomputation and replacing of existing views serves as a major fault tolerance advantage, if one computation gets lost during the process.

Speed Layer. Real time data streams are processed by the speed layer. It aims on a minimization of latency by providing views only for the most recent data and therefore, replaces the delayed view calculation of the batch layer by real-time processing. Although these results might be not as accurate or complete, they can be immediately provided when the data is received and replaced as soon as the batch layer's view is available. Often used stream technologies for these purposes are *Apache Storm* [226] or *Spark* [94].

Serving Layer. The outputs generated by batch and speed layers are joined and stored in the overlying serving layer, which serves as a direct connection

to incoming ad-hoc queries and returns views, constructed by the underlying layers. Used technologies are *Druid* [243], *Apache Cassandra* [150] and *HBase* [100] for merging speed-layer and batch-layer output.

While providing efficient access on recorded time series and live stream data, the system easily can become complex and hard to maintain. Each layer consists of different systems and code, but must be kept in sync in order to produce identical results. Therefore, other flexible streaming solutions are currently discussed that could provide the same processing and latency management in a single framework.

2.2. Data Stream Management Systems and Query Languages

In the last section we discussed a general model for streams. In this section we will go through a survey of data stream management systems using the paradigm of continuous queries [81]. SQL as a declarative standard language for relational databases inspired the industry to create several data stream languages and continuous query designs based on the declarative language as well. In the following we introduce some examples for the given stream languages by using a simple *measurement* input stream that shows *values* as a single attribute.

2.2.1. TelegraphCQ

TelegraphCQ [71] (CQ for continuous queries) is such a data stream management system, based on the standard language for relational databases SQL. It has been implemented as an extension of *PostgreSQL*[2] in C++ with an SQL like query language for streams, which its authors introduced as *StreaQuel*. As an extension, it relies on all relational operators of SQL, including aggregates. Furthermore, StreaQuel adds a specific new window operator called *WindowIs* to declare various types of windows.

Listing 2.1: Basic StreaQuel query (ST = start time)

```
1  Select AVG(value)
2  From Measurements
3  for (t = ST; t < ST + 50; t +=5 ){
4  WindowIs(Measurements, t - 4, t);
5  }
```

[2]http://www.postgresql.org/

The declarative language allows multiple *WindowIs* operators, one for each input stream, which is expressed in a *for* loop. Listing 2.1 shows an example for a single *WindowIs* operator. The loop starts at time ST and ends at ST. While the operator allows for a different window *width* in each stream, it does only allow one slide parameter in the *for* loop. By adopting an explicit time variable, TelegraphCQ enables users to define their own policy for moving windows. As a consequence, the number of items selected at each processing cycle is unknown in advance, since it is not possible to determine how many elements the time window contains.

As the *WindowIs* operator can be arbitrarily defined by free time variables, TelegraphCQ naturally supports the evaluation of historical data, which have to be read from disk and thus, can be significantly slower than the input stream. To keep up with the speed of the incoming live data a shedding technique is used. OSCAR[3], which has been implemented for TeleegraphCQ, saves different sizes of the data by multiple sample rates on the disk as summaries, which can be picked for replaying the historical data depending on the speed of the arriving live data, the system picks the right resolution level to use in query processing [70, 196].

2.2.2. NiagaraCQ

NiagaraCQ [73] is a system developed for high-level access on internet based data retrieval, where the data is stored in rapidly changing XML data sets. Therefore, it extends an XML based SQL-like language called *XML-QL* [90] by adding transformation rules and constructors for creating a continuous queries or deleting it. Rules are valid for defined time intervals and are either evaluated periodically or when changes are retrieved from the sources. Defined actions can be performed each time a specific rule is evaluated positively. Instead of creating a new output stream the result data is appended into a table and can either be retrieved by users on demand, or users can be notified if new results are available by email. The kind of data sources distinguishes NiagaraCQ from other DSMS systems, as they are widely distributed internet sources over a geographical area and as they are also XML based internet sources, they do not provide any explicit timestamps with the data. The NiagaraCQ engine itself runs in a centralized way, while for scalability and efficiency a caching algorithm is provided for reducing access time on distributed sources.

[3] Overload-sensitive Stream Capture and Archive Reduction

2.2.3. OpenCQ

OpenCQ [158] relies on the same data stream processing models as NiagaraCQ. Like NiagaraCQ, it is a data stream management system that was designed to work on internet database update monitoring as an web-event streaming system.
Its rules extend SQL queries that define operations on data, which are based on a trigger and a stop conditions that define the start and end point of a rule. OpenCQ has been implemented on top of the DIOM framework [157].

2.2.4. Tribeca

Tribeca [219] has been designed for network traffic monitoring and analytics. It uses rules for defining a sequence of operators that the input stream has to pass through. Those operators are based on three standard algebra operators, namely selection, projection, and aggregation. Additional operators are available for splitting or merging streams.

Tribeca also supports window operators for count and time based operations with different slide parameters. They specify the amount of input data that is being processed when using aggregates. As Tribeca does not use explicit timestamps, processing must be performed in direct arrival order. On the other hand, the amount of input data for each time window is impossible to know in advance. Therefore, a rule may be satisfied by more than one object in each window, while an element may be used for more than one processed window, as it is never explicitly consumed.

2.2.5. Aurora/Borealis

Aurora [3] is a DSMS that, in comparison to TelegeraphCQ and NiagaraCQ, uses an imperative language called *SQuAl*[4]. It can be described as a data-flow system that uses an interface based on boxes and arrows, where application administrators describe the system flow through a loop-free directed graph connected to processing operators. The graphical user interface of Aurora supports hierarchical collection of grouped boxes. A designer starts his design at the top-level of the hierarchy with a few super boxes on the screen. Then he can zoom into specific network groups and start the redesign by replacing it with boxes (i.e., operators).

[4] [S]tream [Qu]ery [Al]gebra

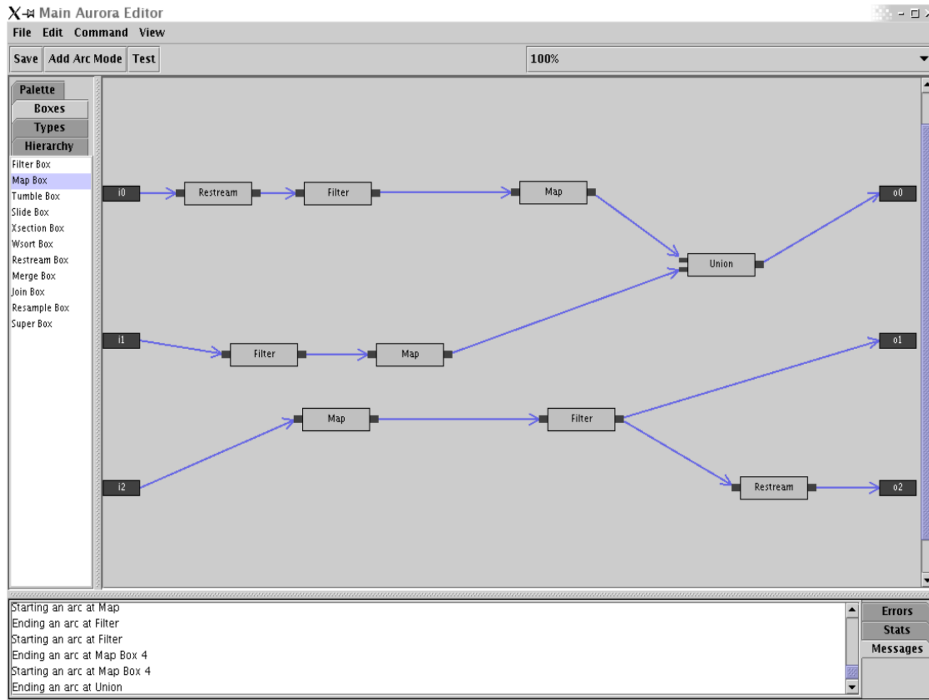

Figure 2.5.: Aurora graphical user interface [68]

Auroras query language SQuAl defines windowed operators or single-tuple operators, they either connect a single evaluation function to an input window or to a single data tuple, when operating on one item at a time. It contains built-in support for seven primitive operations or filter operators such as `SELECT` or `JOIN`, union and group by.

Although SQuAl provides multiple input and multiple output flows, there is no explicit split, but the administrator can connect the output to several inputs of other boxes by graph connections. On the other hand. there is an operator for stream merges available, which is called *union* operator. Additionally in the interface it is possible to add quality of service information to each output, which makes system behavior customizable for application requirements. So for example one application domain may require reduced answer precision in order to provide really fast response times. The design plan of the administrator is processed by a scheduler, which works as an optimizer on allocated resources for the different operators in order to manage their load and QoS constraints.

The project has been extended to different domains by merging the Aurora project with the Medusa project into the Borealis stream processor [2].

2.2.6. STREAM

We already mentioned CQL [17] in Section 2.1.5 for defining specific stream operator classes. CQL is a declarative and expressive SQL based language that was implemented for a prototype data stream management system at Stanford called STREAM[5] [15]. It introduces abstract semantics based on two data types, namely streams and relations, and three stream operator classes: *stream-to-relation*, *relation-to-relation*, *relation-to-stream*.
Those three abstract operator types describe generic properties of the streaming engine as black boxes for operators of that group (see Section 2.1.5). CQL expresses relation-to-relation operators in SQL. Its big advantage is that main parts of the definitions are realized by a widely used language. Stream-to-relation operators are derived from SQL-99 with additional window specifications, while three specific operators (IStream, DStream, and RStream) define relation-to-stream operations (compare Section 2.1.5). The STREAM system is able to compute query execution and schedule plans based on CQL rules, while taking performance criteria into account for optimization.
Additionally, load shedding techniques are used to overcome the problems of resource overload and limited memory by computing approximate answers for window joins when the memory is insufficient to keep the operator state [216]. CQL has been used to specify the Linear Road benchmark (see Section 2.6.2), which is typically proposed for the evaluation of data stream systems.

2.2.7. Tapestry

Continuous queries were first introduced for the *Tapestry* system [222] in 1992, that was designed for append-only databases without triggers, meaning that data is added as it arrives, but never removed.
Its original idea was querying for changes of mail messages or news articles on a standard database and sending notifications whenever new data matches the query, which as such could be implemented on any standard database that supports SQL. Users can write queries in a language called *TQL*[6] that is similar to SQL and can express usual queries on static data. Users can query the database with ad-hoc queries until it fits their requirements and then register it to the Tapestry system

[5][ST]anford st[RE]am dat[AM]anager
[6]Tapestry Query Language

to be executed in a loop (see Listing 2.2).

Listing 2.2: Basic Tapestry Algorithm for periodic query execution

```
1 FOREVER DO
2 Execute Query Q
3 Return results to user
4 Sleep for some period of time.
5 ENDLOOP
```

Conceptually, a TQL query is executed once every time instant as a one-time SQL query over the snapshot of the database at that time instant, and the results of all one-time queries are merged using set union.
The system is capable of executing queries periodically, while avoiding duplicates, but guarantees deterministic behavior for newly arriving tuples and evaluation results by its semantics.

2.2.8. StreamCloud

The *Stream Cloud* [111] framework is designed for scalable and elastic stream processing on top of the streaming engine Borealis [2] from section 2.2.5. It supports scalable and elastic processing of streams for large volumes of data on shared nothing nodes.

The StreamCloud compiler takes the query (based on the Borealis system) and uses a parallelization technique that splits queries into subqueries to deploy them on an independent set of nodes, while load balancers minimize the communication overhead.

2.2.9. Exareme

Exareme[7][227] is a system for elastic large-scale data-flow processing in the cloud [140] (formerly named ADP[8]). The system defines an elastic infrastructure in two parts, efficient dynamic allocation and deallocation of computational or storage resources and an elasticity model, which the authors call *eco-elasticity*, for a trade-off between time and money.

[7]http://www.exareme.com
[8]Athena Distibuted Processing

Queries can be defined by a combination of two declarative languages. *ExaDFL* allows definitions to describe the data-flow and parallelism (e.g., data distribution) and *ExaQL* is an extension of SQLite for query formulation. Additionally supported are user defined functions (UDFs) that allow user defined aggregations or virtual table functions defined in Python.

The UDF functions support an implementation of a data stream management system Ontop of the DBMS in the cloud [41]. To realize the data stream system a combination of several python functions (or UDFs) is required.

First, a UDF called *streamdata* is used to declare an arbitrary source (e.g., table, file, tcp socket) as an input stream, then one of two window operator functions is applied. The first function *slidingwindow* creates a virtual table with an additional column *windowId*, which groups incoming tuples in groups of smaller windows, indicated by a numbering for each window.

A second function (called *timeslidingwindow*) groups windows as a time-based window operator and requires additional timestamps in an extra column for each incoming tuple (see Listing 2.3 for an example). Both functions use *slide* and *width* as window parameters.

Listing 2.3: Example for a window operator with UDFs in ExaQL

```
1  create stream sensor1 as
2     select wid, value from
3        (ordered timeslidingwindow timecolumn:0 timewindow:60 frequency:1
            //Window parameters
4        select * from
5           (streamdata 'sensor1.csv')); //Streaming source
```

The window operator creates virtual tables with an extra column for the *windowId*. In the example in listing 2.3 a *timewindow* of width 60 seconds and slide of 1 minute is created, while the input is read from streamed csv-file.

Another important operator is an operator for joining streams, named *wcache*(i.e. window cache). For joining two or more streams the operator implements a hybrid hash and merge-join algorithm with the *wid* as key and a list of windowed tuples as values. As each stream is potentially infinite, the hash join is shifted in time by a merge-join algorithm [211].

2.2.10. PipelineDB - SQL

As TelegraphCQ was an extension of PostgreSQL back in 2003, so is *PipelineDB* [190] a new streaming extension fork for PostgreSQL as well. In its current version it includes a 100% of the PostgreSQL operators. While it retains their syntax, it adds specific new streaming features like *continuous views* and (as their sources) new incoming *streams* to the set of possible operators.

Streams are abstractions similar to table rows, which allow clients to provide input data to continuous views. Furthermore, as the interface of streams in PipelineDB is identical to the writing into tables in PostgreSQL, only functions such as *INSERT INTO* and *COPY* are allowed, which transforms PostgreSQL into a push-based streaming as a service system in comparison to several others, e.g., Exareme, which provides port listening. Thus, for PipelineDB we have to write our own client, that reads data (e.g. from port) and pushes it into the system.

Continuous views differ from normal SQL views in the way that they are producing different results each time they are queried depending on the incoming data streams. As in a regular streaming system *continuous views* only store parts of the incoming data, which is explicitly queried from it by a *select* statement. All other tuples from incoming streams are discarded immediately.

For each stream several continuous views can be set as so-called *targets* or removed as *targets* during runtime. Each time a tuple is pushed into a particular continuous view, its timestamp column is added automatically to the continuous view and called "arrival_timestamp", which means the exact system arrival time (also called transaction time).

The continuous view can be queried as any regular SQL view and therefore, no explicit window operator is provided. Window operations are simulated by referencing the timestamp column in the *Create View* declaration (see Listing 2.4).

Listing 2.4: Continuous view in PipelineDB with simulated window

```
1  CREATE CONTINUOUS VIEW recent_temps  WITH (max_age = '1 hour') AS
2     SELECT temp::integer FROM stream
```

The listing shows the creation of a continuous view with recent temperature values. The view can be seen as a window operator, although there is no real operator definition, because of the declared relation between the arrival timestamp and the

current system time that can be used to define the window width (a window slide parameter is missing). Continuous joins can be directly declared in the continuous view definition, but are restricted to joins between streams and static tables. Joins between two streams (as in the Exareme system) are currently not supported in version 0.9.3.

2.2.11. Spark

The *Apache Spark* project provides a framework for distributed cluster computing based on open source software, while originally developed at the University of Berkeley, it became an Apache top-level project in 2014. Spark provides an application programming interface centered on so-called RDD data structures for Java, Python, Scala and R.

RDDs

The underlying data structure for all Spark operations are so called RDDs [244] (Resilient Distributed Data sets), which are a collections of elements partitioned across the nodes of a cluster. While created from external sources (e.g., JDBC, files, network ports) RDDs support two types of operations: transformations, which create a new dataset from existing ones, and actions, which start a computation on RDDs and return values to the main program.

For example, map is a transformation that returns a new RDD representing the results of a function on each dataset element. An action in Spark on the other hand could be a *reduce* operation that uses an aggregation function over the RDD elements and returns a resulting value to the driver program.

As all transformations result in a new RDD, they are seen as immutable or read-only in general. Additionally, transformations are lazy, which means that they are only computed, if an action requires a result to be returned to the driver program. Until then, they are just remembered as functions that should be applied to some base set. This design enables Spark to run more efficiently if data is only returned for the final action and not for each transformation step. Spark also ensures fault-tolerance, as keeping track of all steps in the transformation chain allows a reconstruction in the case of data loss.

However, one may also use a persist method on RDDs to store the elements in memory or on disk for reproduction and much faster access in the case of future queries or previous system failures.

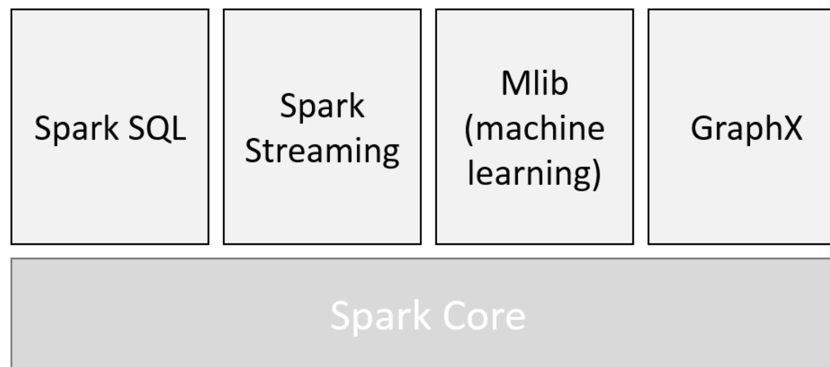

Figure 2.6.: Architecture of Spark components

Spark Architecture

The global Spark architecture consists of several partially depending components based on RDDs explained above and the Spark Core as an underlying system in Figure 2.6.

Spark Core. *Spark Core* is the founding library of the Apache project. It provides functionalities for task distribution, scheduling and I/O functionalities through an API for accessing the described RDD data structure. A *driver* program invokes executable transformations and actions on the RDD and passes the respective function to the Spark core, which finally executes all functions on the cluster in parallel.

Spark SQL. *Spark SQL* can be seen as an additional library on top of Spark Core for SQL that introduces new data concepts and structures. It supports reading and writing from JDBC or Apache Hive with HiveQL in *SparkSessions*. Therefore, the library provides several new transformations and actions, e.g, for creating views and sending a tuple result to the output. Besides some other features of Hive, the usage of indexes is not yet implemented due to the in-memory computation of Spark SQL.

Spark Streaming. *Spark Streaming* makes direct use of the fast scheduling, implemented in Spark Core, to enable its streaming analysis. It sorts the input data into small batches and performs RDD transformations of Spark on them. This architecture enables the application of the same functionalities for streaming analysis

as for batch analysis, while also allowing an easy implementation of lambda architectures [174]. On the other hand, this approach directly limits its latency equal to the processing duration of the small batch.

Spark MLlib. *Spark MLlib* is a distributed machine learning library for in-memory use on Spark, which makes it nearly nine times as fast compared to the disk-based implementation of Apache Mahout [163].

GraphX. *GraphX* is the fourth library for Apache Spark and supports distributed graph processing.

2.2.12. Flink

Flink [93] is a streaming system of the Apache Hadoop stack. It is based on the research project *Stratosphere* [7], which was originally started in 2010 as a collaboration of the Technical University Berlin, Humboldt-Universität zu Berlin, and Hasso-Plattner-Institut Potsdam. The Stratosphere fork became an Apache top-level project in December 2014. It is currently driven by the start-up company dataArtisans [9].

Flink Architecture and APIs

Apache Flink was directly developed as a distributed streaming engine and as such, follows a *streaming first* paradigm. It offers programming APIs in Java or Scala, which automatically compile and optimize the code into dataflow programs that are executed as batch and stream processing programs. It also offers APIs for distributed storage systems such as HDFS/YARN and for consuming data from message queues (e.g., Kafka).

The Flink framework provides different libraries similar to those in Spark, e.g., for its DataStream API (stream processing) a CEP or for its DataSet API (batch processing) the FlinkMLlib and Graph processing library respectively. A SQL API is offered in both cases.

Further, the API uses transformations similar to those in Scala (e.g., filter, map and flatmap), but also joins, grouping, definition of windows and aggregations. The execution itself is then done in so-called lazy processing (as also used in Spark),

[9] http://data-artisans.com/

meaning that the data loading and transformation operations are first added to the processing plan that is not executed, until the plan is explicitly triggered by an *execute* command.

Comparison to Apache Spark

Flink has been compared to data processing engines such as Storm [226] and Spark in recent benchmarks [74][213].

The results show a faster processing by Apache Flink in the case of data mining operations and the evaluation of relational data. The authors of [74] argue that the reason can be seen in the efficient processing of basic relational operators such as `GROUP BY` and `JOIN`. While Apache Spark on the other hand has advantages in its native processing of mini batches and therefore, the map and reduce operators in Spark are faster processed resulting in a significant higher throughput.

Moreover, Flink evaluates each streamed tuple as soon as it becomes available, which leads to a better overall latency in the streaming case, where Spark uses a stepwise behavior according to the processing of mini batches.

Despite of its performance disadvantages, the Spark framework also has some benefits towards Flink. Spark is embedded by Hadoop distributions such as MapR, Hortonworks or Cloudera and has a larger community of users and contributors that ensure its future development.

2.2.13. Summary and Overall Comparison

We gave a survey of 13 data stream management systems and eleven query languages. We have chosen the most popular streaming systems, which are based on relational query languages. Nevertheless, the list of systems could be extended with many more examples from the recent years (e.g., Odysseus [14]).

In Table 2.1 a comparison of the mentioned systems is shown. Most systems are based on relational data stream tuples. Only NiagaraCQ is used for monitoring web sources and transfers more abstract XML objects.

While all individual implementations are different, nearly half of the systems follow a periodic evaluation model (clock based). The table also shows that more than fifty percent of the systems allows for scalability, which is important for a high throughput. On the other hand also load shedding is supported for fast reading of historical data on disk.

Table 2.1.: Comparison of DSMS systems

Name	Data Type	Deployment Model	Clock	Implementation Specifications	Load Shedding
TelegraphCQ	Tuples	Clustered	Yes	Extension of PostgreSQL	Yes
NiagaraCQ	XML	Centralized	Yes	XML source monitoring	Yes
OpenCQ	Tuples	Centralized	Yes	NiagaraCQ + trigger conditions for rules	No
Tribeca	Tuples	Centralized	No	Network traffic analysis	No
Aurora/Borealis	Tuples	Clustered	No	Dataflow by graphs and boxes	Yes
STREAM	Tuples	Centralized	No	Stream ->Relation ->Stream operators	Yes
Tapestry	Tuples	Centralized	Yes	CQ on append only DB	No
StreamCloud	Tuples	Clustered	No	Scalable extension of Borealis	No
Exareme	Tuples	Clustered	No	Elastic SQLite extension with UDFs	No
PipelineDB	Tuples	Centralized	No	Extension of PostgreSQL	No
Spark	DataFrames	Clustered	Yes	Mini-batch processing on cluster	No
Flink	Tuples	Clustered	Yes	Parallel distributed dataflow	No

More differences can be seen for the used query languages in Table 2.2. A collection of seven declarative SQL like and two additional graphical or imperative languages are shown, which are used in the described data stream management systems.

The languages differ in the kind of supported window constructors. While only five of them support a window operator, most of them have one fixed window size. Only StreaQuel is able to move one window end only for a landmark window or use both fixed, for a fixed window. PipelineDB only supports slided windows, which means that one end is always fixed at the latest time point, while the other end reaches into the past. This does not allow for jumping or tumbling window, as no specific slide value is supported by the streaming engine.

Another criterion is the availability of joins from two perspectives. Either a join of two infinite data streams or a join of a single stream with a static table are possible. Some systems do not support the join of streams with static tables, while it is only one system (i.e., PipelineDB) that does not allow streams to be joined with other streams, which is a major disadvantage of the system. On the other hand, most systems support basic SQL operators such as `UNION` and `GROUP BY`.

Table 2.2.: Comparison of DSMS query languages

Name	**Type**	**Selection/Projection**	**Windows**	**stream join**	**static join**	**union**	**except**	**intersect**	**Aggregation**
StreaQuel	Declarative	Yes	Fixed, Landmark, Slide, Tumbling	Yes	Yes	Yes	Yes	Yes	No
NiagaraCQ	Declarative	Yes	No	Yes	No	No	No	No	No
OpenCQ	Declarative	Yes	No	Yes	No	Yes	Yes	Yes	No
Tribeca	Imperative	Yes	Slide, Tumbling	No	No	Yes	No	No	Yes
Aurora/Borealis	Graphs and Boxes	Yes	Slide, Tumbling	Yes	Yes	Yes	No	No	Yes
Aquery	Declarative	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes
CQL	Declarative	Yes	Slide, Tumbling	Yes	Yes	Yes	Yes	Yes	Yes
Exareme	Declarative	Yes	Slide, Tumbling	Yes	Yes	Yes	Yes	Yes	Yes
PipelineDB	Declarative	Yes	Slide	No	Yes	Yes	Yes	Yes	Yes
Spark	Imperative/Declarative	Yes	Slide, Tumbling	Yes	Yes	Yes	Yes	Yes	Yes
Flink	Imperative/Declarative	Yes	Slide, Tumbling, Global, Session	Yes	Yes	Yes	Yes	Yes	Yes

We conclude that many different streaming systems and engines with various query languages and optimizations do exist that could be used as a strong base also for streaming triple-based semantic representations. In the following section we give n overview on semantic representations and discuss how relational data bases can be used for querying semantic data.

2.3. Description Logic and Semantic Representation

In the last section we reviewed several data stream management systems mostly based on declarative query languages for accessing tuple streams. The input streams for those systems are often raw data streams, which can lead to difficulties, when facing scenarios with multiple streams and complex queries. Answering high level and complex queries on low level data is a desired goal that pushes research in the field of databases. As an example, imagine a medical system managing sensor input streams for representing a diagnosis. Instead of tempering with raw blood pressure data, the system could be directly queried for a diagnosis on high or low blood pressure or even reason about a medical treatment concerning the diagnosis. Thus, in this case, the medical knowledge would be directly used to represent medical diagnosis based on sensor information.

Knowledge representation and reasoning (**KR**) is part of research in the area of artificial intelligence (**AI**). The idea of **KR** is to represent information about the world in a way that it is readable by computer systems that can help to solve more complex tasks such as medical diagnosis or evaluation of system behavior. Common knowledge systems are split into a *knowledge base*, which includes facts about a specific world and an additional *inference engine* for applying rules to the stored knowledge and answering complex queries on the data.

We describe knowledge of a system by ontologies, originally used in philosophy for describing which things do exist and how they can be grouped and related. Furthermore, in computer science ontologies are a formal description of the vocabulary that is used to talk about a domain, although they are independent from the actual stored data.
Several languages for describing ontologies exist. They generally use three components:

- **Concepts** describe abstract domain objects. Person could be a concept representing humans or in more complex cases the concept of children could be defined by all persons under the age of 18.

- **Individuals** stand for domain objects that exist in the described world and are identified by a name, which could be "Bob" for the class of a male person or "Anne" for a female person.
- **Roles** describe relations between individuals (e.g., between persons: *hasChild* or *isMarriedTo*).

Several large ontologies for different purposes exist. Some famous examples are the friend-of-a-friend ontology FOAF [106], which, describes social networks, and SNOMED-CT [69] for medical systems, or the GENE ontology [52] for biological purposes.

Ontology languages differ in their expressivity. In some cases number restrictions are necessary to form concepts, e.g., for saying that child is a person with an age under 18 that has exactly one mother and one father, but in other ontologies those restrictions might not be required. We can say that ontology languages that support, for example, number restrictions are more expressive compared to other languages that do not support these concepts. But higher expressivity on the other hand, can result in higher computational complexity, which is not desired in cases, in which a low response time of a query system is mandatory. Ontology languages and their expressivity are formally described by different description logics. We will now give an introduction into that particular field.

2.3.1. Description Logics

Description Logics (or DLs) are a family of languages for knowledge representation and reasoning. As DLs are a huge research topic with many different languages, we focus on the most important parts that are needed for accessing data bases efficiently (in our case backend streaming systems). For detailed information and historical backgrounds, we refer to [26].

Our main focus lies on a family of related description logics, the basic description logic DL-Lite and especially its family member DL-Lite$_{\mathcal{R}}$ [62, 63], which is used as a standard for the web ontology language OWL 2 QL (see Section 2.3.1). Many more dialects of the logic family exist, but for a more detailed discussion we refer to [20, 61].

An ontology that is written in a DL language is constructed by different sets of axioms. One set includes all statements that describe concepts (concept descriptions) and is also called terminological part or TBox of the ontology, a second set

includes all statements about individuals also called assertional part or ABox. A third set (only used in more complex logics) is used explicitly to describe relations between roles, such as role hierarchies and complex role conceptions and is simply called role part (or *RBox*).

The description of the ontology is commonly used by a reasoning system for complex reasoning and inferencing tasks. One computational problem solved by reasoning systems is, for example, concept classification, where the idea is to find all concepts C names that subsume a concept D.

Our focus in this thesis is on a framework for efficient access to streaming data sources and specifically to relational streaming sources. The state of the art technique for accessing relational data sources based on ontologies can be found in the DL-Lite framework for Ontology Based Data Access [20] (see Section 2.4).

The framework is set up by a family of DL languages called DL-Lite [62]. We now introduce the basic syntax and semantics of concept descriptions in the language DL-Lite.

A Description Logic Family for Data Base Access: DL-Lite

For DL-Lite the trade-off between expressivity and computational complexity of reasoning is optimized regarding the needs for ontology-based data access [61]. In particular, query answering can be solved directly by query transformation to relational database technologies [66], which is a great benefit of the language.

We define the syntax of the DL-Lite$_{core}$ language with concepts, roles and individuals. Subsequently, we define the semantics of DL-Lite$_{core}$ Finally, we have a further look at three of its family members: DL-Lite$_R$, DL-Lite$_A$ and DL-Lite$_F$.

Syntax of DL-Lite$_{core}$. Let N_C, N_R and N_I be countable, infinite and pairwise disjoint sets of concept names, role names and individual names respectively. Then, we call the triple $\Sigma = (N_C, N_R, N_I)$ a signature.

The set Rols(Σ) is defined as $N_R \cup \{r^- | r \in N_R\}$, where r^- is called the inverse role of r.

The set of general DL-Lite$_{core}$ concepts over Σ is the smallest set, constructed with the following grammar:

$$C ::= \top | \bot | A | \neg A | \exists R | \neg \exists R | C_1 \sqcap C_2 \tag{2.1}$$

where $A \in N_C$ and $R \in Rols(\Sigma)$.

A DL ontology $\mathcal{O} = \langle \mathcal{T}, \mathcal{A} \rangle$ represents a domain that consists of intensional knowledge (a TBox $\mathcal{T}$) and extensional knowledge (an ABox $\mathcal{A}$).

A *TBox* is a set of general concept inclusions (GCIs) of the form:

$$B \sqsubseteq C \tag{2.2}$$

where $B \in N_C$ or $B \in \exists R$ and C is described by an ontology language grammar. The GCI expresses that all instances of concept B are also instances of concept C. For the special case of DL-Lite$_{core}$ ontologies, C is defined by its grammar for DL-Lite$_{core}$ TBoxes given above.

An ABox $\mathcal{A}$ is built by a set of membership assertions on atomic concepts or roles of the form:

$$A(a) | P(a, b) \tag{2.3}$$

defining that a is an instance of concept A and the pair (a,b) is an instance of role P.

Semantics of DL-Lite$_{core}$ The semantics of the language is given in terms of interpretations. An interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ consists of an interpretation function $\cdot^{\mathcal{I}}$ and the interpretation domain $\Delta^{\mathcal{I}}$. It maps every concept B to a subset of the domain $\Delta^{\mathcal{I}}$, every individual a to an element $a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$ and every role name p to a subset of $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$.

In particular the semantic for DL-Lite$_{core}$ is given by [66]:

$$
\begin{array}{rcl}
A^{\mathcal{I}} & \subseteq & \Delta^{\mathcal{I}} \\
P^{\mathcal{I}} & \subseteq & \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \\
(P^{-})^{\mathcal{I}} & = & \{(o_2, o_1) \mid (o_1, o_2) \in P^{\mathcal{I}}\} \\
(\exists R)^{\mathcal{I}} & = & \{o \mid \exists o'.(o, o') \mid\in P^{\mathcal{I}}\} \\
(\neg B)^{\mathcal{I}} & = & \Delta^{\mathcal{I}} \setminus B^{\mathcal{I}} \\
(\neg R)^{\mathcal{I}} & = & \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \setminus R^{\mathcal{I}} \\
(C_1 \sqcap C_2)^{\mathcal{I}} & = & C_1^{\mathcal{I}} \cap C_2^{\mathcal{I}}
\end{array}
$$

We say that $\mathcal{I}$ is a model of a GCI or assertion α, written $\mathcal{I} \models \alpha$, if:

- $\alpha = C_1 \sqsubseteq C_2$ and $C_1^{\mathcal{I}} \subseteq C_2^{\mathcal{I}}$ holds for the general concepts C_1 and C_2, or
- $\alpha = R_1 \sqsubseteq R_2$ and $R_1^{\mathcal{I}} \subseteq R_2^{\mathcal{I}}$ holds for $R \in Roles(\Sigma)$, or
- $\alpha = C(a)$ and $a^{\mathcal{I}} \in C^{\mathcal{I}}$, or
- $\alpha = r(a, b) and (a^{\mathcal{I}}, b^{\mathcal{I}}) \in r^{\mathcal{I}}$ for $r \in Roles(\Sigma)$.

Further, we say that $\mathcal{I}$ satisfies an ontology $\mathcal{O} = \langle \mathcal{T}, \mathcal{A} \rangle$ or $\mathcal{I}$ is a model of $\mathcal{O}$ iff $\mathcal{I}$ satisfies all global concept inclusion axioms in $\mathcal{T}$ and all assertions in $\mathcal{A}$.

We will now show three variants of our base language DL-Lite$_{core}$, followed by examples for each case.

DL-Lite$_{\mathcal{R}}$, also known as DL-Lite$^{\mathcal{H}}_{core}$, is an extension for the ontology language RDFS [232] and is used as a web standard in OWL 2 QL [229], which is a description logic for accessing large amounts of data with respect to ontologies, where the data can be directly evaluated using query transformation strategies on an SQL engine. It extends the inclusion assertions for roles with axioms of the form:

$$R_1 \sqsubseteq R_2 \tag{2.4}$$

where $R_1, R_2 \in Rols(\Sigma)$.

In addition this extension of DL-Lite$_{core}$ allows us two further specifications:

1. *Disjointness*, e.g. between concepts using $A_1 \sqsubseteq \neg A_2$
2. *Mandatory non participation*, e.g. using $A \sqsubseteq \neg\exists P$

The authors of [62] showed that query answering in DL-Lite$_R$ is still of the same complexity as DL-Lite$_{core}$.

DL-Lite$_\mathcal{F}$, also known as DL-Lite$^\mathcal{F}_{core}$, extends DL-Lite$_{core}$ with additional TBox axioms as a specification of functionality roles, which are noted in the form:

$$\text{(funct R)} \tag{2.5}$$

Following with its semantics noted as follows:

$$\text{(funct R)}^\mathcal{I} is satisfied if \{(o, o_1) \mid (o, o_2) \in R^\mathcal{I} \implies o_1 = o_2\} \tag{2.6}$$

DL-Lite$_\mathcal{A}$ is also a well-known extension in the family of DL-Lite. It was first introduced in [192] and implemented in the query engine QuOnto[5]. The explicit "$\mathcal{A}$" stands for *attribute* and stems from the distinction of DL-Lite$_\mathcal{A}$ between abstract objects and data.

In DL-Lite$_\mathcal{A}$ we can distinguish between objects and data values, between concepts and data types, and between roles and attributes, which is also adopted in the OWL language. Nevertheless, with respect to [66] we see no change in the complexity of reasoning since datatypes can be seen as special concepts that are a disjoint from the set of real concepts.

Instead, it is shown in [62] that DL-Lite$_\mathcal{A}$ includes DL-Lite$_\mathcal{F}$ as well as DL-Lite$_\mathcal{R}$. However, the query answering complexity results are lost, when combining role inclusions of DL-Lite$_\mathcal{R}$ and functionality of DL-Lite$_\mathcal{F}$ in an unrestricted way. Therefore, for keeping the desired complexity for query answering and satisfiability, functional roles are restricted to be non negated on the right hand side of a concept inclusion in DL-Lite$_\mathcal{A}$.

We close this section by giving a concrete example from the sensor measurement scenario in DL-Lite.

Let a TBox $\mathcal{T}$ be given in DL-Lite$_{core}$ as follows:

$$\begin{aligned} Assembly &\sqsubseteq \exists hasSensor \\ Sensor &\sqsubseteq \exists locatedAt \\ \exists hasSensor^- &\sqsubseteq Sensor \\ Assembly &\sqsubseteq \neg Sensor \end{aligned}$$

Derived from the TBox, we can say that assemblies have sensors, sensors have a location, assemblies are no sensors and sensors are no assemblies.
By using DL-Lite$_R$, we can express the following additional axiom:

$$hasSensor^- \sqsubseteq hasLocation$$

stating that if an assembly has a sensor, it is directly located at the assembly.

Another assertion can be added in DL-Lite$_\mathcal{F}$:

(funct hasLocation)

Saying that any object can only have one location.

After giving an example for a TBox, we finish the example by adding a small ABox $\mathcal{A}$ with respect to $\mathcal{T}$.

$$\begin{aligned} \text{Assembly}(Turbine1), \\ \text{Sensor}(Sensor1), \\ \text{hasSensor}(Turbine1,\ Sensor1) \end{aligned}$$

Considering the TBox, for our example we can directly derive that Sensor $Sensor1$ is located at $Turbine1$.

More Expressive DLs

We have discussed description logics from the family of DL-Lite, which have well designed properties for data base access, while sacrificing some necessary expressive power. But different users have different requirements on expressivity, for example in some cases expressivity is more important than computational efficiency and scalability. For these cases more expressive description logics have been developed. In the following we show some of its most important representatives, starting with the family of attributive languages.

The base set for many expressive description languages is seen in the attributive language or short $\mathcal{AL}$. The syntax of its extension $\mathcal{ALC}$ already goes beyond the syntax of DL-Lite and is shown in the following:

$$C ::= \top | \bot | A | \neg C | C_1 \sqcap C_2 | C_1 \sqcup C_2 | \exists R.C | \forall R.C \tag{2.7}$$

As we have seen for DL-Lite, each description language has its own naming scheme and acronym with respect to its expressivity. The added $\mathcal{C}$ stands for "complement" and extends the atomic negation in $\mathcal{AL}$ by complex concept negations. Additionally, $\mathcal{ALC}$ includes concept unions (usually symbolized by an $\mathcal{U}$) and full existential quantification (usually symbolized by an $\mathcal{E}$).

However, compared to the former discussed logics of DL-Lite, we have more complex role restrictions (e.g., universal value restrictions), which results in a higher complexity. It was shown in [224] that the satisfiability problem of $\mathcal{ALC}$ is in PSPACE.

We can further extend the expressivity of $\mathcal{ALC}$ in the following steps, starting with $\mathcal{ALCHI}_{R+}$. In $\mathcal{ALCHI}_{R+}$ we add more expressivity to roles, i.e., role hierarchies, inverse roles and transitivity. This language is commonly shortened by the acronym $\mathcal{SHI}$ [122]. From there we can extend our logic with cardinality restrictions to $\mathcal{SHIN}$ or $\mathcal{SHIQ}$ [123]. Extending the expressivity with nominals (i.e., individual names) and *oneOf*-constructors in the TBox results in a logic that is named by $\mathcal{SHOIN}$ or $\mathcal{SHOIQ}$ [124]. Finally, role expressivity can be further extended with role reflexivity or irreflexivity and role disjointness, included in the DL $\mathcal{SROIQ}$. The complexity of these logics has been analyzed in [224]. Its authors show that the satisfiability problem of $\mathcal{SHIQ}$ is in EXPTIME, while the complexity for $\mathcal{SROIQ}$ is in 2NEXPTIME [129].

Web Ontology Language

The *Web Ontology Language* or short OWL [228] is a web standard for describing ontologies from the W3C Web Ontology Working Group. As an extension of the Resource Description Framework (short RDF) [166] data formulated in OWL is encoded in RDF/XML documents [40], where its constructors are parsed into a RDF triple schema. The idea is to store the OWL ontology in semantic web files for being exchanged as RDF documents over the web.

In the last section we have seen that computing conclusions of an ontology can become a challenging task that depends on the description logic and expressivity that is used. An update for the standard OWL was given by the W3C consortium with OWL 2 to address this problem for standard use cases [149]. The consortium introduced three new sublanguages also called *profiles*: OWL 2 EL, OWL 2 RL and OWL 2 QL. While OWL EL is used in context with huge biomedical ontologies and OWL RL for reasoning on web data, OWL 2 QL is the preferred choice for database application with ontology based data access.

OWL 2 QL aims for applications using very large volumes of instance data with a specific need for query answering, while based on a description logic we have already introduced: DL-Lite$_{\mathcal{R}}$ (or respectively DL-Lite$^{\mathcal{H}}_{core}$). Nevertheless, OWL 2 QL shows an important difference compared to DL Lite regarding the unique name assumption, which is generally adopted by DL-Lite, but not in OWL 2 QL. Instead, the OWL 2 QL language provides specific constructors (i.e., *sameAs* and *differentFrom*) stating that two object names are denoting the same or different individual [63].

2.3.2. Ontologies for Sensor Networks

Sensor networks are an area where semantic technologies can overcome the hurdles and complexity of heterogeneous standards. These semantic descriptions can be viewed as OWL ontologies (as we explained above). Many recently proposed semantic models for describing sensor architectures are based on the W3C OWL recommendation [13] or the OWL 2 standard profiles [149]. For the case of sensor networks, the OGC[10] provides a Sensor Web Enablement suite of standards as a syntactical model [53]. While specifically designed for sensors, it includes a general model and XML encodings for observation and measurements (O&M) [195], a data model for exchanging sensor related data and a sensor modeling language (SensorML) [54] for describing sensor processing systems as well as several interfaces and data models for sensor based web services.

Many earlier sensor models simply use meta and static data to represent sensors, while the modeling of time-based observations was not present, these models rely on the SensorML language, but not on the O&M model. Those kinds of ontologies include for example: the OntoSensor ontology [207], the Suggested Upper Merged Ontology (SUMO) [139], the SWAMO ontology [242] and the CSIRO sensor ontology [78, 176]. For more detailed information on these or other related ontologies we refer to [77].

Based on the results above, where sensor observations have rarely been modeled in earlier ontologies, the W3C Semantic Sensor Network Incubator group [237] proposed an OWL 2 ontology for modeling sensor meta data. It describes the act of sensing and sensor observations all together, while still compatible to the OGC standard models for sensors (SensorML) and observations (O&M). On the other hand, non sensor specific models were not included into the ontology, such as measuring units, spatial information or hierarchies of types and physical elements. The idea was to give engineers the possibility to include the model directly into their system,

[10]Open Geospatial Consortium

where such data already exists or could be included from external ontology for an explicit use case. Therefore, the ontology only offers place holders for such concepts.

The SSN (Semantic Sensor Network) ontology [76] is organized by ten different modules (see Figure 2.7). Its heart is the *Stimulus-Sensor-Observation*(SSO) pat-

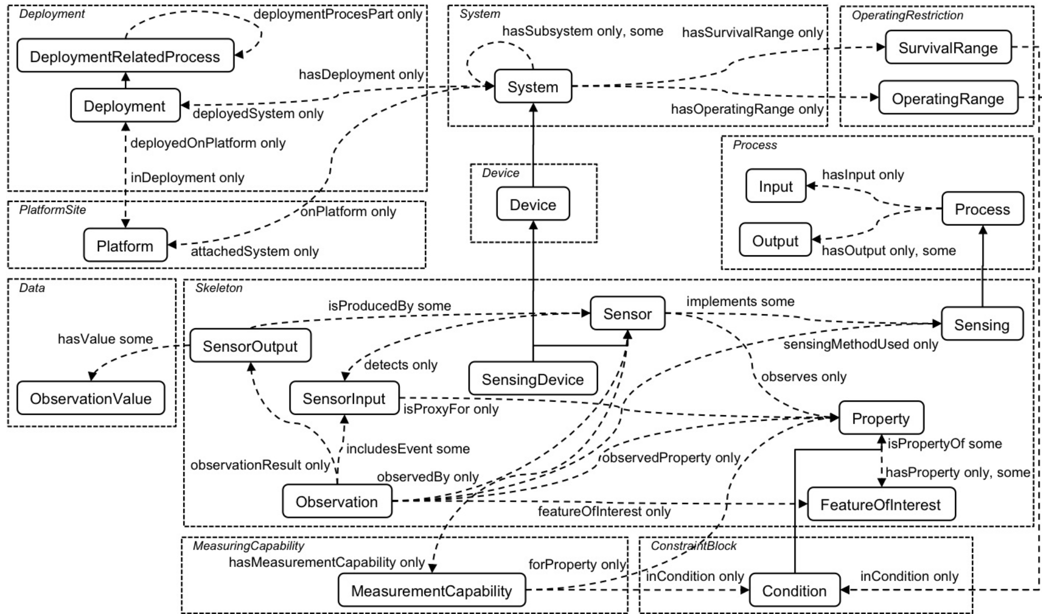

Figure 2.7.: Structure of the SSN ontology [76]

tern module [126], which is focused on the relations between sensors, stimulus and observations and offers constructors for each element. Stimuli (ssn:stimulus) are detectable changes recognized by a sensor (ssn:sensor) for measuring a property used as a proxy for sensors (denoted as *SensorInput* in Figure 2.7), which could be for example the electrical resistance or current for changing temperature or speed. Sensors transform the stimuli into a more abstract new representation (ssn:sensorOutput). The central point of the SSO pattern are observations (ssn:observations). They can be seen as the link between sensor, stimulus and observed property, while setting the observations into a context of time and spatial information.

Additional modules are the *Data* module for storing the sensor observations or the *Deployment* and *System* modules, which describe the topological structure between sensors and devices.

Examples for the SSN ontology are use cases in the SPITFIRE FP7 project [187] and work at 52° North [1], an initiative for geospatial data representation. For more

detailed information on the SSN we refer to [76].

2.3.3. The SPARQL Query Language

In the section above we described representations of data in the case of sensor networks with respect to ontologies. We presented the Web Ontology Language standard of the W3C community for describing ontology based data and gave an example for sensor observations formulated in OWL. The missing piece for data retrieval is an expressive query language, tailored for data represented in the described format.
We will now introduce SPARQL [234] [185], the standard query language of the W3C for querying RDF data [141]. We define briefly the abstract syntax of the RDF data sets queried by SPARQL and proceed with covering the main operators of the SPARQL query language. We conclude with an SPARQL example from the sensor measurement scenario.

RDF Data Format

The Resource Description Framework (RDF) is a W3C recommendation for data representation in the web. Its abstract syntax is based on two important data structures: RDF graphs and RDF data sets. For the following formalization, we closely follow the notation introduced in [141].

We start by defining three disjoint sets, namely the set of IRIs $\mathbb{I}$, blanknodes $\mathbb{B}$, and literals $\mathbb{L}$, which build the ground for RDF data descriptions and, according to [141], can be understood as follows.

- The set $\mathbb{I}$ includes **IRIs** (Internationalized Resource Identifier), which are extensions of Uniform Resource identifiers by allowing also characters of the unicode character set.

  They can be seen as a string for identifiying resources over a network and are constructed by a prefix namespace (given by an URL) and an Uniform Resource Name (URN). An example IRI is *http://www.sensor.net#Sensor1*. The namespace URLs are often shortend by a defined equivalent prefix, which could be written as *sn*, forming sn:Sensor1 as a shortcut for the IRI.

- The set $\mathbb{B}$ of **Blank nodes** is disjoint from IRIs and Literals, they do not have any specific RDF identifier, but can be seen as placeholders for anonymous resources with local scope. Blank nodes are often used to create anonymous

groups of RDF data and can be distinguished from IRIs by using the character "_" as a namespace (e.g. in _:SensorNet).

- The set $\mathbb{L}$ of **literals** is used for values such as strings, numbers, or dates. A literal typically consists of two parts: a lexical form, being a unicode string and an IRI identifying the data type, such as "1"^^xsd:integer. Datatypes are often omitted, as concrete syntaxes support *simple literals*, which consist only of the lexical form.

The RDF format is designed to describe data by triples, which consist of a subject, a predicate and an object, where the object o of a specific property p is a description for subject s. As mentioned before, the value of o could be either a new resource or a literal, e.g. when specifying a sensor, a string naming its specific type. We can formalize RDF triples as the following.

Definition 2. ***RDF Triple.*** *Let $\mathbb{I}$ be a set of IRIs, $\mathbb{B}$ be a set of blanknodes and let $\mathbb{L}$ be a set of literals, such that $\mathbb{I} \cap \mathbb{B} \cap \mathbb{L} = \emptyset$. Then an RDF triple is defined as an element $(s, p, o) \in \mathbb{I} \cup \mathbb{B} \times \mathbb{I} \times \mathbb{I} \cup \mathbb{B} \cup \mathbb{L}$.*

A set of RDF triples builds a RDF graph. Graphs consist of nodes and edges, while subjects and objects form nodes in the graph, predicates make up edges respectively. Each described resource (except literals) can be a subject or object node and as such also be described by more than one triple in the graph.

Definition 3. ***RDF Graph.*** *An RDF Graph $\mathbb{G}$ is defined as a set of $n \geq 1$ RDF triples t_i: $\mathbb{G} = \{t1, t2, \ldots, tn\}$.*

Furthermore, we can group RDF triples in graph collections and associate each graph with an IRI, which is also called graph name. Multiple graphs are closely aligned with SPARQL. An RDF dataset may consist of multiple named graphs and one unnamed graph, also called the $defaultgraph$. Following that idea, we informally define RDF datasets.

Definition 4. ***RDF dataset.*** *An RDF dataset is a set of RDF graphs, including an unnamed default graph and a number of optional named graphs.*

The W3C standard for querying RDF data sets is the SPARQL query language as described in [185].

SPARQL Description

We give a short introduction into the formal syntax of SPARQL and follow the notations given in [185] and [10].

A basic SPARQL query is syntactically formed by a query form (e.g. SELECT, CONSTRUCT), a WHERE clause and optional solution modifiers (e.g. DISTINCT, ORDER BY).

We define a *basic graph pattern* as follows.

Definition 5. ***Basic Graph Pattern*** *Let* $\mathbb{V}$ *be an infinite set of variables,* $\mathbb{I}$ *be a set of IRIs, and let* $\mathbb{L}$ *be a set of literals, such that* $\mathbb{V} \cap \mathbb{I} \cap \mathbb{L} = \emptyset$*. and an element of* $(\mathbb{V} \cup \mathbb{I}) \times (\mathbb{V} \cup \mathbb{I}) \times (\mathbb{V} \cup \mathbb{I} \cup \mathbb{L})$ *is called triple pattern, then a finite set of triple patterns is defined as basic graph pattern.*

Basic graph patterns are the basic constructs for graph pattern expressions, which build the WHERE clause. They can be combined recursively as follows:

1. If P_1, P_2 are graph patterns, then (P_1 UNION P_2), (P_1 AND P_2), and (P_1 OPT P_2) are graph patterns
2. If P is a graph pattern and R is a filter condition, then (P FILTER R) is a graph pattern.

A SPARQL filter condition can restrict specific variables in the where clause. It connects constants or elements from $\mathbb{I}$, $\mathbb{V}$ or $\mathbb{L}$ by inequality symbols ($<$, $\leq$, $\geq$, $>$), ($=$), logical operators ($\neg, \vee, \wedge$) and specific unary predicates such as `BOUND` or `IS BLANK` (a complete list can be found in [185]).

For simplicity, we restrict the definition of filter conditions to the connective equality symbol ($=$) and the unary predicate `BOUND()`. Filter conditions are defined for other mentioned operators accordingly.

Definition 6. ***Filter Condition*** *Let* $?X, ?Y \in \mathbb{V}$ *and* $c \in \mathbb{I} \cap \mathbb{L}$*, then filter conditions are defined as follows:*

1. $?X = c$, $?X = ?Y$ *and bound(?X) are atomic filter conditions.*
2. *If* C_2 *and* C_2 *are filter conditions then* $(\neg C_1)$*,* $(C_1 \vee C_2)$*, and* $(C_1 \wedge C_2)$ *are complex filter conditions.*

SPARQL Example

The graph pattern expression of the WHERE clause is matched against the RDF dataset binding the variables in the pattern for query evaluation.

We show a SPARQL query example using a graph pattern and a filter condition in its `WHERE` clause in Listing 2.5

Listing 2.5: A query formulated in SPARQL with filter constraints

```
1  PREFIX sn : <http://www.sensor.net/>
2
3  SELECT ?sens
4  WHERE {
5     {  ?sens sn:hasLocation ?loc .
6        ?loc sn:installedSensors ?num.
7        FILTER (?num > 5)}
8     UNION { ?sens sn:hasLocation "Turbine1" }
9  }
```

The query selects all sensors, which are located at *Turbine1* or are found at some location, where a minimum number of six sensors is installed. Its `WHERE` clause consists of two graph patterns connected by an `UNION` operator. While the first pattern uses two triple patterns and a filter condition for restricting the number of locations, the second one adds all sensors to the output, which are located at *Turbine1*.

Query Forms. The SPARQL syntax provides several different query forms. In Listing 2.5 we used a `SELECT` query for retrieving simple variable binding lists. The four possible forms are listed below.

- **SELECT** queries provide all possible variable bindings in a list, they are the most typical used query form.
- **ASK** queries are boolean queries, they return *true*, if an answer set to the query exists and *false* otherwise.
- **CONSTRUCT** queries return a new RDF Graph concerning the defined graph pattern in the query head or `CONSTRUCT` clause respectively
- A **DESCRIBE** query form returns information about the variable bindings from the query head in a single result RDF graph. The exact returned dataset is not defined in the SPARQL query, but directly selected by the SPARQL query processor.

The output of the SPARQL query can also be restricted by a so called **optional solution modifier**. We give a list of informal descriptions for the most important modifiers below.

- The **DISTINCT** modifier prevents duplicate entries from the result binding list.
- The **ORDER BY** clause is used for sorting answer sets with respect to specific variables in an ascending or descending order (using the keyword `ASC` or `DESC` respectively).
- The **LIMIT** modifier can be used to limit the returned output to a fixed number of bindings. It is typically used in combination with the `ORDER BY` modifier (e.g. for selecting the five highest/lowest values in a result set).

For more details on SPARQL operators and extended examples regarding solution modifiers, we refer the reader to [185].

SPARQL 1.1. Since 2013 an extended version of SPARQL, called SPARQL 1.1 [115], is recommended by the W3C [235] and includes several new operators and features. To give an idea of the extended features, we list a short informal description below. Detailed information about the new specifications can be found in [115].

- The extended SPARQL query specification introduces **new language operators**.
  - Inspired by other query languages such as SQL, **aggregation** operators were added to the feature list of SPARQL 1.1. The aggregation operators are computations over groups of solutions by applying different aggregation functions to variables, such as `COUNT`, `SUM`, `MIN`, `MAX`, `AVG` or others.
    Aggregations are often used together with a `GROUP BY`, mentioning a list of variables for grouped computations, or a `HAVING` clause, for arithmetic expressions on aggregation results (e.g. "HAVING AVG(?num) > 5" restricts the result set of ?*num* to an average higher than five).
  - **Negation** is added to the feature list using `FILTER NOT EXISTS` or connecting two graph patterns by an `MINUS` operator.
  - **Subqueries** can be used as a replacement of a basic graph pattern.

  - The new **property paths** are used to define possible routes through a graph between two nodes. For example, in a scenario that declares social network graphs of who knows who, we could directly ask for persons who know another person over two or more edges, by defining a subgraph schema of the RDF dataset in the query.

- New **UPDATE** statements allow for updating the RDF dataset.
- One can define **query results** in specific formats such as XML, JSON or CSV.
- Another important new feature is **federation**, which combines different RDF data sets from different sources that can be identified by an IRI for to be queried in a single query.

In this section we showed how ontology based data can be accessed through the web, while using the SPARQL query language. The next section uses the ontology based data view to access relational data by the use of a query transformation technique called ontology based data access or short OBDA.

2.4. Ontology Based Data Access

Ontology based data access (for short OBDA) describes a paradigm for accessing relational backend data sets through a query language based on concepts and models defined in an ontology. A common goal behind this approach is easy access to large volume and heterogeneous data sets by lifting the data to an abstract level with an ontology based query language.

OBDA can be seen from two perspectives. The classical approach of OBDA (also named virtualized approach) is a setting where the huge dataset is kept in external data stores. An ontology based query is transformed into the query language used by the backend data base. Therefore, we say that the accessed ABox assertions only exist *virtually*. A possible advantage of this approach is that one can rely on already available optimizations and index strategies for relational sources.

As a result of the transformation process, the compiled query is possibly more complex and potentially bigger in size, and thus several optimization strategies for rewriting these queries considering efficient query answering have been developed (e.g., in [202]).

Following the rewriting approach, it is also possible to integrate different heterogeneous sources under a single interface, while rewriting queries automatically to each

backend without noticing it from the front end perspective. Even if it is necessary to join different tables from different sources for answering a transformed query, these joins could be provided by an additional service layer. Furthermore, the model of a TBox and ABox allows a clear distinction between intensional knowledge and the assertional facts that are stored in the database.

A second approach of OBDA exists that does not use query transformation techniques, but transforms relational backend data into triple based representations and stores it in an additional triplestore. This approach is also called *materialization approach*. Although the ontology based queries can be evaluated in a real triple store environment, additional space for storing the data in triples is necessary.

Although this approach might lack the optimizations of the relational backend, it might possibly be directly optimized on the triple store (e.g., with index structures or other techniques) for the specific use case. Anyway, this approach seems to be impractical considering that data has to be updated periodically for example and thus, the materialized data has to be reset each time the source is updated.

Additional problems can be seen with respect to really large data sets, where the materialization step takes a lot of time. For these reasons, the virtual approach with query transformation techniques is the most preferred for big data use cases.

OBDA has become an interesting topic for closing the gap between description logics and database research, based on lightweight ontologies such as DL-Lite (see Section 2.3.1). But also strategies for accessing huge data sets with more expressive ontologies have been proposed [167]. Recent work [23, 25, 58] shows that not only static, but also temporal and even streaming data can be accessed and processed by an appropriate ontology based query language and interface. In this case, we see that related industrial projects and use cases currently arise, two examples are the Statoil and Siemens use cases in the FP7 OPTIQUE[11] Project[12].

The following section gives an overview on basic OBDA technologies regarding recent temporal and streamified applications that use a query transformation approach, more details and references are described in [179].

[11] http://www.optique-project.eu
[12] http://www.optique-project.eu

2.4.1. Classical OBDA

We start by discussing theoretical issues of the classical OBDA approach. For a detailed description, we refer to [61]. The idea of the approach is a reduction of the complex inference problem with respect to ontologies to a query answering problem on relational data sources. As in this scenario the TBox is relatively small in size compared to the large ABox, we are able to restrict the computational complexity to *data complexity*, which only takes assertional facts of the ABox into account. Furthermore, answering queries with respect to a DL-Lite (see Section 2.3.1) ontology can be reduced to the problem of query evaluation for First-Order Logic (i.e., SQL) over relational databases [61], which is known to be in the complexity class of AC^0 [4] and can be described as the class of problems which can be solved in polynomial or constant time, while adding a polynomial number of processors.

Query Answering

We consider in the following query answering w.r.t. unions of conjunctive queries (UCQs), i.e; a subclass of FOL queries.

Answering FOL queries w.r.t.ontologies, while the assertional facts of the ABox are stored in a database, requires rewriting of the TBox into another FOL query.

A FOL *query* $q = \psi(\vec{x})$ is a FOL formula $\psi(\vec{x})$ with free and (pairwise) distinct variables in vector $\vec{x}$. The arity of $\vec{x}$ is the arity of Q. If the vector $\vec{x}$ is empty, Q is called a boolean query.

A *conjunctive query* (CQ) over a DL-Lite ontology $\mathcal{O}$ is a FOL query $q(\vec{x})$ in which $\psi(\vec{x})$ is of the form

$$\exists \vec{y}.conj(\vec{x}, \vec{y}).$$

Where $conj(\cdot)$ is a conjunction of atomic formulas over variables from $\vec{x}$ or $\vec{y}$. Mentioning this, *unions of conjunctive queries* (UCQs) are simply unions of conjunctive queries [34].

First Order Logic Rewritability In the case of rewriting an FOL query, we consider query answering over databases and therefore, we have to look for answer sets that do not depend on the interpretation of unknown data.

Therefore, we define the certain answers to a UCQ w.r.t.ontologies according to [145] by considering its signature (see Section 2.3.1).

Definition 7. ***Certain Answers.*** *Let O be an ontology $\mathcal{O} = (Sig, \mathcal{T}, \mathcal{A})$ and $\psi(\vec{x})$ be a UCQ with respect to O, then the set of certain answers $cert(\psi(\vec{x}), \mathcal{O})$ is defined by n-ary tuples of constants $\vec{a} \in I_N$ from $Sig = (C_N, R_N, I_N)$, where a substitution $\psi_{[\vec{x} \to \vec{a}]}$ is entailed by O, such that:*

$$cert(\psi(\vec{x}), \mathcal{O}) = \{\vec{a} \mid O \models \psi_{[\vec{x} \to \vec{a}]}\}$$

After having defined certain answers on the ontology side, we would like to define an equal canonical model for queries on the classical relational database side with respect to the transformation of ontology based queries.

We denote by $DB(\mathcal{A})$ the representation of $\mathcal{A}$ that can be stored in a relational database (DB).

Additionally, we define that $a^{DB(\mathcal{A})} \in A^{DB(\mathcal{A})}$ iff $A(a) \in \mathcal{A}$ and $(a^{DB(\mathcal{A})}, b^{DB(\mathcal{A})}) \in R^{DB(\mathcal{A})}$ iff $R(a, b) \in \mathcal{A}$.

Using the described canonical model $DB(\mathcal{A})$, we define FOL-rewritability according to [61].

Definition 8. ***FOL rewritability.*** *Answering UCQs in a DL $\mathcal{L}$ is FOL-rewritable, if for every TBox $\mathcal{T}$ over $\mathcal{L}$ and every UCQ q, there is an FOL query $\mathcal{Q}_{\mathcal{T}}$ such that for all ABoxes $\mathcal{A}$ it is the case that:*

$$cert(q, O) = Q_{\mathcal{T}}^{DB(\mathcal{A})}$$

The rewriting of FOL queries guarantees a complexity for query answering in AC^0 data complexity. It can be shown that if a DL $\mathcal{L}$ is not in data complexity and has a higher complexity, it is also not rewritable in first order logic [61].

2.4.2. Query Transformation for Access on Static Data

After having defined FOL rewritability, we will now explain how a standard transformation algorithm can convert static ontology-based FOL queries into queries in relational algebra.

The classical (i.e. virtual) OBDA approach without querying time or time sequences has been investigated for many years and is commonly applied by transforming the query language SPARQL (Section 2.3.3.

For the ontology based transformation of SPARQL a number of tools and systems that at least partially implement the OBDA paradigm have been developed so far, including D2RQ [45], Mastro [192], morph-RDB [194], Ontop [60], OntoQF [173], Virtuoso [91] and Ultrawrap [209].

Such systems proved to be successful in a number of areas, including culture heritage [67], governmental organizations [75] and industrial applications [132, 136].

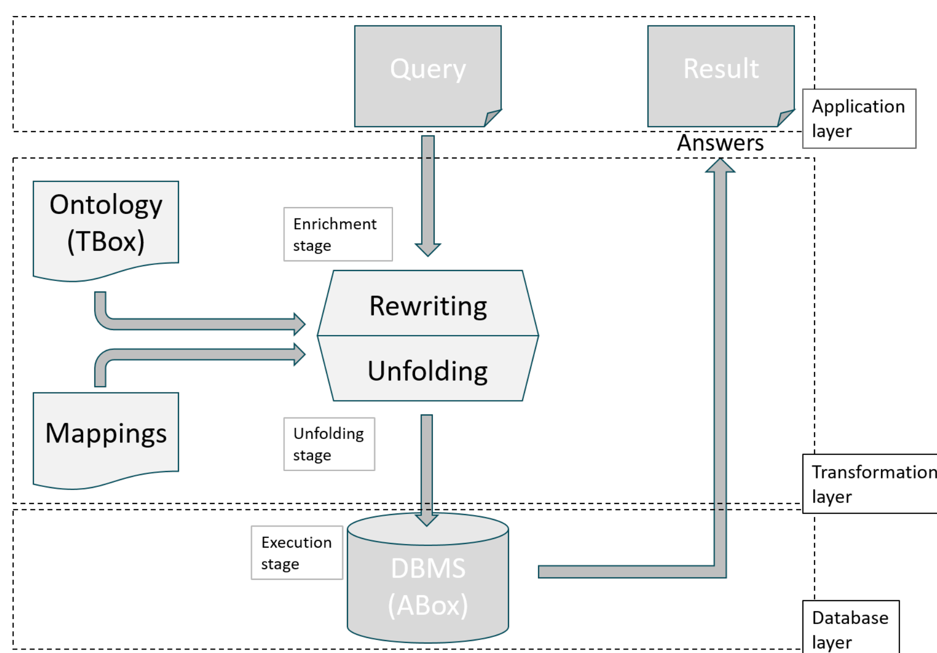

Figure 2.8.: Schematic OBDA process for static data

An explicit FOL rewriting algorithm is, e.g., realized in [61] by the so called *perfect rewriting*. It rewrites a FOL query in such a way that the axioms of the TBox are directly included into the query by using a backward-chaining method, which can possibly increase the query exponentially by its size, considering the used TBox axioms.

An alternative to the perfect rewriting approach is the abstract computation model of combined rewriting: The given query is rewritten w.r.t. the TBox and then posed to a pre-processed ABox resulting from the original ABox by (partially) materializing TBox axioms and adding them to the ABox. (See [159, 160] for the original definition used in the context of the description logic $\mathcal{EL}$ and [144] for an application w.r.t. DL-Lite). This approach is used for specific engineering

scenarios, e.g., in the transformer *kyrie* (see [169]), which provides an expressivity of $\mathcal{ELHIO}$.

Common OBDA tools such as the ones mentioned above use three steps for the query evaluation process (see Figure 2.8): *(i)* in the *rewriting* stage ontology axioms are used to expand the ontological query in order to access the complete possible answer set with respect to a TBox $\mathcal{T}$; *(ii)* in the *unfolding* stage the mappings are used to translate the enriched ontological query into (possibly many unions of) queries over the data; and *(iii)* in the *execution* stage the unfolded data queries are executed over the data.

Query Enrichment

To compute the certain answer set to a UCQ $q(x)$ over an ontology $\mathcal{O} = \mathcal{T}, \mathcal{A}$, where the ABox $\mathcal{A}$ is considered as a relational database $DB(\mathcal{A})$, we apply a query rewriting algorithm that is able to include all necessary axioms of $\mathcal{T}$ into $\psi(\vec{x})$.

Several rewriting algorithms and optimizations exist in the literature. While some of them use the perfect rewriting approach (e.gṖerfectRef [61]), others employ optimizations for datalog programs such as Presto [206] or Prexto [205]. For a survey and comparison of different algorithms we refer the reader to [108] or [170].

For this work we decided to use a query enrichment and unfolding tool that is commonly used, regularly updated and optimized, namely Ontop [13] [60]. It implements the perfect reformulation algorithm from [61], which is described in details below.

Perfect Reformulation Algorithm. The PerfectRef Algorithm is designed to enrich UCQs for answering them on relational databases w.r.t. an DL-Lite ontology. Before giving the concrete algorithm we make some preliminary definitions according to [61].

We call an argument of a query atom *bound*, if it is either a variable occurring twice in the query body or a constant. We call all other arguments *unbound* and denote them in the following by the symbol '_'.

The general algorithm (shown as Algorithm 1 below) transforms a UCQ $q(x)$ into enriched UCQs based on a TBox $\mathcal{T}$. It is divided into two steps, which are executed in a loop over the set of atoms in q until these atoms no longer change. The first step is then executed by two inner loops on the atoms g in q and the positive inclusion axioms (PI for short) α, given in $\mathcal{T}$. It identifies those α, which are applicable to

[13] http://ontop.inf.unibz.it/

one of the atoms g of the given UCQ and adds the result defined by a function $gr(g, \alpha)$ into the input set of the CQs until no further PIs are applicable.

Algorithm 1: `PerfectRef` algorithm computing the perfect reformulation of a CQ w.r.t. $\mathcal{T}$ [61]

```
input  : UCQ q(x), DL-Lite_A TBox T
output: UCQ pr

pr:=q;
repeat
    pr':=pr;
    foreach CQ q' ∈ pr' do
        foreach Atom g in q' do                          /* Step 1 */
            foreach PI α in T do
                if α is applicable to g then
                    pr:= pr ⋃ q'[g/gr(g,α)];
                end
            end
        end
        foreach pair of atoms g1, g2 in q' do            /* Step 2 */
            if g1 and g2 unify then
                pr:=pr ⋃ anon(reduce(q',g1,g2));
            end
        end
    end
until pr'=pr;
return pr
```

Two important aspects of the algorithm are shown by the authors of [61]. First, they proved that it is only necessary to consider positive inclusions, as negative inclusions (with a negation on its right-hand side) do not have effects on rewritings and only have to be considered for satisfiability problems w.r.t. the ontology. Moreover, the algorithm always terminates, while the generated number of distinct atoms is polynomial with respect to the size of the output query.

The cases, where a positive inclusion axiom α can be applied to an atom g, are shown in Table 2.3 together with its application result in column $gr(g, \alpha)$. More formally, they can be defined by the following rules:

1. A PI α is applicable to an atom $A(x)$, if α has A in its right-hand side.
2. A PI α is applicable to an atom $P(x, y)$, if one of the following conditions holds:

Table 2.3.: Results of application function $gr(g, \alpha)$ for applying a PI α to an UCQ atom g [61]

Atom g	**Positive inclusion** α	$\mathbf{gr}(g, \alpha)$
$A(x)$	$A' \sqsubseteq A$	$A'(x)$
$A(x)$	$\exists P \sqsubseteq A$	$P(x, _)$
$A(x)$	$\exists P^- \sqsubseteq A$	$P(_, y)$
$P(x, _)$	$A \sqsubseteq \exists P$	$A(x)$
$P(x, _)$	$\exists P' \sqsubseteq \exists P$	$P'(x, _)$
$P(x, _)$	$\exists P'^- \sqsubseteq \exists P$	$P'(_, y)$
$P(_, y)$	$A \sqsubseteq \exists P^-$	$A(x)$
$P(_, y)$	$\exists P' \sqsubseteq \exists P^-$	$P'(x, _)$
$P(_, y)$	$\exists P'^- \sqsubseteq \exists P^-$	$P'(_, y)$
$P(x, y)$	$P' \sqsubseteq P$ or $P'^- \sqsubseteq P^-$	$P'(x, y)$
$P(x, y)$	$P' \sqsubseteq P^-$ or $P'^- \sqsubseteq P$	$P'(x, y)$

a) $y = _$ and the right-hand side of α is $\exists P$;or
b) $x = _$ and the right-hand side of α is $\exists P^-$;or
c) α is a role inclusion assertion and its right-hand side is either P or P^-.

After an extension by the additional inclusion axioms, as defined under step one of the algorithm, the second step reduces the resulting union of conjunctive queries by using two functions in a loop on every distinct pair g_1 and g_2 in q'. The function $reduce(q', g_1, g_2)$ unifies the two atoms, if possible and returns a new resulting query q". Afterwards the function $anon(q'')$ realizes a variable anonymization by substituting the unbound variables of q'' with a '_' symbol (representing non-shared variables as explained above). One further benefit of using the reduce function is that variables, which are bound in q', can become unbound in q'' and therefore, become applicable to a positive inclusion axiom in the following iteration of the algorithm.

Example 1 (A CQ q and DL-LITE$_{\mathcal{A}}$ TBox $\mathcal{T}$ in sensor measurement scenario)**.**

$$\begin{array}{rl} Assembly \sqsubseteq \neg Sensor & \exists mountedAt^- \sqsubseteq Assembly \\ Assembly \sqsubseteq \exists hasSensor & \exists hasSensor^- \sqsubseteq Sensor \\ Sensor \sqsubseteq \exists mountedAt & (funct\ mountedAt) \end{array}$$

$$CQ\text{: } q(x) \leftarrow \ hasSensor(x,y),\ mountedAt(y,_)$$

We illustrate the *PerfectRef* Algorithm by a practical example adopted from the sensor measurement scenario. Let be given the TBox $\mathcal{T}$ from Example 1. Here, we make use of the atomic concepts *Sensor* and *Assembly*, and of the roles *hasSensor* and *mountedAt*. The TBox further states that no assembly is a sensor (and vice versa), while assemblies have sensors, which are mounted to something that is an assembly and further that each sensor is mounted at most to one assembly.

Moreover, we consider the conjunctive query CQ over $\mathcal{T}$ from Example 1, asking for assemblies that have sensors, which are mounted somewhere. At the first execution of step 1 from the *PerfectRef* Algorithm the PI *Sensor* $\sqsubseteq$ ∃mountedAt can be applied to the Atom mountedAt(y,_), which inserts to *pr* the new query:

$$q(x) \leftarrow \text{ hasSensor}(x, y), Sensor(y).$$

At the second iteration of step 1, the positive inclusion axiom $\exists\text{hasSensor}^- \sqsubseteq$ *Sensor* is applied to *Sensor*(y) and inserts to *pr*:

$$q(x) \leftarrow \text{ hasSensor}(x, y),\ \text{hasSensor(_,y)}.$$

As the two atoms can be unified by the reduce function, the following atom can be added to *pr* by the algorithm in step 2

$$q(x) \leftarrow \text{ hasSensor(x,_)}.$$

The variable y is unbound in the new query (appears only once) and can therefore be replaced by the symbol '_'. At the next iteration we execute once again step 1 and apply *Assembly* $\sqsubseteq$ ∃hasSensor to hasSensor(x,_), resulting in:

$$q(x) \leftarrow Assembly(x).$$

A final execution of step 1 applies $\exists\text{mountedAt}^- \sqsubseteq$ *Assembly* to *Assembly*(x), which adds to *pr* the query:

$$q(x) \leftarrow \text{mountedAt(_,x)}.$$

The algorithm PerfectRef($q, \mathcal{T}$) then finally returns a union of the initial query atoms and the set of the five queries listed above. ■

Query Unfolding

After having discussed the enrichment of a query w.r.t. axioms of a TBox, we will now look deeper into the necessary unfolding steps for directly accessing the data.

Instead of generating materialized triples by evaluating a union of conjunctive queries q on an ABox $\mathcal{A}(\mathcal{M}, \mathcal{DB})$ constructed by mappings $\mathcal{M}$ and a database $\mathcal{DB}$, we "unfold" q according to $\mathcal{M}$, i.e., compute a new query q', which is an SQL query that can be executed over the source relations, such that the set of tuples evaluated by q' over the data source coincides with the set of tuples computed from q and evaluated over DB($\mathcal{A}(\mathcal{M}, \mathcal{DB})$) (see Section 2.4.1). A rule set for generating mappings that connect ontology-based fragments to relational database queries is given below.

Mapping of Graph Patterns to a Database In the case of classical mappings, we define a mapping μ for a concept C as $C(x) \leftarrow \mu_C(x)$ where $\mu_C(x)$ is an SQL query, which further is defined as an unfolding of the query $q = C(x)$ in SQL. A simple example for mapping the concept of $Sensor$ to a table `SENSOR(SID, Sname, Cname, TID)` is given as

$$SQL_{sensor}(x) = \texttt{SELECT SID as x FROM SENSOR s.}$$

Accordingly, we can define a mapping for a property $mountedAt$

$$SQL_{mountedAt}(x, y) = \texttt{SELECT SID as x, Cname as y FROM SENSOR s.}$$

We define a mapping μ of a triple pattern tp to a table $t1$ by giving the algebra for $\mu_{tp} = \pi_{f^s f^p f^o}(t1)$, where $\pi_{f^s f^p f^o}$ is the projection function for the triple pattern tp constructed by π_{f^s} for the subject, π_{f^p} for the predicate and π_{f^o} for the representation of the object. Therefore, our previous sensor example can also be written by the algebraic expressions: $\mu_{sensor}(x) = \pi_{f^s(sensor.sid) f^p() f^o()}(\texttt{SENSOR})$.

Until now we have described the unfolding of simple triple patterns, but in STARQL or SPARQL graph patterns can become more complex. They can be constructed by unions and conjunctions of triple pattern or even optional and filter patterns. We give a short overview on the mapping constructions below.

Let be given a graph pattern $gp = gp_1 AND gp_2$, the mapping is constructed as given below:

$$\mu_{gp} = \mu_{gp_1} \bowtie \mu_{gp_2}.$$

For a graph pattern $gp = gp_1 UNION gp_2$, the mapping is constructed as follows:

$$\mu_{gp} = \mu_{gp_1} \cup \mu_{gp_2}.$$

For a graph pattern $gp = gp_1 OPT gp_2$, the mapping is constructed as given below:

$$\mu_{gp} = \mu_{gp_1} ⟕ \mu_{gp_2}.$$

For a graph pattern $gp = gp_1 FILTER\ expr$, the resulting mapping can be constructed as follows:

$$\mu_{gp} = \sigma_{expr}(\mu_{gp_1}).$$

Example 2. *For concluding with an example let be given the following UCQ:*

$$\begin{array}{rcl} q(x) & \rightarrow & \{Sensor(x),\ mountedAt(x, :turbine1)\} \\ & & UNION\ \{Sensor(x),\ mountedAt(x, :turbine2)\}. \end{array}$$

By applying the rules listed above, we are able to evaluate the mapping for each graph pattern. The result of the unfolding is finally given as:

$$\begin{array}{rcl} \mu_{q(x)} & \rightarrow & \pi_{f^s(sid)}(\mathtt{SENSOR}) \\ & & \bowtie \pi_{f^s(sid) f^o(:turbine1)}(\mathtt{SENSOR}) \\ & & \cup\ \pi_{f(sid)}(\mathtt{SENSOR}) \\ & & \bowtie \pi_{f^s(sid) f^o(:turbine2)}(\mathtt{SENSOR}) \end{array}$$

■

We have given a formal representation for the mapping of $\mu_{q(x)}$. Practically, mapping rules can be noted in different formats, which are described in the following section.

Mappings

When querying databases, where a rewriting of the query is required, the ABox is not directly given. It is implicitly defined by mappings. But not only in the virtual approach mappings are needed, also if we would like to materialize ABox assertions from a relational database mapping can be exploited.

mapping rules can be encoded in a mapping language, which can be classified into four categories [120]: Direct mappings, read-only general purpose mappings, read-write general purpose mappings and special purpose mappings.

In the case of a direct mapping, data is mapped directly from a table into ABox assertions without any further interaction, which means that the data domain of the ontology is directly represented in the model of the relational data.

Practically, each table name is mapped into a specific class with the same name, while each row of that table is mapped to an individual member of the table class. Further, each column name is mapped to a property that connects the individual, either to another individual in the case of a foreign key or to a literal otherwise. This process of direct mapping generation can be done by an so called automatic bootstrapping algorithm such as in BootOx (e.g. see [127]).

General purpose mapping languages are much less restricted. They are defined by rules with an ontological query on the left hand side and a query in the language of the data base source (e.g., SQL) on the right hand side (see [191]). Mapping rules can be quite expressive, which results in complex mapping formulations and is a reason why many mapping languages are read-only such as D2RQ [45] or R2RML as a W3C recommendation [230].

Read and write general purpose languages are usually less expressive, while suffering from the *view update problem*. Thus, updates to the database from a view perspective are only possible by restrictions to the mapping language. An example for a read-write general purpose mapping language is R3M [119].

The W3C RDB2RDF working group [231] recommends the mapping language R2RML as one standard for realizing mappings from a relational database to RDF graphs.

R2RML. R2RML is a specification for a read-only mapping language and designed as being independent from any given application. Its mappings, which are defined in RDF itself, create an RDF view to a database source. The R2RML mapping language consists of several constructors and we give an overview of the most important types below.

- The **Triples Map** is the main construct in a mapping definition. It describes the triple specification mapped to a logical table consisting of several sub parts listed below.
- The **Logical Table** is defined as a table or view, which is the data source for the triple generation. It could be directly noted as a table/view name or indirectly as a SQL query.
- The **Term Maps** specify the triple values mapped from the logical tables in subject, predicate, object or graph maps as listed below. It could be of a constant value, of a column-value retrieved from a column of the logical table or a template value for specifying custom URIs, blank nodes or literals.
  - The **Subject Map** maps the subject specifications for each triple as a URI or blank node. Can contain a graph map that connects each triple with the subject to a specific graph.
  - The **Predicate Map** specifies predicate terms, which are usually constant values. It is paired with objects by the *PredicateObjectMap*.
  - The **Object Map** specifies URIs, blank nodes or literals as objects and is paired with predicates by the *PredicateObjectMap*. It also can contain additional data types or language tags.
  - The **Graph Map** stores triples in specific graphs. Further, it is included in the subject- or predicate object map and must contain an URI for specifying the graph name.

A short example for a R2RML mapping from the sensor measurement scenario is given in Figure 2.9.

Here, two tables, containing data about sensors and assemblies, are mapped to their triple representation. The mapping is a straightforward direct mapping, except for the naming scheme of each subject URI, which uses a template, and the two columns *AssemblyName* and *SensorName*, which are both mapped to the predicate *hasName*. One additional R2RML construct used here is the *RefObjectMap* that was not discussed before. The *RefObjectMap* is a construct used in the case of foreign keys, like in the given example, where the *location* column of the sensor table is a key to the assembly id of the assembly table. In fact, we define a parent

SENSOR

SId	SensorName	Location
S1	TempSens	A1
S2	PresSens	A1
S3	SpeedSens	A2

ASSEMBLY

AId	AssemblyName
A1	AssemblyOne
A2	AssemblyTwo

```
@prefix rr : <http://www.w3.org/ns/r2rml#>
@prefix sn : <http://www.sensor.net/>

sn:AssemblyMap
   a rr:TriplesMap;
   rr:logicalTable [ rr:tableName "Assembly" ];
   rr:subjectMap [
      rr:template "http://www.sensor.net/assembly/{AId}";
      rr:class sn:Assembly;
   ];
   rr:predicateObjectMap [
      rr:predicate sn:hasName
      rr:objectMap [ rr:column "AssemblyName" ];
   ];
   rr:predicateObjectMap [
      rr:predicate sn:hasId
      rr:objectMap [ rr:column "AId" ];
   ];

sn:SensorMap
   a rr:TriplesMap;
   rr:logicalTable [ rr:tableName "Sensor" ];
   rr:subjectMap [
      rr:template "http://www.sensor.net/sensor/{SId}";
      rr:class sn:Sensor;
   ];
   rr:predicateObjectMap [
      rr:predicate sn:hasName
      rr:objectMap [ rr:column "SensorName" ];
   ];
   rr:predicateObjectMap [
      rr:predicate sn:hasId
      rr:objectMap [ rr:column "SId" ];
   ];
   rr:predicateObjectMap [
      rr:predicate sn:hasLocation
      rr:objectMap [
         a rr:RefObjectMap;
         rr:parentTriplesMap sn:AssemblyMap;
         rr:joinCondition [
            rr:child "Location";
            rr:parent "AId";
   ]; ]; ];
```

Figure 2.9.: Example tables and R2RML mapping in a sensor based scenario

column (the adressed table column) and a child column (the foreign key). Both are joined in a *joinCondition* of the R2RML mapping, which can be represented by a related SQL query join.

Mapping Related Implementations of Classical OBDA There have been early approaches and implementations on the paradigm of OBDA before R2RML became a standard by the W3C community. A survey on different mapping implementations of the W3C RDB2RDF incubator group gives an overview on early systems in 2009 [208]. In 2009 various applications and specific mapping languages existed. We provide some examples. Virtuoso [91] takes a direct mapping approach that maps tables to RDFS classes, but also takes foreign and primary keys into consideration. The mappings are composed of *quad map patterns*, which define the RDF views from relational table columns.

D2R and its mapping language *D2RQ* [45] provides multiple options for accessing RDB data, including a materialization or *RDFdump* approach, Jena/Sesame API and SPARQL endpoint with user defined mappings.

R2O [200] is a declarative and XML-based mapping language implemented by the ODEMapster engine [201] for virtual or batch use.

The Dartgrit toolkit provides a list of mapping and querying tools for RDB2RDF approaches. Mappings are defined by the user in a visual table tool, while the creation of SPARQL queries is assisted by a visual tool as well. These queries are then translated to SQL queries with respect to the defined mappings.
With the presentation of R2RML as a standard by the W3C, most applications adopt this mapping language.

Recent systems on the market that implement R2RML mappings, are the *Ontop* system [60, 203] with implemented optimization strategies on the query rewriting and unfolding side as shown in [202, 204], the morphRDB system [109, 194], as well as the kyrie transformator [169].

2.4.3. ABDEO

As observed, FOL rewritability of DL-Lite induces only restricted representation capabilities at the ontology level. For example, number restrictions are not allowed, though those structures are sometimes necessary to express interesting constraints. Another example are transitive roles, as they directly prevent FOL-rewritability.

Transitive roles are useful to model part-of relations for describing turbine topologies, as an example of the sensor measurement scenario.

Expressing transitive structures requires a query language that supports recursion, which is not available in SQL. Nevertheless, such a rewriteability for more expressive description logics (e.g., $\mathcal{SHI}$) (see Section 2.3.1) is desired. As the rewritability does not hold in these cases, other approaches have been investigated that materialize only small parts of the complete dataset. They are subsumed under the acronym ABDEO[14].

In [238] the authors present an idea for ABox modularization, which investigates the TBox for splitting a huge ABox in smaller manageable parts. The resulting (small) ABoxes are used for reasoning services such as instance retrieval in many practical cases, where it is not required to use the complete large ABox.
The approach is extended in [167] for processing grounded conjunctive queries [167, 238].

2.4.4. Temporalizing OBDA

We already discussed settings of processing static data. Now, we would like to extend this view in order to add a fourth dimension to represent temporal RDF data, by mapping extensions that consider time for the RDB2RDF approach.

Temporal data processing has been under research for many years, for example by using temporal logic in the areas of computer vision [177, 178, 198] and artificial intelligence [24, 46, 117, 118, 146, 147].

A very simple use of time domains would be a direct extension to the OBDA approach of mapping static data described above (e.g. in. tOWL [165]). Here, we introduce an additional time column to the desired table. By sticking to the sensor measurement scenario, let a table $Measurement$ have the following columns: $Timestamp$, $SensorID$, $Value$, which is a common case in measurement scenarios, as sensors measure different values at different points in time.

A direct mapping (see Section 2.4.2) that maps tables to classes, rows to individuals and columns to properties, would produce individuals of a *measurement* class connected to properties that add the sensorId and timestamp as new objects or literals.

The downside of this direct mapping is the additionally required measurement object for formulating the time dimension. Instead of adding a fourth dimension and

[14][A]ccessing [B]ig[D]ata with [E]xpressive [O]ntologies

storing quads, we require three additional triples. The representation of timestamps is a known problem in research and discussed under the term *temporal reification* (e.g. see [8, 96]).

Non-reified time is modeled by a so-called *flow of time*, an ordered structure of timepoints $(T, \leq)$. With the *flow of time* we can consider one interpretation $\mathcal{I}$ per timepoint $t \in T$, while all constants stay rigid and are interpreted in the same way for each interpretation (see also the handbook of modal logic [121]).

While FOL temporal logic makes use of time as a predicate, modal temporal logic is a state based approach, where we can access states of the *flow of time* on a more abstract level. Thus, it is possible to make relative statements seen from the current state, e.g., for accessing the next state or some state in the past in the ordered time sequence without using absolute time values.

Therefore, we can adjust interpretations w.r.t. temporal logic regarding a flow of time $(T, \leq)$. Having a family of interpretations $(\mathcal{I}_t)_{t \in T}$, we can say that $\mathcal{I}_t \models \mathcal{A}_t$ for every $t \in T$.

Approaches for extending description logics with a time dimension can be found in [21]. More practical approaches for extending RDF triples and SPARQL with time dimensions are found in [43, 112, 113, 172, 186, 221].

Strabon[15] [43], which was originally designed as a geospatial extension, extends the W3C standard SPARQL (see Section 2.3.3) by a fourth dimension (in this case: time). While this additional dimension element consists of an absolute timepoint in the dataset that links every temporal triple to a subgraph, which contains the temporal ABox, Strabon also provides operators to query time intervals besides single timepoints. Similar examples for extending SPARQL with time annotations are $\tau-$SPARQL [221] or TA-SPARQL [199].

Recent work considering temporal OBDA can be found in [25, 49, 50] and describes the new temporal query language *TCQ* for temporal conjunctive queries. The approach from Baader et alhas developed in recent years from using a classical DL-Lite TBox to an expressivity of $\mathcal{SHOIQ}$. While a finite sequence of ABoxes is used to simulate the *flow of time* supported by linear temporal logic operators added to the query language of TCQ. LTL operators (e.g. $\bigcirc$ next state, $\diamond$ some time in the future) allow the formulation of advanced temporal queries, such as "a critical event happened exactly three times between ten time points in the past and now". Nevertheless, only small parts of this expressive language are FOL-rewritable and as such *TCQ* itself is restricted to a materialized and non virtual OBDA approach (see Section 2.4).

[15] http://www.strabon.di.uoa.gr/

Other approaches also include operators of modal temporal logic in the TBox as seen in [23].

2.4.5. Streamifying OBDA

In the last section we tried to extend the view of OBDA with approaches on temporal data. Having in mind the extended examples on temporal data, we can view those for streaming data from two perspectives: the perspective of relational streaming systems and the perspective of a streamed RDF model that uses an appropriate query language for streams. We already discussed several Data Stream Management Systems in Section 2.2. Therefore, we are going to extend the query model, also described by the W3C standard query language SPARQL, for streams.

RDF Stream Model. We formalize the RDF stream notation similarly to the model given in [29, 143, 156] and [6]. An RDF triple is a tuple $\langle s, p, o\rangle$ consisting of subject, predicate, object as given in the RDF description of [141], see Section 2.3.3. We define a *timestamped triple* st as a pair of a triple and a timestamp $(\langle s, p, o\rangle, \tau)$, where the timestamp τ is defined on a sequence of monotonically increasing time values $(T, \leq)$.
Then an RDF Stream can be described as a (potentially) unbounded sequence of timestamped triples in monotonically increasing order. For every $i > 0$, $(\langle s_i, p_i, o_i\rangle, \tau_i)$ is a timestamped RDF statement and the stream S:

$$\begin{aligned} &\dots \\ (\langle s_i, p_i, o_i\rangle, \tau_i) \\ (\langle s_{i+1}, p_{i+1}, o_{i+1}\rangle, \tau_{i+1}) \\ &\dots \end{aligned} \tag{2.8}$$

is a sequence on RDF triples in non decreasing order over the timestamp values τ, where $\tau_i \leq \tau_{i+1}$.
The definition of an RDF Stream is not enough for actually accessing ontology based streaming data. For example, operators have to be defined for accessing the data in a sliding window fashion, operating on temporal sequence or merging streams with static data and much more. In the literature some approaches are already provided, most cases extend the standard for querying RDF data with temporal and stream functionalities other use a mix of SPARQL and temporal operators. In the following we give an overview on the most important approaches.

2.5. Stream Based SPARQL - Extensions

Following the idea of streamifying OBDA from the last section, we see that the underlying streaming systems and query languages, already presented in Section 2.2, are explicitly designed for relational models and languages that are not able to formulate ontology based queries. Therefore, we have an additional interest in ontology based streaming query languages. The most obvious solution that most applications choose is an extension of the W3C standard for querying RDF data: SPARQL (Section 2.3.3). In this section we give an overview and a comparison of the main applications for querying RDF data streams. We show an example query in the sensor measurement scenario for each language that demonstrates specific operators and functionalities in each case. A discussion on SPARQL extensions for streams can also be found in [56].

2.5.1. Streaming SPARQL

Streaming SPARQL [48] was the first approach for a SPARQL streaming extension with window operators in 2008. While extending the query language with streaming operators, it is the goal of Streaming SPARQL to preserve as much syntax and semantics of the SPARQL W3C standard as possible. Streams are explicitly stated in the `FROM` clause of the query together with window annotations. A special feature of Streaming SPARQL (compared to other approaches) is that they allow an additionally finer window granularity in the graph patters of the `where` clause (see e.g. Listing 2.6).

Streams are identified with the `STREAM` keyword, followed by an IRI. The language provides several different window types, starting with sliding windows, which require parameters that are defined by `RANGE` (width) and `SLIDE`. Tumbling windows can be abbreviated by a keyword `FIXED` instead of the slide parameter. Or it is directly set to one, if neither a `SLIDE`, nor a `FIXED` keyword is used. On the other hand, tuple based windows can be defined by using the `ELEMS` keyword, followed by a tuple count instead of a time based definition with `RANGE`.

As mentioned above, additional sub windows can be defined explicitly for graph patterns. In that case, the inner window definition is preferred to the original window definition and evaluated w.r.t.the graph pattern (see Listing 2.6).

An example query in streaming SPARQL is given in Listing 2.6. It contains two window definitions based on time in the `FROM` clause or tuple count in the optional part of the graph pattern, respectively. The results of the query show all assemblies

with measured values from the last 30 minutes, which are included in the last 100 measurements.

Listing 2.6: A query formulated in Streaming SPARQL (values of 30 minutes)

```
1  PREFIX sn : <http://www.sensor.net/>
2
3  SELECT ?sens ?z
4  FROM STREAM <http://www.sensor.net/data/sd.RDF>
5     WINDOW RANGE 30 MINUTE SLIDE 1 MINUTE
6  WHERE {?x sn:hasSensor ?sens .
7     OPTIONAL { ?sens sn:hasValue ?z . WINDOW ELEMS 100 }}
```

2.5.2. C-SPARQL

C-SPARQL [30] was proposed in a first version about one year later in 2009 and in a second version in 2010 [32] by Barbieri et al. It was the first SPARQL streaming extension that offered aggregation by an additional aggregation clause, while SPARQL 1.0 itself did not support aggregations. With the introduction of SPARQL 1.1 [115] as a W3C standard, which includes aggregations, C-SPARQL dropped its aggregation clause and followed the notation of SPARQL 1.1 in 2011 [29].

The authors use a similar approach as in Streaming SPARQL regarding window definitions, but support only a single window per stream. Additionally, they adopt the idea of registering continuous queries and execute them regularly on the system as seen in Listing 2.7.

In the example an evaluation frequency of ten minutes is given. Periodic evaluation and computation gets more important for queries that use more than one stream with several window slides. Although, the window formulation for a stream is only changed in its syntax, C-SPARQL also allows a join of streamed and static data, which can be added in a `FROM` clause without `STREAM` keyword (see Listing 2.7). Additionally, an example aggregation function is used that measures the average sensor value in each window.

2.5.3. CQELS

CQELS[16] in [189] was presented in 2011 as the first complete query engine for unifying the process of querying Linked Stream Data and Linked Data. It differs

[16]Continuous Query Evaluation over Linked Streams

Listing 2.7: A query formulated in C-SPARQL (average of 30 minutes)

```
1  PREFIX sn : <http://www.sensor.net/>
2
3  REGISTER QUERY AvgSensorVals30min COMPUTE EVERY 10m AS
4  SELECT ?sens ?loc avg(?z) as ?avg
5  FROM STREAM <http://www.sensor.net/data/sd>[ RANGE 30m STEP 1m ]
6  FROM <http://www.sensor.net/data/location>
7  WHERE { ?sens hasLocation ?loc . ?sens sn:hasValue ?z }
8  GROUP BY { ?sens, ?loc}
```

from the previous approaches in the case that it is not only an extension of the SPARQL query language with streaming operators, but a complete streaming engine built for linked data, whose language differs to the previously presented query languages in several details. And as such it does not rely on query transformation and backend systems. Therefore, the complete query processing is under control of the CQELS engine. The advantage of that solution is a query specific reordering of operators during query execution for reducing delay and complexity of the execution process.

We give the example query for the 30 minute average of sensor values in 2.8.

Listing 2.8: A query formulated in CQELS (average of 30 minutes)

```
1  PREFIX sn : <http://www.sensor.net/>
2
3  SELECT ?sens ?loc avg(?z) as ?av
4  FROM NAMED <http://www.sensor.net/data/location>
5  WHERE {
6     STREAM <http://www.sensor.net/data/sd>[ RANGE 30m ]
7        {?sens sn:hasValue ?z}
8     GRAPH <http://www.sensor.net/data/location>
9        {?sens hasLocation ?loc}
10 }
11 GROUP BY ?sens, ?loc
```

One can see that the declaration of windows has changed quite a bit. The `FROM` clause in CQELS only includes static linked data graphs, where streams are declared directly in the where clause with its own graph pattern each. The goal of managing linked data in graph patterns directly for each stream or static graph is, on the one hand, to divide between data from each stream with a named reference, which is not possible in other languages, and on the other hand for optimization purposes of the processing, where operators can handle streams separately.

A very important issue regarding CQELS is the relation-to-stream operator that evaluates CQELS triples as fast as possible without respect to any slide parameter. Therefore, only the latest tuples are considered in the result set (see Section 2.1.5), which is evaluated each time a new tuple enters the window (Content-Change Policy), while C-SPARQL uses a Window Close, Non-empty Content policy and a periodic evaluation with respect to its slide parameter.

2.5.4. SPARQLStream

Having been developed between 2010 and 2012, *SPARQLStream* [56, 57, 58] is influenced and inspired by previous stream extensions to SPARQL (e.g. C-SPARQL). In comparison to C-SPARQL, it also supports features of the SPARQL 1.1 standard (e.g. aggregations), but relies on an OBDA query transformation approach with backend streaming systems, which is a big difference compared to C-SPARQL and streaming SPARQL above that only support access on materialized linked data.

An example query formulated in SPARQLstream is shown in 2.9.

Listing 2.9: A query formulated in SPARQLstream (average of 30 minutes)

```
1  PREFIX sn : <http://www.sensor.net/>
2
3  SELECT DSTREAM ?sens ?loc avg(?z) as ?avg
4  FROM NAMED STREAM <http://www.sensor.net/data/sd.srdf> [NOW - 30 MINUTES]
5  WHERE {
6     ?sens sn:hasValue ?z .?sens hasLocation ?loc
7  }
8  GROUP BY ?sens, ?loc
```

Compared to CQELS, it lacks a possibility to distinguish between graph patterns for each stream. Nevertheless, SPARQLstream does not distinguish between static and streaming data and therefore we are also unable to identify *hasLocation* as a static property.

Additionally to other languages it supports window-to-stream operators inspired by CQL (see Section 2.1.5). In the example shown above, the DStream operator generates only results, which are no longer a valid output compared to the previous evaluation step.

2.5.5. EP-SPARQL

The approaches reviewed until now do not support detection of state sequences or comparison of temporal states. *EP-SPARQL*[17] [11] tries to solve this issue, as it focuses on *situatedness of triple assertions*, meaning sequence based event streams. EP-SPARQL is based on the underlying streaming system ETALIS [12] and extends SPARQL by four new binary operators: `SEQ`,`EQUALS`, `OPTIONALSEQ` and `EQUALSOPTIONAL`. The operators are used to combine graph patterns in the same way that `OPTIONAL` and `UNION` do in SPARQL, but as joins based on temporal relations.

For example the expression BGP_1 SEQ BGP_2 states that two basic graph patterns build a sequence and occur next to each other for two points in time, while BGP_1 EQUALS BGP_2 states that two graph patterns occur exactly at the same timepoint. The two additional operators OPTIONALSEQ and EQUALSOPTIONAL are the equivalent time based versions of the original OPTIONAL in SPARQL.

Moreover, as no other new operators are added to SPARQL in the case of EP-SPARQL, we witness that a window operator is totally missing. Instead, a function *getDURATION()* is added to the filter conditions, defining a duration for the expressed graph pattern time sequence. Examples given in [11] are exclusively based on stock exchange scenarios. Nevertheless, we adopt the sensor based example query from above for EP-SPARQL in Listing 2.10.

Listing 2.10: A query formulated in EP-SPARQL (values incrby 100 in 30 mins)

```
1   PREFIX sn : <http://www.sensor.net/>
2   PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
3   SELECT ?sens
4   WHERE { ?sens sn:hasValue ?val }
5      SEQ { ?sens sn:hasValue ?val2 }
6   FILTER (?val + 100 < ?val2 &&  getDURATION() < PT30M^^xsd:duration)
```

2.5.6. TEF-SPARQL

The authors of [138] propose a new type of data model in the RDF streaming context and designed a query language called *TEF-SPARQL* [138]. TEF-SPARQL converts information from events, which is stored as facts in a temporal table for being maintained between different window instances of the stream. The facts are usually triggered by events to represent temporarily valid background data.

[17]Event Processing SPARQL

Listing 2.11: A query formulated in TEF-SPARQL (values incrby 100 in 30 mins)

```
1    PREFIX sn : <http://www.sensor.net/>
2
3    CONSTRUCT FACT AssemblyStart {?ass sn:hasStatus sn:critical}
4    WHERE ((SINCE ?ass sn:hasSensor ?sens . ?sens sn:hasStatus sn:
         maxTempReached)
5       UNION(TILL ?ass sn:hasSensor ?sens . ?sens sn:hasStatus sn:normal)
6    )
7
8    SELECT ?ass AS ASSEMBLY
9    (AGGREGATE COUNT ?ass
10   WHERE (CURRENT ?ass sn:hasStatus sn:critical)
11   EVERY ''P10S''^^xsd:Duration)
```

Therefore, the language proposes two new kinds of basic graph patterns: Events and facts, which can be connected by different operators. We give an example of a TEF-SPARQL query in Listing 2.11 for further explanations.

The query consists of two parts. The first part creates facts about assemblies in a critical state if their sensors have reached the maximum temperature. This status holds until the sensor reaches a normal level again. The second part is a select query that counts all assemblies, which are currently in a critical state. Several other operators for creating temporal facts are allowed in TEF-SPARQL, such as `BEFORE`, `DURING`, and `WITHOUT`. For more details we refer the reader to [138]. An implementation of TEF-SPARQL with a comparison to EP-SPARQL and C-SPARQL can be found in [97].

2.5.7. RSP-QL

The *RSP-QL*[18] [6, 89] query language was recently proposed by the W3C RSP Group, which has started to work on a common model for querying RDF streams. The group is a team of authors, previously working on CQELS, C-SPARQL and SPARQLstream, that tries to develop a unifying formal model for querying RDF streams, which combines the different semantics of the existing systems with elements from the streaming world (CQL [16] and SECRET [51]).

Their goal is to find a standard query language, which is able to express all desired operators from previous approaches. The `FROM STREAM` clause with the use of multiple windows on one stream, direct timestamp access from C-SPARQL, window-to-

[18] RDF Stream Processing - Query Language

stream operators such as in SPARQLstream, named sliding windows with references from CQELS, facts from TEF-SPARQL and finally sequences from EP-SPARQL.

However, this work is still in the beginning and no clear decisions have been made, only some preliminary ideas and proposals exist.

An example query formulated in the current status of RSP-QL is shown in Listing 2.12.

Listing 2.12: A query formulated in RSP-QL (average values of 30 minutes)

```
PREFIX sn : <http://www.sensor.net/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
REGISTER STREAM :GallehaultWasTheBar AS
CONSTRUCT ISTREAM {
?sens sn:hasVal ?val2; sn:hasAvg ?avg;
}
FROM NAMED WINDOW :oneDay ON :stream1 [RANGE P1D STEP PT1H]
FROM NAMED WINDOW :oneHour ON :stream1 [FROM NOW-PT30M TO NOW]
WHERE {
 WINDOW :oneDay {
  {?sens sn:hasVal ?val}
 }
  WINDOW :oneHour {
  {?sens sn:hasVal ?val1} BEGIN AT ?t1
  {?sens sn:hasVal ?val2} BEGIN AT ?t2
  FILTER(?t1<?t2 && val1 + 100^^xsd:float < val2
   && abs(?t1-?t2)<"PT5M"^^xsd:duration )
 }
AGGREGATE {
GROUP BY ?sens
AVG(?val) AS ?avg
}
```

The query selects the average value of sensors that have increased by 100 for the last five minutes in a window of 30 minutes, and it shows several features inspired by from C-SPARQL, CQELS and SPARQLstream.

2.6. Comparison of Semantic Streaming Languages

In the last section we showed that a lot of different approaches exist for accessing RDF based data streams, which all have its own features and advantages or disadvantages regarding functionalities.

But till now, none of the available query languages was really satisfying especially w.r.t. reasoning about time and timepoints as it is necessary for streaming data,

where the main goal is to search for changes of high speed incoming data over time without any memory overhead.

For that purpose a group around the W3C RDF streaming community began to evaluate the available proposals with respect to several benchmarks and tried to find a semantic baseline, which would be able to describe most streaming applications for RDF data under a hood. They designed their benchmarks in categories of functionality, correctness and performance and explicitly with respect to semantics of the query languages.

In this section we are going to analyze work that tries to connect the approaches above. We will start by discussing general stream semantics for relational and linked data and follow with an overview on available benchmarks.

2.6.1. General Semantic Models for Streams

Based on the observation of many different approaches and query engines, several proposals have been made for abstract models that are able to describe and predict the general behavior of these systems in the world of relational streams [51, 148, 156, 184] as well as for linked data streams [37]. In the following, we will discuss two of the most important general stream semantics for this work. The SECRET model for relational data and the LARS framework for RDF data streams.

Semantics for Stream Processing Engines - SECRET

SECRET defines an abstract model for analyzing execution semantics of relational streaming engines with a focus on time based windows and was proposed in [51]. The authors developed their semantical model along four dimensions ScopE, Content, REport and Tick (SECRET).

Scope. The Scope as a function maps an application-time value into a time interval, which can be then evaluated by a continuous query. An input value to scope is the starting time t_0, which is the application time of the very first starting window and is measured, depending on the streaming engine, as an absolute time value. Besides that, the scope is also computed by the window width and slide parameters of the query.
As an example, we assume a given query q with width 5 seconds, slide 2 seconds and an additional system starting time of 10 seconds. Then the Scope of window

w at time $t = 10$ seconds is Scope(10) $= (6, 10]$ meaning that we have at timepoint ten a window with open border at time 6 and closed border at 10. Though, that the window opens at timepoint 6 and closes at timepoint 10.

In the Scope definition of SECRET, the author defined the scope to map to the earliest open window. But in fact, it could also be defined as the most recently closed window as in [184] or all open windows respectively. Choosing one of the different definitions stays a design decision.

Content. The Content is a complementary dimension to Scope and formalizes exactly the object set according to a stream S within the interval of the Scope. And as such can be viewed as the mapping from application time t_n to elements of S. Unlike scope, the content directly depends on the actual data of the stream and how much of it is already available. So it might return different values even with the same value of t_n concerning different systems and settings.

Report. A Report formalizes conditions for evaluating a window's content and for reporting results. According to SECRET, it can be defined in four categories:

- **On content change**: A report is only done w.r.t. t if the content of time t changed compared to $(t - 1)$.
- **On window close**: A report is emitted each time t a window closes.
- **On non-empty content**: Reporting is done only for windows at a timepoint t that are not empty.
- **Periodically**: A report is only done if the current time t is a multiple of the frequency λ.

Of course a system can also combine several of these definitions for their reporting strategies and, e.g., say a report is done each time a window closes and its content is non-empty.

Tick. Tick defines conditions under which the system reacts and takes actions on its input, i.e., adding newly arrived data to the active time window for being evaluated.

Basically there are two strategies defined by SECRET. First, the system can react in a *tuple-based* way each time a new data tuple arrives to the system. And on the

other hand a system can react in a *time-based* way according to the application time and can add new tuples to the window to be processed for each new time instant.

LARS Framework for Stream Reasoning Query Languages

In comparison to relational query languages, the W3C community for RDF stream processing [233] developed its own streaming model. The LARS Framework for Stream Reasoning Query Languages was presented in [87] and previously in [37] as a unifying language in which stream reasoning query languages can be translated to. It may serve as a formal language to express and compare semantics of respective languages to analyze the semantic differences between CSPARQL and CQELS. The general semantics and transformation model of LARS is shown in 2.10, preliminary work for this model can also be found in [35] and [36]. Furthermore, the transformation strategy is organized in four steps tailored for CSPARQL and CQELS, but also applicable for non RDF streaming engines:

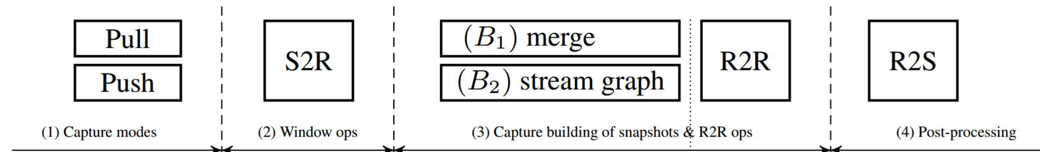

Figure 2.10.: The LARS streaming model to capture RSP queries (from [37])

1. The two general push (CQELS) and pull (CSPARQL) modes of the example languages are represented in the LARS program.
2. The window definitions are transformed into general LARS window operators (see Figure 2.11).
3. The relation-to-relation operators of the query language are formed into a Datalog program and LARS operators, e.g., given as $\boxplus @_t a$ for "a holds at time point t" or $\boxplus \diamond a$ for "a holds at some given time point". In the case of CSPARQL and CQELS, two general strategies can be followed that join all input streams in the default graph (CSPARQL) or each stream graph pattern has to be directly transformed into LARS rules separately as in CQELS.
4. A post-processing method imitates RStream, IStream or DStream operators of CQL (as seen in SPARQLstream).

The analysis of CSPARQL and CQELS with the help of the LARS framework in [87] shows that the output of both streaming engines is identical under certain

Window expression ω	$\tau(\omega)$
`[RANGE L]`	$\boxplus^{L}$
`[RANGE L SLIDE D]`	$\boxplus^{L,0,D}$
`[ROWS N]`	$\boxplus_{\#}^{N}$
`[NOW]`	$\boxplus^{0}$ or $\boxplus_{NOW}$
`[RANGE UNBOUNDED]`	$\boxplus^{\infty}$

Figure 2.11.: The LARS operators as shown in [37]

conditions. Those are for example *(i)* the query does not include any `MINUS` or `NOT EXISTS` operator, *(ii)* the query only uses time-based windows with a slide of 1, *(iii)* the static dataset and the streaming data sets are disjoint.

The investigation on streaming semantics of SECRET and LARS shows that not only the syntax of stream query languages differs significantly, but also the semantics and data processing in the stream engine. The pros and cons of each query language and engine make the search for a standard streaming language a difficult task. Therefore, some benchmarks on streaming engines for linked data have been established in recent years. The following section gives on overview on the most important implementations.

2.6.2. Benchmarks for Linked Data

Regarding RDF streaming data, most benchmarks concentrate on functionality and correctness tests of the query language and thus, only some minor performance benchmarks for stream processing engines exist (e.g., CSRBench [88], YABench [142]), although some approaches have been implemented for comparing specific systems that implement their own engines, such as CQELS, C-SPARQL and JTALIS in [189] or TEF-SPARQL, C-SPARQL and EP-SPARQL in [97]. However, systems that rely on query transformations and external query processing are not directly comparable with respect to performance, because the direct query execution depends on external processors.

Linear Road Benchmark

The Linear Road Benchmark[19] [18, 223] is a benchmark for Data Stream Management Systems based on relational data and as such can be used to benchmark

[19]http://www.cs.brandeis.edu/~linearroad/

backend streaming systems w.r.t. the OBDA approach for comparing the performance of systems although, it is not directly connected to OBDA or linked data.

The LRB can be seen as a specification for a adaptive tolling system on express highways, where tolls are determined by changing factors on the street, such as traffic and accidents.

The goal of the tolling system is to discourage drivers of using high traffic roads, because of an increased toll. On the other hand, the drivers would use most likely less frequently used highways, because of decreased tolls.

Each vehicle on the highway emits its exact location, coordinates and speed at least every 30 seconds, while the system computes tolls for each segment of each highway in real time. The position data for each vehicle is provided by a data generator, which can be downloaded from the web page of the project and generates data as streams to the DSMS, where a combination of continuous and historic queries evaluates the tolling.

Linear Road tests Data Stream Management Systems by measuring how many express highways a system can handle in real time by giving time constraints for each query. A validation tool is used to check correctness of the results.

Implementations of the linear road benchmark for several systems exist, e.g., as an evaluation of the STREAM Processing Core system in 2006 [125] or an implementation on the streaming system Storm [225].

Uppsala University Linear Road Implementation The Uppsala University provides an additional implementation available for Windows PCs[20] as described in [220] and [128]. They used their implemented system to evaluate SCSQ[21] (pronounced 'sisque') a scalable data stream management system (see e.g. in [101, 245]). They were able to achieve a score of L=1.5 express highways on a local machine [220] and more than 512 express highways simultaneously on a multi machine cluster (see [246, 247]) with SCSQ.

[20]http://www.it.uu.se/research/group/udbl/lr.html
[21]Supercomputer Stream Query processor

LSBench

The LSBench[22] [154] is a linked data stream synthetic benchmark that uses a social network setting. It was designed and published by the authors of CQELS (see Section 2.5.3) and presented in 2012 for evaluating the CQELS engine against C-SPARQL (see Section 2.5.2) and JTALIS (a Java implementation of EP-SPARQL described in Section 2.5.5). The social media streaming scenario is based on the data generator S2Gen from the CQELS authors, who extended the S3G2 [188] social graph generator by a sliding window approach to form a materialized dataset, consisting of streamed and static data. An example execution as well as results of the benchmark for comparing CQELS, CSPARQL, JTALIS can be found in [154].

The systems are tested with respect to 12 example queries, composed of operations such as joins, nested queries, aggregations and negations. The test results show that the streaming engines still miss several important features such as static data joins for CSPARQL and JTALIS or nested queries for CQELS and JTALIS, as well as negation for all implementations.

Beside these basic functionality tests, the evaluation further showed that the semantics of the three SPARQL extensions had direct influence on the correctness and throughput test. In fact, CSPARQL evaluates queries periodically in specified time intervals and therefore, emits many duplicates at slow data rates and certain inputs are ignored for high data rates. On the other hand, CQELS and JTALIS follow the eager execution strategy, meaning that they evaluate input once as soon as it comes in and thus, do not miss inputs or produce duplicates.

SRBench

SRBench [248] from 2012 was developed by a team around the authors of SPARQL-stream and concentrates on the coverage of SPARQL and SPARQL 1.1 operators by different RDF streaming systems. But in comparison to the LSBench, different operational semantics of the systems are not considered.

The authors evaluated SPARQLstream, CSPARQL and CQELS on the Kno.e.sis LinkedSensorData [116], which is a real world dataset and contains US weather data collected since 2002 from about 20.000 weather stations with a total of 100.000 sensors and roughly 110 GB of data. Additionally, GeoNames [99] and DBpedia[23] data

[22] http://code.google.com/p/lsbench/
[23] http://wiki.dbpedia.org

is used for static data, as the LinkedSensorData refers to geographical places and other subjects in both datasets.

SRBench consists of 17 predefined queries, each of which have different requirements from 7 feature groups, namely: graph pattern matching, solution modifiers (projection and `DISTINCT`), query forms (`SELECT`, `CONSTRUCT` and `ASK`), features of SPARQL 1.1 (e.g., aggregation, negation, property paths), reasoning (RDFS:subClassOf, RDFS:subPropertyOf, owl:sameAs), window operators, and access to different data sets. The queries can be found on the SRBench wiki page [215].

The evaluation using SRBench in [248] shows that the query engines SPARQL-stream, CSPARQL and CQELS are still at the beginning of their development and fail several feature tests.

So for example seven of the given queries could not be answered on any system, because none of these supports property paths of SPARQL 1.1. Furthermore, it is shown that none of the three tested query languages supports `ASK` queries or `IF` expressions, SPARQLstream was not able to execute ten queries, because of missing a join between static and streaming data. Reasoning can only be evaluated for CSPARQL as both others do not support any reasoning feature.

This benchmark led to the CSRBenchmark for additional testing of query correctness and performance in 2013.

CSRBench

A benchmark for testing RDF streaming engines w.r.t. correctness is the CSRBenchmark [88] designed by the SPARQLstream team. First, they analyze the semantics of CSPARQL, CQELS and SPARQLstream regarding the streaming models of CQL (see Section 2.2.6) and SECRET (see Section 2.6.1) and address the different answer sets that the query engines provide for identical queries. Finally, they present an extension of the SRBenchmark with automatic correctness validation called CSR-Bench.

The two streaming models define basic concepts on streams. SECRET introduces two different kinds of timestamps to streams, the system time and application time, additionally the behaviour of time windows is described by the four functions *Scope,*

Content, Report and Tick (see Section 2.6.1), and the three window operators of CQL: IStream, DStream and RStream (see Section 2.1.5).

The evaluation of the streaming model shows that all three streaming engines differ concerning their window operators. The *Report* strategy is realized in CSPARQL and SPARQLstream by period windows, but in CQELS by any given content change. SPARQLstream supports all three window operators of CQL, while both other engines only support Rstream or Istream respectively and time in CSPARQL is measured in seconds, while CQELS and SPARQLstream also support hundreds of milliseconds.

In order to stress the S2R operators of the query engines, the dataset of the SR-Benchmark was retained in the CSRBenchmark, but extended by three query types: a variation of the window size and slide parameters, extended aggregate queries, and comparison of values at different timestamps.

For evaluating the correctness of each system, the authors propose an oracle that is able to generate and check results for the stream engines. The oracle gets as input a stream, a query and the operational semantics for executing the query over the stream. Afterwards, it can check the answer of a real streaming system in comparison to the theoretical result of the oracle. The implemented oracle (see Figure 2.12) is built on top of the Sesame framework and can be downloaded as an open source project, including input data and queries [80].

The author use seven queries on their streaming model in the oracle and compare the results to those of CSPARQL, CQELS and SPARQLstream for a subset of SRBench dataset (the weather data of hurricane Charley).

In the experimental results it is shown that all three systems have correct results regarding window operators and tumbling windows. On the other hand, still three different kinds of errors have been found. CSPARQL shows difficulties in computing correct results for sliding windows with a slide parameter smaller than the window width, because of emitting unexpected windows in the starting phase. As soon as the window has slided for more timepoints than its width, the problem does no longer occur. SPARQLstream has difficulties with aggregations as it uses a different starting time t_0, compared to other systems, resulting in different window inputs and average values. Additionally, average values of 0 are compiled to *null* values, which means that the results are different although the semantics are identical.

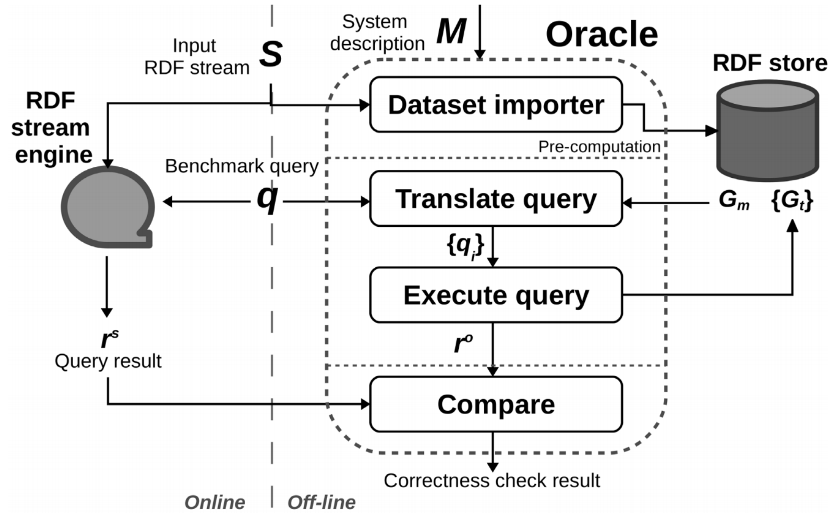

Figure 2.12.: The oracle of the CSRBench (from [88])

Finally, CQELS is unable to compare different time stamps because of its time handling strategy and therefore fails in query six and seven.

YABench

In 2015 another benchmark was shown in a demo connected to the RDF Stream Processing Workshop at ESWC 2015, called YABench[24] [142]. It was flagged as work in progress and to the best of our knowledge no clear description is available at the time of this writing. The authors promised an extension to previous work that includes joint evaluation of functional, correctness and performance testing achieved by five components: stream generator, prepared test queries, integrated streaming engines and a test oracle such as in CSRBench with different semantics and an application for visualization of results.

Results in YABench can be measured by precision, recall and f-measure, while measuring delay and performance. A current version of YABench can be downloaded as open source project [104].

[24]Yet Another RDF Stream Processing Benchmark

2.7. Concluding Remarks

In Chapter 2 we have shown an overview on state of the art technologies in the field of semantic stream access.

After an introduction to general stream access, we gave a survey on relational streaming system and languages that are already well established and highly optimized (e.g., Spark and Flink, see Section 2.2.11). Based on the use of relational data stream management systems, we have described the strategy on ontology based data access (OBDA) for linked data and relational databases.

The final investigation on current languages and systems for semantic stream access has shown that there exist many different systems and languages for different purposes, e.g., EP-SPARQL for event processing and TEF-SPARQL for storing data facts, but there is currently only one system that is able to connect to relational streaming systems (not even to state of the art systems such as Spark).

A second important problem of semantic streaming languages is the handling of the flow of time or temporal states. Most systems are not able to distinguish between timepoints within a temporal window. While some systems use additional temporal function to compare timepoints of different triples (e.g., C-SPARQL, see Section 2.5.2), others see the complete temporal window as a single state (e.g., Streaming SPARQL).

The convoluted handling of time and differences between the current streaming systems for linked data prevents the creation of a standard query language and makes further research for industrial applications necessary. In the following chapter we present a new approach and query language for semantic stream access that unifies the advantages of the presented solutions with the use of temporal sequences.

3. A New High Level Stream Query Language: STARQL

In the last Chapter we introduced several state of the art technologies for querying data streams in general, but also explained data access with respect to ontologies and mappings on relational systems and finally gave an overview on current approaches for accessing RDF streams. In the following, we describe why the presented ontology-based stream querying languages are not sufficient for industrial sensor network and data analytics use cases.

Furthermore, we introduce our own new stream query language called STARQL[1]. STARQL is able to handle the *flow of time* within a window as a state sequence (see Section 2.4.4) together with ontologies in state sequences and comes along with several new features for combining evaluation of live streaming, recorded time series and static data.

The work described below contributes to recent efforts for adapting the paradigm of ontology-based data access to scenarios within streaming data [30, 47, 57, 189] as well as temporal data [23, 49]. The STARQL query language serves the need for industrially motivated scenarios such as semantic sensor stream access on Siemens gasturbines described in the Optique project [64]. STARQL also provides a unified interface for querying historical data—as needed for reactive diagnostics—and for querying streamed data—as needed for continuous monitoring and predictive analytics in real-time scenarios.

In this chapter we proceed with a description of the Siemens sensor measurement use case. Based on the use case, we define resulting problems for this and hypotheses on solutions with STARQL in Section 3.1. We explain the syntax and semantics of STARQL as a possible solution for the problems defined in Section 3.2 and conclude with a deeper look on its features and a comparison to respective functionalities and operators in SPARQL and SPARQL 1.1.

[1] [S]treaming and [T]emporal ontology [A]ccess with a [R]easoning-based [Q]uery [L]anguage

3.1. OBDA Challenges in Sensor Measurement Scenarios

Today, when dealing with huge amounts of data a typical problem is efficient data access, which is an increasingly difficult task with respect to the three dimensions of Big Data: volume, velocity and variety [162]. The increasing volume comes in hand with the need for accessing different sources and formats (which is meant by variety) from a single access point. Industrial engineers are challenged in extracting data from the large pool of information for their tasks in limited time. Therefore, strategies in context of easy query formulation with queries in natural language for non expert users are needed. The Optique project on "Scalable End-user Access" to Big Data' tries to fill this gap as a EU funded FP7 project by bringing ontology based data access to the industrial market [103, 134].

3.1.1. Optique - Use Case

The project is based on two use cases, which offer a real industrial setting with real data and users. Usually, industrial users need to rely on predefined queries for accessing industrial data, but as soon as they need to formulate new queries, the help of IT experts is needed. This situation is a clear bottleneck since it may require a lot of additional effort and time on either side for experts as well as non expert users.

One of the use cases is provided by Siemens[2] [133, 137]. It focuses especially on a combination of temporal and streamed data. In the Siemens scenario a stream of sensor data is delivered with a rate of about 30 gigabytes a day, with many terabytes of historical sensor data already stored in the database.

End users need to combine huge data sets of historical or streaming data as well as non temporal (static) data within a single query. The formulation of queries based on ontologies should help users to define queries in a more natural way. For accessing stream and historical data with ontologies, there is no clear standard available.

Till now, although some solutions exist, a combination of historical and streaming data is generally ignored by the only existing OBDA system (see Section 2.5), which seriously limits the applicability in enterprises such as Siemens, where one has to deal with large amounts of streamed data from many turbines and diagnostic centres in combination with historical temporal relational data sources.

[2] www.siemens.com

The sensor stream setting of the Optique use case will be adopted as a motivating example for specifying the needs and requirements of a new ontology-based data stream query language.

The Sensor Measurement Scenario

Sensor measurements in general can be seen as a typical scenario for applications based on stream reasoning, where reasoning is used to reason about abstract formulations of events or objects (e.g., some named critical event that is actually defined by complex system states). Sensor based use cases are usually described by the standard ontology for semantic sensor networks (SSN ontology, see Section 2.3.2), which is embedded into the context of Linked Open Data and provides concepts and properties related to sensors, measurements as well as topological structures, observations and events. Additionally, the sensor measurement scenario makes our results better comparable to other stream querying languages, e.g., CQELS or SPARQLStream (Section 2.5) in the SRBenchmark (Section 2.6.2), which is especially designed for comparing functionalities in a sensor measurement setting.

As an introductory example, we describe a more simple sensor measurement ontology compared to the W3C standard, also inspired by the industrial setting in the Siemens use case [133].

The use case is based on two different kinds of data. First, continuously queried streaming data from thousands of sensors on thousands of turbines that are combined with an additional event (or message-) input stream, which describes the general current state of the turbine and can be queried simultaneously. Additional messages from the event stream are generated by specific control units or computational modules (seen as black boxes).

Other relevant data of the system is non-streamed and already stored historical or static data. That includes data such as turbine infrastructure data, as well as recorded measurement data from the past.

A simplified version of a static dataset is shown below as a (normalized) relational DB schema. The schema was identified by the Siemens Corporated Technology (CT) division as a good representation of their original usage in central databases and will be used as a running example in the further sections. Primary keys are underlined, and foreign key specifications are introduced in the text.

For the static dataset we describe the topology of the turbine, the sensor (`Sname`) and component names (`Cname`) as well as corresponding types (`TID`). The type

`TID`, points as a foreign key to the `SENSORTYPE` relation, indicating a temperature, pressure or rotation per time measuring sensor.

```
SENSOR(SID, Sname, Cname, TID)
SENSORTYPE(TID, Tname)
```

As mentioned above, streamed data in the turbine scenario consists of two main types, measurements and event data (also called event messages or, shorter, messages). In the following we will describe the schema of sensor measurement data.

```
MEASUREMENT(MtimeStamp, SID, Mval)
```

Here, a measurement is displayed by a tuple of three items, having a timestamp `MtimeStamp`, a sensor ID `SID` (foreign key to `SENSOR`) and an associated value `Mval`, which was measured at the exact time signaled by `MtimeStamp`. The schema of event data can be identified accordingly. The sensorID is replaced by an AssemblyID `AID` and the associated value by an event text `Etext` (not shown here).

As we are also able to process streamed measurement data from the past, stream querying can also be effectively used for event detection in a reactive diagnosis setting by "replaying" historical measurements. While streaming historical events and measurements as fast as possible, the process can be evaluated much faster than in real time (e.g., looking for a specific sequence of past events). We call these recorded data for reactive use cases *historical data*.

Though it is possible to have a stream of data for every sensor and every control unit, we assume in the following that all measurements and events are combined into one single input stream. The two streams are denoted in the following by S_{Msmt}, the stream of measurement data, and S_{events}, the stream of event data.

The measurement streams are streams in the classical sense, namely, homogeneous streams containing timestamped tuples of the same type (here: the same relation schema) and thus could be directly processed by a DSMS such as STREAM (see Section 2.2.6).

It should be noted that in contrast to continuous queries running on a data stream management system, temporal historical queries can indeed also be handled by standard database systems such as PostgreSQL.

Lifting the Data to a Logical Level

Ontology-based query answering is very helpful in the industrial sensor measurement scenario, because for different historical datasets one might use ontologies with different axioms and mappings to the underlying relational data sources (such as PostgreSQL) without changing the queries.

For ontology-based data access the usually chosen method is that of declarative mappings (Section 2.4.2), formally realized as rules with a conjunctive query on the left and a SQL query on the righthand side in which all the variables of the CQ are selected.

For our sensor scenario, we assume that the ontology signature contains, e.g., a concept symbol *Sensor* and an attribute symbol $hasValue$. The following mapping induces a set of ABox assertions, stating which individuals are sensors and are located at the burnertip of a turbine.

$BurnerTipSensor(x) \longleftarrow$
```
SELECT SID as x
FROM SENSOR s
WHERE s.cName = 'BurnerTip'
```

In the same way we can define mappings for ABox sensors of various kinds, which are installed at different components. These assertions are not time based, as the sensortypes do not change over time. Its data is completely static and therefore, we declare the set of non temporal ABox assertions as a *static ABox*.

On the other hand we also define mappings for assertions that change over time such as the $hasValue$ attribute (given below).

$hasValue(x, y)\langle t\rangle \longleftarrow$
```
SELECT SID as x, Mval as y, MtimeStamp as t
FROM MEASUREMENT
```

The ABox axiom of the mapping is written in non-reified time (see Section 2.4.4) with a temporal fourth dimension. Further, we call the set of ABox assertions with (possibly different) temporal dimensions a *temporal ABox*.

The role of the TBox, as a means to constrain the interpretations to the intended ones, is demonstrated for the concepts of sensors and burnertip temperature sensors. The mapping-induced extensions of both concepts should be such that all burner

tip temperature sensors are also temperature sensors and also sensors (for which we do not have a mapping, say).

An example for a TBox is shown below, containing the following axioms:

$$BurnerTipSensor \sqsubseteq TempSensor, TempSensor \sqsubseteq Sensor.$$

For example, assume that the mapping for burner tip temperature sensors generates (virtual) ABox assertions $BurnerTipTempSensor(s_0)$, $BurnerTipTempSensor(s_1)$, $\ldots$. Then, the individual s_0 is also an instance of $TempSensor$, or, to put it in other words, the TBox and ABox entail the assertion $TempSensor(s_0)$. Therefore, a query that asks for all temperature sensors or pressure sensors can be defined without additional mappings if one uses the intensional knowledge defined in the TBox to get all (correct) answers. On the SQL side, the top category of sensors is then simply replaced by the union of its subtypes in a rewriting step (see Section 2.4.1).

The stream of ABox assertions that underlies most of the following examples is the measurement stream S_{Msmt}. The initial example stream, which is called $S_{Msmt}^{\leq 3s}$ here, contains timestamped ABox assertions giving the value of a temperature sensor s_0 on a three seconds interval, starting at $0s$.

$$\begin{aligned} S_{Msmt}^{\leq 3s} \quad = \quad \{ \; & hasVal(s_0, 90^\circ C)\langle 0s \rangle, \\ & hasVal(s_0, 93^\circ C)\langle 1s \rangle, \\ & hasVal(s_0, 94^\circ C)\langle 2s \rangle, \\ & hasVal(s_0, 92^\circ C)\langle 3s \rangle \; \} \end{aligned}$$

The input streams on which the query language operates are different from relational streams; in the latter, the domain of stream objects are tuples of some type, more concretely, instances of a relational schema. While in the OBDA/ABDEO setting for stream processing, the domain of streamed objects are ABox assertions over a given ontology signature. In general, these streams are inhomogeneous, meaning that they may contain timestamped ABox assertions with different signature elements.

For instance in the (virtual) stream of measurements, we may additionally have timestamped ABox assertions of the kind $Event(criticalEvent_0)\langle t_0 \rangle$ at arbitrary timepoints, which make the sensor stream heterogeneous.

$$
\begin{aligned}
S^{\leq 3s}_{Msmt} \;=\; \{\; & hasVal(s_0, 90°C)\langle 0s\rangle, \\
& hasVal(s_0, 93°C)\langle 1s\rangle, \\
& Event(criticalEvent_0)\langle 1s\rangle, \\
& hasVal(s_0, 94°C)\langle 2s\rangle, \\
& Event(criticalEvent_0)\langle 2s\rangle, \\
& hasVal(s_0, 92°C)\langle 3s\rangle \;\}
\end{aligned}
$$

It is not necessarily the case that we use heterogeneous streams in the scenario directly, but indirectly as we have to combine two streams (e.g., a measurement and an event stream), which are merged internally and have to be handled as heterogeneous stream in the sense defined above.

3.1.2. Natural Query Examples

Having described the data setting, we will now show some tasks for typical users. These tasks build the basis for requirements of the stream query language that we propose. A general task that users face in context of diagnosis is monitoring a sensor stream and looking for sequences of typical or specific event messages. Interesting sequences of values should be specified and formulated in advance by a non-expert user. Afterwards, a data stream management system can evaluate the resulting (and potentially complex) query on a DSMS.

Further examples for interesting diagnostic showcases that can be explored for a single time window have been evaluated in the Optique project and are listed below.

- The signal itself exceeds a predefined value.
- The frequency of spikes or outliers exceeds some value in a defined interval.
- Signal noise exceeds a specific level.
- Two sensor signals are correlated or not.
- The sensor signal is locked and does not change anymore, or its standard deviation is below a specific value.
- The value of a specific sensor monotonically increases (or decreases).
- Looking for temporal error patterns on incoming signals.

We are able to identify three major categories of components to express the tasks above: (i) basic functionalities, such as filtering values by a threshold, (ii) aggregation operators to separate outliers from average values, and (iii) temporal sequence operators for defining temporal patterns on the sensor data.

The specification of tasks and patterns is often hard to derive. Furthermore, many different temporal operators and functionalities must be provided to meet practical requirements, including, data mining functionalities, aggregation operators, or integrated correlation functions for streams.

On the other hand, these functionalities also require easy user access by the used query language with abstractions of the raw sensor values to the user on a higher level. Additionally, each operator (e.g. aggregation operators and grouping) must also be available for time series description.

Thus, we can see the requirements for ontology based stream querying language from two perspectives.

The first perspective is about characterizing and classifying sensor data, which has to be realized by an ontology based layer that enriches the sensor data with semantic information. The underlying stream query language should allow diagnosis and classifications of complex time sequences using input streams as well as historical or static data.

And second, from the data access point of view, we have to guarantee that the DSMS can handle continuous queries on multiple streams on behalf of abstract ontology models. Thus, considering the large number of DSMSs, CEPs and the heterogeneity of access methods they provide, it is necessary to evolve methods for querying these systems under an ontology based abstraction layer.

Therefore, an appropriate query language as well as corresponding query rewriting techniques from the proposed stream query language to the underlying DSMS have to be developed.

Functionality Requirements

We have already mentioned one important requirement for organizing time structures, additionally there are several more important things to mention. In the following, we collect requirements inspired from the use case described above or other stream querying languages and split them into two groups (inspired by CQL [17]) with respect to an access on different sources (i.e., R-to-S and S-to-R operators) or direct stream operators on relations (e.g., aggregation operators).

Window-to-Stream / Stream-to-Window Mapping Requirements: Input and output describes the operators for a temporal access on different data sources. In that context the following goals should be achieved by the language to be developed.

Sliding Windows. We require the access to time-tagged data with sliding windows, which should at least provide the use of parameters for defining window borders and its movement over the stream.

Live/Historic Data. We require access to live data streams as well as recorded time series. In the industrial use case we observe that live streaming data is important on the one hand, but also combined with archived historical streaming data, which can be used for predictive use cases, data mining and machine learning or correlation of a current data stream, on the other hand.

Static Data. We require access to static tables for merging temporal with non temporal data. The example includes static data sets, describing components and types of sensors. This data is non time-tagged and additionally stored in tables. It should be available and merged with live or historic data for querying.

Multiple Streams. We require access to multiple streams, which are joined for each window. This should be possible for historically recorded time series as well as real time inputs with various numbers of streams.

Cascaded Streams We require orthogonality for combining streams. With orthogonality we are able to build infrastructures of streams and use the output of one stream as input for another stream. For instance, we should be able to merge several STARQL output streams into one single stream.

Output Operator. We require at least the RStream operator for generating new stream output from each input window.

Output Synchronization. Input streams might be based on different window slides and therefore, we have to provide means for generating or synchronizing the output with a defined sampling rate.

Requirements for Stream Operators: For direct operations on streams, we define the following operator requirements.

Basic Query Operators. We require basic functionalities on streams, such as joins, unions, filtering and optional.

SPARQL 1.1 Operators. We require SPARQL 1.1 functionalities for data analytics and query formulation, such as aggregation functions, grouping, negation, arithmetic expressions and negation.

Sequence Operator. We require an operator for building temporal states and patterns as explained above. Additionally, an operator for accessing time-based sequences in an efficient way is mandatory.

Reasoning. Basic reasoning functionalities are required to solve problems such as subsumption of concepts in concept hierarchies.

3.1.3. A New Query Language for Streams?

In chapter two we have introduced several datastream management systems (Section 2.2) as well as RDF-Stream querying languages (Section 2.5). Although the stream reasoning community is working on a standard ontology base stream language called RSP, it is still work in progress and many problems are not solved, e.g., standardization, the handling of time, reasoning on streams and dealing with incomplete or noisy data.

So, as there are already a lot of stream languages that make use of the OBDA paradigm and at least partly fulfill the mentioned requirements, it is a justified question to ask: *"Why inventing a new one?"*.

Our examples above show that for industrial problems it is very important to describe and query the behavior of sensor data over time, especially w.r.t. what happens before or after certain events. This could be, for example, a sequence of events that has to be identified, which in general is accomplished by CEP systems. EP-SPARQL (Section 2.5.5) is an ontology based extension for ETALIS (a general CEP engine) and extends the well established SPARQL standard by two additional operators called SEQ and EQUALS, which can join two graph patterns, if they occur after each other (SEQ) or at the same time point (EQUALS). A problem is that EP-SPARQL completely lacks window operator, but they can be simulated by additional time constraints.

Other languages, not specifically designed for event processing, lack the SEQ operator. Thus, a sequence of events cannot be expressed with languages such as CQELS (Section 2.5.3) and C-SPARQL (Section 2.5.2), only reference a timestamp for each triple by additional functions, but are not able to declaratively define state sequences in an appropriate way. A reason for limited sequence support of these languages can be seen in their status as an extension to the static query language SPARQL, where every input has the same timestamp and thus, cannot be arranged in a temporal sequence. There are basically three disadvantages of this solution.

The semantics presupposes mixed interim states in which the constraints and consequences of the ontologies (in particular inconsistencies) are not considered.

Second, these solutions do no longer adhere to the requirements of an orthogonal query language, where the inputs and interim-outputs are structures of the same categories.

And finally, in the case of knowledge bases that allow for the formulation of consistency assumptions, one has to keep track of timepoints in the window operators as they may lead to consequences at later timepoints. For example, if a sensor is broken at a previous timepoint, it stays broken at future timepoints. One solution to this problem was already introduced in TEF-SPARQL (see Section 2.5.6), by using so-called data facts.

Our conclusion is that indeed data stream management systems exist, which can be used as instruments for operating on raw streams, but for practical and industrial use cases on stream querying with respect to ontologies no appropriate language exists up to now.

Therefore, we propose a query language that is not only an extension of SPARQL, but a new language that involves features from SPARQL, in addition to capabilities for managing sequences of timepoints with specific operators. We formulate resulting research problems and hypotheses for this work below and introduce our query language called STARQL[3] by examples and continue with its detailed syntax and semantics.

3.1.4. Resulting Problems and Hypotheses of this Work

As discussed above, some of the requirements are fulfilled by the previously discussed streaming extensions. Nevertheless, none of them is able to fulfill all of them or activate full ontology access for timeseries as they rely for ontology based data access on the rewriting of a single ABox in the standard SPARQL (Section 2.3.3).

Thus, we propose a new ontology based stream query language with rewriting algorithms to complete the picture of querying RDF streams.

From analyzing the measurement scenario as well as state of the art query languages in previous sections, we have identified the following three research problems:

P1. How can we enable full ontology usage for temporal sequences, reasoning over time, semantic enrichment and access to streaming or historical data, while keeping all operators and functionalities for RDF data provided by SPARQL?

P2. Are rewriting techniques to different (relational) backend stream sources still feasible if we fulfill the listed requirements for time series analysis?

[3][S]treaming and [T]emporal ontology [A]ccess with a [R]easoning-based [Q]uery [L]anguage

P3. Is query processing still efficient enough for handling large input streams or temporal/static data together in possible big data scenarios?

The goal of this work is to give answers to these research problems. In connection to these problems we also formulate a more detailed list of hypotheses, which are going to be verified in the following chapters.

H1. We allow the basic functionalities of SPARQL, as well as the operators from SPARQL 1.1 in our query language.

H2. We guarantee stream access on different input data, namely, real-time, historical and static data.

H3. Different kinds of streams (real-time and historic) can be joined, and the output corresponding to a query needs to be synchronized, if different slide parameters are provided for respective windows.

H4. We allow basic reasoning in the range of DL-Lite$_{\mathcal{A}}$ and semantic enrichment for STARQL.

H5. We are able to handle an OBDA approach, while using standard ontologies such as the SSN ontology.

H6. We are able to rewrite the temporal sequences and operations without any loss of efficiency on the backend system.

H7. We are able to translate the continuous relational query results of high throughput streams without delays (or at least not relevant delay).

H8. Our approach can be used in parallel and horizontally scaled on different machines for use in Big Data scenarios with modern technologies and data bases as backend.

In the following we will present our new stream query language STARQL[4] [181] and describe its syntax and semantics by example. We will explain how a rewriting to relational algebra is possible and show comparisons to other stream query languages. In the following chapters we will then provide an inside view on query transformation techniques and a proof-of-concept implementation with evaluations for several backend systems.

[4][S]treaming and [T]emporal ontology [A]ccess with a [R]easoning-based [Q]uery [L]anguage

3.2. Introduction to STARQL

We will introduce the features of our streaming query language step by step according to examples from sensor measurement scenarios and the requirements mentioned above.

Now let us first 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(pronounced Star-Q-L) on an abstract logical level, thereby assuming that the data in databases and relational streams have already been mapped to the logical level static or temporal ABoxes, as well as ABox assertion streams.

STARQL combines elements from SQL, SPARQL and descriptions logics. Instead of the logical notation of DLs we use the machine processable turtle notation of SPARQL [232]. Moreover, by partially using SPARQL we can also rely on its namespace handling mechanism (though we will skip most of the namespace declarations for keeping the examples short).

The syntax of STARQL extends so-called *basic graph patterns* of the W3C standardized SPARQL query language for RDF databases and describing their time-based relations. Its queries can express basic graph pattern relations, and typical mathematical, statistical and event pattern features needed in real-time diagnostic scenarios by relying on backend computational capabilities through an *OBDA* approach.

3.2.1. Introduction of STARQL by Example

We give an overview on STARQL's main syntactical features below to start with a basic example and get into the detailed features afterwards.

`CREATE STREAM`: A continuous query refers to one or more streams and produces a stream again. The *create* statement indicates the creation of a new substream connected to an identifier (requires a `CONSTRUCT` query form). Moreover, by referencing stream identifiers, STARQL queries can be nested, in the sense that the result of one substream may be used as input to another one.

`SELECT/CONSTRUCT`: The output of a STARQL query can be defined through several different query forms. It can either be a `SELECT` query that provides answer sets that are given by variable binding lists or a `CONSTRUCT` query

where the answers are defined as a new RDF Graph that can either be stored in an RDF Dataset or sent as input to another STARQL stream query.

`FROM:` In this clause the input to the query is defined. Inputs can be streams (defined with window parameters), static ABox data or an TBox ontology.

`USING:` References the periodic pulse for the specific substream, given by an execution frequency and its absolute starting time.

`SEQUENCE BY:` References the periodic pulse for the specific substream, given by an execution frequency and its absolute starting time.

`WHERE:` Inspired by the SPARQL standard for querying static RDF data, the `WHERE` clause uses graph pattern matching for identifying possible variable bindings in the static data (included in the `FROM` clause).

`HAVING:` As mentioned above, basic graph pattern matching is extended for time based relations over all input streams in the `HAVING` clause. Each temporal graph is identified by an index indicating its point in time. The temporal patterns are combined with filter conditions to a safe first order logic formula.

`GROUP BY/AGGREGATE:` The aggregation operator consists of two clauses: a grouping can sort the result list based on one or more free variables in the query and finally the `Aggregation` clause calculates results by an aggregation function for each group. Thereby STARQL is not bound to specific functions such as AVG, MAX, MIN, SUM, COUNT, it even supports multi-dimensional aggregations such as correlation functions.

For the sake of the following example let us first assume that the terminological TBox is empty.

An engineer may be interested in whether the temperature measured by the sensor s_0 exceeds a certain threshold in the last two minutes, i.e., in the interval $[NOW - 2M, NOW]$ (including borders). We first present the solution in our streaming language STARQL to give an impression of the language and cover its details afterwards.

Listing 3.1: Basic STARQL example 1

```
1  CREATE STREAM S_out_critical AS
2
3  CONSTRUCT GRAPH NOW { ?sens rdf:type :Critical }
4  FROM measurements [NOW - 2m, NOW]->1s
5  SEQUENCE BY StdSeq AS stateSequence
6  HAVING EXISTS ?i IN stateSequence(
7  GRAPH ?i {?sens :hasVal ?x} AND ?x > 90 )
```

The solution of the threshold example is shown in Listing 3.1. It is a simple example,

which registers a stream called $S_{out_critical}$ in the streaming system. The `CONSTRUCT` clause defines its output, which is a temporal graph that defines a sensor s_0 being of the type *Critical* at timestamp *NOW*, where the `HAVING` clause is true. The input stream is called *measurements* and has a sliding window with a window size of three minutes (borders included) and a slide of one second as declared in the FROM clause. We use first order logic to describe the behavior of a time sequence in general and define its content in the so called `HAVING` clause. Here, we use a variable x for referring to sensor values and enforce that one of these values appears in some time point of the sequence and is higher than $90°C$.

Inconsistency

In the case of sensor measurement scenarios, such as the described above, we often have to deal with unclean data that can lead to inconsistent or indefinite knowledge. Regarding the scenario, we may have the following problem of one sensor having more than one value at the same time. For instance, there may be two values for the sensor s_0 at time point $1s$.

$$\begin{aligned} S_{Msmt}^{\leq 1s} &= \{\ hasVal(s_0, 90°C)\langle 1s\rangle, \\ &\quad\ \ hasVal(s_0, 93°C)\langle 1s\rangle\ \} \end{aligned}$$

This situation, where a sensor has different values at a time, can happen in practical applications because of delayed streams or data providers and an unexpected sensor setup.

We could solve this problem by directly manipulating the sensor data stream of the DSMS, but as we follow the idea of ontology based data access, we do not want to touch the data directly. In our case we would define mappings to the data stream that include the cleaning. One possible mapping could select all measurements for a given sensor at the same time point calculating the respective mean value. The outcome describes a sensor value as shown below.

$hasVal(x, y)\langle z\rangle \longleftarrow$
```
SELECT f(SID) AS x, AVG(val) AS y, timestamp AS z
FROM measurement GROUP BY SID, timestamp
```

On the other hand, a drawback of this mapping-centered approach is the fact that a user might see the mappings as a black box without having access to them. Hence, a diagnostic engineer for example should have his own way for dealing with multiple

sensor values. Our example for *threshold* sensor values already solves this problem as it only requires that one value is required to exist, without mentioning the number of values per timepoint. But we could equally use an aggregation function for each timepoint in STARQL for immitating the mapping (see our requirements for aggregation functions), where we require the windows to be small, as too many values could make the aggregations expensive, thus slowing down the system.

A second possible problem with respect to sensor values is that there might be no received value at all for a given point in time from the measurement stream, although a value is expected. For the open world assumption of the OBDA view this means that we simply do not know whether a value actually exists or not. Without mentioning a possible TBox that could tell us about existing values, our *threshold* example does not incorporate the assumption of unknown values. We solve the problem of unknown values by a told value approach, considering only those cases where actually a concrete value exists as input.

Not only the raw data itself can lead to inconsistencies. Even in the case of lightweight description logics such as DL-Lite, the TBox can contain constraints that lead to inconsistencies with ABox assertions too. In the sensor measurement scenarios 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:

$$(\text{func } hasVal)$$

It states that at every time point there can be at most one filler of the role $hasVal$ in a particular state. we assume that the TBox holds 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 knowledge base becomes inconsistent.

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

$$Sensor \sqsubseteq \exists hasVal$$

If there is a missing value for a sensor at some temporal state 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, we could also think of a notion of 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 [145].

Hence, in our query language the aggregation operators will work only with told-values similar to the epistemic approach of [65]. So, for all knowledge bases KB_i in a possible 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 $hasTempVal(s_0, v_0)$.

Orthogonality

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. It is in general realized similarly to the `CONSTRUCT` operator from the SPARQL query language, which provides the means to define the format in which the bindings of the variables should be generated.

The approach of stream topologies is motivated by the idea that query outcomes are going to be used as inputs to other queries as well as the generation of (temporal) ABox assertions in the application scenario itself. The produced ABox assertions hold only in each window of the output stream in which they are generated—and not universally.

Otherwise the generated assertions would also hold in the input stream again and require recursion in queries, which might lead to performance issues due to a theoretically high complexity of the query answering problem. Though the ABox assertions are limited to hold in the output streams, they may interact with the TBox, leading to entailed assertions.

An engineer could reuse results of other streams. For example he could define a continuous query that filters temperature sensors, which show a temperature value higher than 90 degrees. Such a query is shown in Listing 3.2. It uses the prior *threshold* example as input (see Listing 3.1), adds a filter constraint for temperature sensors and produces new ABox assertions defining critical temperature sensors at the output.

Listing 3.2: Basic STARQL example 2

```
1   CREATE STREAM S_out_criticalTemp AS
2
3   CONSTRUCT GRAPH NOW { ?sens rdf:type :CriticalTemp }
4   FROM S_out_critical [NOW - 1s, NOW]->1s
5   WHERE {?sens a :TempSensor}
```

The query is evaluated on the stream $S_{out_critical}$, which contains assertions of the form $Critical(sens)\langle t\rangle$. At every second only the current assertion is put into the temporal ABox (window range = 1s) so that the sequence contains only a trivial sequence of the past one minute (at most). The example might use oversimplification but the reader should be able to understand the main idea.

After discussing this example we will now go deeper into the operators of the query language.

3.2.2. STARQL Stream Operators

According to the previous example we will now discuss the specific operators provided by STARQL. The operators can be split into three categories inspired by CQL see Section 2.2.6). We call them *Stream to Window*, *Window to Window* and *Window to Stream* Operators.

The first category describes operators that transform raw infinite stream data into smaller parts or window structures, which can be queried with different STARQL operators afterwards. The succeeding two categories operate on window structures and finally put each result into the output stream. We start with the former category and describe the generation of window structures and sequences in STARQL.

Figure 3.1.: Example data for a measurement input stream

Time	Temporal ABox
$0s$	$\{hasVal(s_0, 90°C)\langle 0s\rangle\}$
$1s$	$\{hasVal(s_0, 90°C)\langle 0s\rangle, hasVal(s_0, 93°C)\langle 1s\rangle\}$
$2s$	$\{hasVal(s_0, 90°C)\langle 0s\rangle, hasVal(s_0, 93°C)\langle 1s\rangle, hasVal(s_0, 94°C)\langle 2s\rangle\}$
$3s$	$\{hasVal(s_0, 93°C)\langle 1s\rangle, hasVal(s_0, 94°C)\langle 2s\rangle, hasVal(s_0, 92°C)\langle 3s\rangle\}$

Stream to Window - Windowing

The presented example is centered around an input stream and window parameters declared in the `FROM` clause.

Listing 3.3: Window operator from basic STARQL example 1

```
1   FROM S_Msmt [NOW-2m, NOW]->1s
```

The window of the example stream is of range (width) two seconds (see Listing 3.3). Every second (see the slide parameter as denoted above by `->1s`) the NOW moves forward in time and gathers all timestamped assertions in each step whose timestamp lies in the interval `[NOW-2s, NOW]`. Here, `NOW` denotes the current time point. We will follow in our exposition this synchronized approach, where the window moves forward by a query defined time value (i.e. slide parameter). Thus, we actually have a window with content for each step, synchronized by the sliding parameter. For example, this means for our example exactly one evaluation window per second.

Each window content can be seen as a set consisting of timestamped assertions, which together make up a temporal ABox. Assuming that every stream must start somewhere at timepoint 0, there only exists one timepoint in the first window. As time goes on, the time window fills up. So within this example for the first two time points $0s, 1s$ we do not get well defined intervals for `[Now-2s, Now]`, as we do not have information about the three past seconds, but it is natural to declare the contents at $0s$ and $1s$ as the set of timestamped ABox assertions which have arrived up to the current timepoint. For the other time points, we can assume to have information on the complete interval, and so the resulting temporal ABoxes from $0s$ to $2s$ are defined as given in Figure 3.1.

Our example describes at each timepoint a set of timestamped assertions for the content of the specific window, varying in numbers from one to three timepoints.

For each point in time this set of assertions, produced by the window operator, is stored into an ABox. As discussed before (Section 2.3), the classical reasoning is established over single ABoxes only with time-tagged assertions, containing no more than one timestamp for consistency reasons. Therefore, in STARQL we would like to consider reasoning over more than one timestamp or one ABox respectively to analyze streaming data.

So, in order to apply ABox and TBox reasoning on streams, we build an ordered group of ABoxes representing points in time, where assertions are grouped together into the same (pure) ABox for each timepoint. The generation of those sequences is managed by the STARQL sequence operator.

Stream to Window - Sequencing

The result of the grouping is a finite sequence of ABoxes (ABoxes are sequenced with respect to the order of their timestamps). By splitting the axioms of the window operator into temporally related groups, we can prevent problems regarding inconsistency and indefiniteness of values especially regarding functional roles and relations (see Section 3.2.1). This sequencing can be provided by appropriate means of the sequencing operator, such that each pure ABox is consistent in itself. The operator provides different sequencing strategies. In the following example we present a standard strategy, where assertions are directly grouped together according to their identical timestamps.

Listing 3.4: Sequence operator from Example 1

```
1   SEQUENCE BY StdSeq AS stateSequence
```

The ABox sequencing operation is introduced by the keyword `SEQUENCE BY` in the query fragment (see Listing 3.4), followed by the type of sequence and a reference name.

There may be different methods to build the sequence, but the most natural one is to merge all assertions with the same timestamp occurring in a window into the same ABox. The sequence operator shown in Listing 3.4 refers to this built-in sequencing method called `StdSeq` for standard sequence. Other sequencing methods may be defined by following an SQL-like create declaration (for details see the next sections).

We can use this sequence of classical ABoxes to filter those values v for which it is provable that $hasValue(s_0, v)$ “holds” for our sequence of the threshold example (given in Listing 3.1).

Just for illustration, we note that the ABox sequence at time $3s$ is defined by the input data from Figure 3.1.

In our simple example, the standard ABox sequencing leads to simple ABoxes with one assertion each, as the stream does not contain ABox assertions that have the same timestamp. Please note that each ABox in a sequence can be combined with larger static ABoxes (see below).

The sequence contains timestamped ABoxes, e.g., the first ABox contains the triple $\{hasVal(s_0, 93°C)\}$ and timestamp $\langle 1s \rangle$. STARQL itself does not refer to the timestamps of the ABoxes in the sequence, but to a natural number indicating its ordinal position within the abstracted sequence. Hence, the actual sequence at time point $3s$ can be written as follows:

Time	**ABox sequence**
$3s$	$\{hasVal(s_0, 93°C)\}\langle 1 \rangle, \{hasVal(s_0, 94°C)\}\langle 2 \rangle, \{hasVal(s_0, 92°C)\}\langle 3 \rangle$

Stream to Window - Multiple Streams

Many interesting time series features require the input from several different stream sources. As shown above, the sequence for a single input stream can be implemented in a straightforward way according to a standard strategy, but in the case of multiple input streams the sequence states have to be coordinated with respect to the input stream content.

In the sensor measurement scenario this could be the case for values from different measurement streams or the join of measurement and event streams in order to detect correlations and do further data analytics.

The simplest combination of streams is a union, which can be directly provided with the same data schema as in pure SQL. For a multi stream example in STARQL, we consider the join of sensor values.

So, if one had two different measurement input streams and one was interested in the overall maximum value for each time window, then the STARQL `FROM` clause could be formulated as given in Listing 3.5.

Listing 3.5: STARQL example for combining multiple streams

```
FROM  S_Msmt_1 [NOW-2s, NOW]->1s,
      S_Msmt_2 [NOW-2s, NOW]->1s
SEQUENCE BY StdSeq AS stateSequence
```

Here, we define two sliding windows with parameters for two different input streams (for this example all window parameters are identical) and combine both to evaluate the overall maximum value. The semantics of the stream union is given by pure set union of the resulting temporal ABoxes generated by the window operator and no sequencing. In many cases, the simple union of the ABoxes is not an appropriate means, because the timestamps of the streams might not be synchronized for all input streams, which are not comparable if one stream is delayed. And thus, the simple idea of building a state sequence based on identical timestamp may lead to unintended results.

Consider a temperature sensor and a pressure sensor, which are expected to show some value regularly each minute, but the emitted temperature value is several seconds delayed, we see that a sequence operator that arranges assertions into the same state based on identical timestamps (as in the standard sequence) is no longer sufficient. Thus the engineer may have an interest in putting also the delayed assertions with identical timestamps of both streams into the same temporal sate: Because, only if the values of both sensors fulfill some conditions "at the same time" (read as "in the same ABox"), will the engineer be able to infer additional knowledge for this query. In this case 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 temperature value of sensor s_0 is bigger than the temperature value of s_1 at the same time, then the turbine is considered to be in a critical mode as formulated below.

$$\begin{aligned}
&\mathit{hasVal}(s_0, v_1) \wedge \mathit{attachedAt}(s_0, \mathit{turb}) \wedge \\
&\mathit{hasVal}(s_1, v_2) \wedge \mathit{attachedAt}(s_1, \mathit{turb}) \wedge \\
&\qquad\qquad\qquad v_2 > v_1 \rightarrow \mathit{crit}(\mathit{turb})
\end{aligned}$$

Hence, the query language is envisioned to be used for such scenarios, it should provide means to define a sequencing strategy based on some temporal 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 a rougher or finer granularity. So, in the example above, the idea could be to define a sequence, where each state has the granularity of one minute instead of one second, such that a value at timepoint one minute appears as “at the same time” as values between second one and second sixty.

Our query language gives the flexibility to implement different merges (sequences). New sequences can be defined by the `CREATE SEQUENCER` command as demonstrated in Listing 3.6

Listing 3.6: Example for coarsening

```
1   CREATE SEQUENCER seq_min AS
2   GRANULARITY = 1m
```

Sequencers are defined by a granularity parameter, which is set to 1 minute for the example and merges assertions in a time interval of one minute into the same point in time. Merging timestamps into one timepoint can have several disadvantages or inconsistencies, especially for functional roles as mentioned before. The sequencing operator therefore picks only the latest functional role assertional to be merged for one timepoint automatically.

In reality there might be more advanced requirements for the use of functional roles. An engineer could be interested to merge only the average of several sensor values into each time point. Here, we can make use of the orthogonality factor explained in the orthogonality section, which means that we are able declare a pseudo sequencer by adding another STARQL query stream as input to the actual STARQL query. The sub query would then evaluate the average values for each state and provide input to the actual query. This input could be synchronized to other input streams and merged into the sequence.

Window to Window - Adding Static Data and Additional Knowledge

Until now, in all previous examples for STARQL we focused on temporal ABoxes only, and did not refer to entailment w.r.tṪBox knowledge. In this section, we demonstrate the effects of TBoxes and their use with a small example.

Considering our last example, where a user was interested in finding monotonically increasing sensor values, we are now only interested in a subclass of sensors for the evaluation, namely the subclass of temperature sensors.

We assume that there is a data source (i.e., the dataset presented in the last section) with information on sensors types, declaring specific sensors as *BurnerTipTempSensor* (i.e., temperature sensor located at the burner tip of a turbine).

Information on sensor location is static (does not change over time) and could for example be saved in a standard SQL data base table. Additionally, let us assume that there is a mapping for burner tip located temperature sensors that maps the data to ABox assertions. The assertions contain facts that hold at every time point and one assertion for our sensor example, namely $BurnerTipTempSens(s_0)$.

Now we would like to combine the static information with the time tagged sequence of the sensor values stored in several ABoxes for querying. In STARQL this is possible by adding the static dataset in one line to the `FROM` clause of the previous examples, but this would only allow us to query for specific *BurnerTipSensors*.

By adding a TBox, we can make use of additional knowledge. Here, the user may state the relationship between two concepts by a subsumption relationship. As described in the section on the data model, we assume that the TBox contains the following axioms $BurnerTipTempSensor \sqsubseteq TempSensor$, $TempSensor \sqsubseteq Sensor$. We can use the TBox to model the relationship between the concept names $BurnerTipTempSensor$ and $TempSensor$.

Now, we formulate the query by extending our previous threshold example with static information (see Listing 3.7).

Listing 3.7: STARQL example with static information

```
1   CREATE STREAM S_out_static AS
2
3   SELECT ?sens
4   FROM S_Msmt [NOW-2s, NOW]->1s,
5      STATIC ABOX <http://example.com/Astatic>,
6      TBOX <http://example.com/TBox>
7   WHERE { ?sens rdf:type :TempSensor }
8   SEQUENCE BY StdSeq AS stateSequence
9   HAVING EXISTS ?i, ?x in stateSequence: GRAPH ?i { ?sens :hasVal ?x }
```

We see the addition of the static ABox and TBox within the extended `FROM` clause. Although the static information is now included in the dataset, it is not queried yet. While querying time tagged data in the `HAVING` clause, we use a separate clause for querying static data with STARQL.

Within the `WHERE` clause we can specify relevant conditions. Here it is just the condition `TempSensor(sens)` that requires temperature sensors. The evaluation of the conditions in the `WHERE` clause as well as the `HAVING` clause considers not only

the temporal ABoxes in the generated sequence, but also the TBox and the static ABox.

In order to identify the sensor s_0 in the input stream as a temperature sensor, one has to exploit the fact from the static ABox stating that s_0 is a burner tip temperature sensor, and one exploits the respective axiom from the TBox, in order to conclude that s_0 is a temperature sensor.

So, for $ABox_i$ at position i in the time sequence, the relevant local knowledge base is given by $KB_i = TBox \cup ABox_{static} \cup ABox_i$.

Window to Window - Aggregation and Group By

Listing 3.8 shows an example with a more complex condition in the `HAVING` clause using grouping and the average aggregator. We are interested in those temperature sensors with a recent average value higher than 99 degrees in the last 4 hours. Furthermore, we would like to retrieve the average of the sensor values in that case. This can be realized by adding two further clauses, the `GROUP BY` clause, which groups all sensors, and the `HAVING AGGREGATE` clause, which calculates the average value for each group using the aggregation operator.

Listing 3.8: STARQL example for an aggregation operator

```
1  CREATE STREAM S_out_4 AS
2
3  SELECT ?sens AVG(?x) AS ?avg
4  FROM STREAM S_Msmt 0s<-[NOW-3h, NOW]->1s,
5  STATIC ABOX <http://optique.project.ifi.uio.no/Astatic>
6  WHERE { ?sens rdf:type :TempSensor }
7  SEQUENCE BY StdSeq as stateSequence
8  HAVING EXISTS ?i, ?x in stateSequence:GRAPH ?i { ?sens :hasVal ?x }
9  GROUP BY ?sens
10 HAVING AGGREGATE AVG(?x) > 99
```

We see that average value is selected by the output with its respective sensors and bound to a variable called *avg*. In this manner arbitrary aggregation functions can be used, if they are provided by the backend system. In the case of the Exareme system, STARQL even provides multi column operators such as the Pearson correlation function, which calculates the correlation between sequences of two variable bindings in a single window (see Listing 3.12 for an example).

Window to Window - Operating on State Sequences

Now, as there is a sequence of ABoxes generated for each window, at each time point one can refer to every (pure) temporal ABox by state variables in order to apply DL reasoning. This is actually done in STARQL (and our basic query example) within the `HAVING` clause (see Listing 3.9). The `HAVING` clause is a boolean expression. In general, it may be some predicate logical formula with open variables. (For the details see Section 3.3.2).

In the `HAVING` clause we make use of the graph pattern matching used in SPARQL by a notation of 'GRAPH i {[GP]}' and apply these patterns on each ABox of the sequence and bind the state numbers to index variable i wherever the specific graph pattern `GP` holds. Each identified state number is bound in a binding list to variable `i`.

We consider the following example in Listing 3.9 and make use of relations between different states of the generated sequence by referencing temporal states with index variables.

Listing 3.9: Basic STARQL HAVING example

```
1   CREATE STREAM S_out_moninc AS
2
3   CONSTRUCT GRAPH NOW { s_0 rdf:type :RecentMonInc }
4   FROM S_Msmt [NOW-2m, NOW]->1s
5   SEQUENCE BY StdSeq AS SEQ1
6   HAVING FORALL ?i,?j IN SEQ1,?x,?y (
7      IF GRAPH ?i { s_0 :hasVal ?x }
8         AND GRAPH ?j { s_0  :hasVal ?y }
9         AND ?i < ?j
10     THEN ?x <= ?y )
```

The example queries for timepoints, in which values of a sensor s_0 show monotonically increasing behavior for the specified three minutes time window. Therefore, the declaration RecentMonInc(s_0) is evaluated to true each time the HAVING clause holds for the following FOL formula:

$$\forall i, j, x, y((hasVal(i, s_0, x) \wedge hasVal(j, s_0, y) \wedge i < j) \rightarrow (x \leq y)) \quad (3.1)$$

Saying that for any pair of ABoxes in the sequence, where sensor s_0 shows a value, the value of the first state must be lower or equal to the value in the second one.

This formula is represented by the `HAVING` clause of Listing 3.9. We use a further condition in the formula using the `FORALL` operator. It evaluates the boolean condition in its scope on all ABoxes of the sequence and outputs the truth value *true*, if the condition for an operational mode holds in all ABoxes. Further, variables $?i$ and $?j$ are timepoints of time-tagged graph patterns, while the variables $?x$ and $?y$ are variables for the specific sensor values at the given timepoint.

The evaluation of the `HAVING` expression in Listing 3.9 is based on DL deduction: Find all x that can be proven to be a filler of $hasValue$ for s_0 w.r.t. the ABox $ABox_i$. A TBox and other (static) ABoxes used for deduction can also be mentioned in the `FROM` clause (see below).

Considering the example data from Figure 3.1 at 3s, the output stream for the monotonic increase example in Listing 3.9 is defined as follows:

$$S_{out}^{\leq 3s} = \{RecMonInc(s_0)\langle 0s\rangle, RecMonInc(s_0)\langle 1s\rangle, RecMonInc(s_0)\langle 2s\rangle\}$$

So, the query correctly generates assertions saying that there were recent monotonic increases according to the query for time points $0s$, $1s$, and $2s$.

Window to Stream - Synchronizing Output Streams by a Pulse

In the previous section we discussed multiple input streams using the same window parameters slide and width for each stream (see Listing 3.5). Using identical slide parameter means that the input streams are automatically synchronized and windows with the same windowId are automatically matched, considering that for each slide exactly one windowId is collected for all streams and one result graph generated for the query. Thus, mentioning continuous queries, we can say that the continuous output is generated by a frequency similar to the window slide, which is identical for all input streams in this case.

With STARQL we can also formulate more advanced queries using different window parameters for several input streams. An especially interesting feature it provides is the ability to use synchronized output pulses, formulating the frequency of the generated streaming output without taking into account the different frequencies of arriving input streams.

Listing 3.10: Example for pulse definition

```
1  CREATE PULSE examplePulse AS
2  frequency = 1m
```

One simple example for defining a pulse declaration by a reference name is shown in Listing 3.10. The pulse declaration is generally given by a *frequency* value, followed by optional *start* and *end* parameters.

The example query from Listing 3.11 uses the declared pulse *examplePulse* from Figure 3.10 for the output and checks if the average value of *sensorA* from stream S_{Msmt} (updated every minute) has been higher in the last 10 minutes than the maximum of *sensorB* from stream S_{Msmt2} (updated each 24 hours) in the complete last day.

Listing 3.11: STARQL example for advanced multi streams

```
1  CREATE STREAM S_out_pulse AS
2  SELECT ?x
3  FROM S_Msmt [NOW-9m, NOW]->1m
4  FROM S_Msmt2 [NOW-24h, NOW]->1d
5  USING PULSE examplePulse
6  SEQUENCE BY StdSeq as stateSequence
7  HAVING EXISTS i IN stateSequence (
8      { :sensA :hasVal ?x . :sensB :hasVal ?y } )
9  AGGREGATE AVG(?x) > MAX(?y)
```

We see two different sliding window parameters for the two input streams, one with slide parameter one minute and another definition with the slide parameter of one day. In this case the window generation for the input streams is no longer synchronized and therefore it could no longer be automatically equivalent to the output frequency of the input stream slide parameter. In STARQL we therefore have introduced a pulse frequency parameter controlling the global output frequency, which can be formulated in the query as shown in Listing 3.10.

In Listing 3.10 we define the pulse frequency to one minute for a pulse named *examplePulse*. A pulse can also be referenced in each declared stream by the statement *USING PULSE <name>* (see Listing 3.11).

Additionally to the declaration of a pulse frequency, someone could also be interested in using a start or end time in the pulse signal. These parameters of the signal are also specified in the pulse declaration. Start and end parameters are used with respect to historical queries, when, for example, one evaluates data from the beginning of the past year to its end. Thus, we distinguish streams in three different categories regarding the pulse.

Live Stream. In previous examples we showed how to query live streams with continuous result sets. Live streams are the most common use of STARQL. The

backend query engine is connected to live input streams and answers queries regarding to the current input. In this case we do not need to specify any start or endpoint of the pulse and the pulse is always active.

Historical Stream. Another way of using streams in STARQL is to use a recorded stream and evaluate it against other recorded or live streams. This strategy is used in particular for sensor correlation tests, we have a recorded stream of data from one year ago and we know that certain interesting conditions hold for that particular streaming input, then we would like to know if the current live stream correlates to the recorded stream.

Therefore, we set a start parameter for the pulse condition to synchronize between live and recorded streams. Additionally we formulate a `lag` parameter connected to the recorded stream. The parameter is added in front of the window parameter as shown below (here $lag = 1$ Day, see line 9 of Listing 3.12). Having both parameters, we can set the evaluation startpoint of the live stream and automatically calculate the evaluation point of the recorded stream, having two pointers according to the different inputs. The full example is shown in Listing 3.12.

Listing 3.12: Example for comparison of live and recorded streams

```
1   CREATE PULSE historic WITH
2      START = "2015-11-21T00:00:00CET",
3      FREQUENCY = "PT1M"
4
5   CREATE STREAM OperationalModeStream AS
6   CONSTRUCT GRAPH NOW { ?sensor a :InNormalOperationalMode }
7   FROM STREAM
8      measurement [NOW-10S,NOW]->1S,
9      measurementHist 1D <-[NOW-10S,NOW ]-> 1S
10  USING PULSE historic
11  SEQUENCE BY StdSeq AS SEQ1
12  HAVING EXISTS i in SEQ1 (
13     GRAPH i { ?sensor :hasValue ?y . ?sensor :hasHistoricValue ?z } )
14  GROUP BY ?sensor
15  HAVING AGGREGATE correlationFactor(?y, ?z) > 0.75
```

The combination of live and historic data is only applicable on specific backends (e.g. on Exareme, see Section 5.3.2), where the architecture allows for a combination of stream and batch processing as also described in the Lambda Architecture (see Section 2.1.6).

Temporal Query. In the same way we can compare two historically recorded streams. If we compare a recorded stream with a live stream, the data is updated

each time a new tuple arrives at the streaming engine and evaluated with respect to the pulse frequency. But, if we have recorded data only, we do not need to wait for arriving tuples, thus, we can evaluate the query as fast as possible relying on the already recorded dataset. These queries are called *temporal queries*. They are marked by a pulse having a *start* and *end* time parameter from the past.

Window to Stream - Defining the Output

For defining the output window of streams there are basically three strategies based on the definitions of CQL, namely `ISTREAM`, `DSTREAM` and `RSTREAM` operator. While `ISTREAM` and `DSTREAM` only send parts of the result set to the output, namely those assertions that have been changed compared to results beforehand, the *RSTREAM* provides the complete resultset of the time window without dependencies on prior windows. As this appears to be the most intuitive operator for users and both other strategies could be directly derived from the *RSTREAM* operator, it is currently chosen as the only output parameter for STARQL.

The output pattern of STARQL can be directly defined in two ways. First, as we have formulated previously, with variable bindinglists for the example shown in Listing 3.13. The operator returns a list of bindings for each named variable directly. The star operator “*” is an abbreviation that selects all free variables in the query.

Listing 3.13: Example for SELECT operator

```
1   SELECT NOW ?var1 ?var2
```

The second output expression is the CONSTRUCT query form, which returns a single RDF graph specified by a graph pattern. The result is an RDF graph formed by taking each query solution in the solution sequence, substituting for the variables in the graph pattern and combining the triples into a single RDF graph by set union.

The graph pattern can contain triples with no variables (known as ground or explicit triples), and these also appear in the output RDF graph returned by the `CONSTRUCT` query form. In front of the graph template, a time identifier is used. This could also be an abstract *NOW* for using the current time or a directly defined timestamp. An example of a `CONSTRUCT` query form is shown in Listing 3.14.

Listing 3.14: Example for CONSTRUCT operator

```
1   CONSTRUCT GRAPH NOW { ?sensor a :InNormalOperationalMode }
```

The main advantage of using the CONSTRUCT query form is its orthogonality (see 3.2). As the output is defined as new time-tagged RDF triples, those can be directly used as an input source or stream and evaluated in another STARQL query. Thus, STARQL can also be defined in a sub query like matter for enhanced sequences.

3.3. Formal Syntax and Semantics

After having introduced the operators of STARQL, we will now show its formal syntax and semantics in detail.

3.3.1. General STARQL Syntax

A STARQL grammar is shown in Figure 3.2, except for details of the HAVING clause. The HAVING clause grammar uses more complex expressions that are required for query transformations w.r.t. backend systems and will be defined separately in the following section.

The grammar contains parameters OL and ECL that have to be specified in instantiations. There is the ontology language OL and the embedded condition language ECL. ECL is a query language referring to the signature of the ontology language. STARQL uses ECL conditions as atoms in its WHERE and HAVING clauses, defining graph patterns. To directly embed axioms of the ontology into the query language (see also Section 2.4.1), we have chosen an ECL that consists of unions of conjunctive queries (UCQs), which are commonly known to be domain independent [4] and FOL rewritable with respect to DL-Lite ontologies [61].

Every STARQL query consists of one or more CREATE clauses, which define either a stream, pulse or sequence declaration respectively. All of them can be referred to with a specific reference name within the STARQL query declaration, which uses streams as an input source, respectively the specific pulse or sequence for manipulating its output.

Pulse expressions are defined with a frequency and optionally (indicated by square brackets in the grammar) with start and end timestamps for historical queries (see Section 3.2.2). Additionally, the declaration of several sequence methods is possible by a granularity parameter (see Listing 3.6).

STARQLExp	$\longrightarrow$	[*prefixdeclarations*] *createExp*
createExp	$\longrightarrow$	`CREATE`
		(`STREAM` *streamName* `AS` *streamExp*
		| `SEQUENCE` *seqName* `AS` *seqExp*
		| `PULSE` *pulseName* `AS` *pulseExp*)
streamExp	$\longrightarrow$	(`CONSTRUCT GRAPH` *constrHead*($\vec{x}$) | `SELECT` *selHead*($\vec{x}$))
		`FROM STREAM` *listWinStreamExp*
		[`, STATIC (ABOX` | `TBOX)` *URI*]
		[`USING PULSE` *pulseName*]
		[`WHERE` *whereClause*($\vec{x}_{wcl}$)]
		`SEQUENCE BY` *seqName*
		`HAVING` *safeHavingClause*($\vec{x}_{wcl}, \vec{x}_{hcl}$)
		[`GROUP BY` $\vec{x}$]
		[`HAVING AGGREGATE` *aggregateClause*($\vec{x}$)]
pulseExp	$\longrightarrow$	*frequency* [`,` *start*] [`,` *end*]
seqExp	$\longrightarrow$	*seqMeth*
constrHead($\vec{x}$)	$\longrightarrow$	*timeExp* *ECL*($\vec{x}$)
selHead($\vec{x}$)	$\longrightarrow$	*ArithExp*($\vec{x}$) [*varBinding*($\vec{z}$)][`,` *selHead*($\vec{x}$)]
listWinStreamExp	$\longrightarrow$	*streamName windowExp* [`,` *listWinStreamExp*]
windowExp	$\longrightarrow$	[*lag*`<-`]`[`*timeExp*$_1$, *timeExp*$_2$`]->`*slide*
whereClause($\vec{x}$)	$\longrightarrow$	*ECL*($\vec{x}_{wcl}$)
seqMeth	$\longrightarrow$	`StdSeq` | *seqDef*
timeExp	$\longrightarrow$	`NOW` | `NOW -` *constant* | *timestamp*

Figure 3.2.: Simplified syntax for STARQL (OL, ECL)

For each create stream definition, the query head can be defined as either a graph pattern (see Section 3.2.2) or a list of variable bindings (see Listing 3.13) by the respective keywords `CONSTRUCT` or `SELECT`, which bind free variables to instantiate a graph pattern or produce a binding list.

We denote $\vec{x} = \vec{x}_{wcl} \cup \vec{x}_{hcl}$ as the combined set of all free variables for individuals or constants in the `WHERE` and `HAVING` clause.

Besides we define the input to the STARQL query in the stream declaration by the `FROM` statement, which is followed by some optional static data source (see also Listing 3.5). There are several optional elements for each stream declaration related to the evaluation of the input data. In the `WHERE` clause we can formulate conditions depending on static data, which becomes mandatory in all states that are defined by the sequence operator (see Section 3.2.2). The syntax of the where cause itself is given by an embedded conditional language (ECL).

As STARQL is designed to be a framework, we can embed different languages to define intra state conditions. Thus, for the scenario of ontology based data access we require the language to be transformable into other backend system domains. Furthermore, we extend the ECL by an `OPTIONAL` operator and thus, allow simple basic graph patterns from SPARQL (see Section 2.3.3) without additional filtering, which is not necessary as we allow global filtering for the complete `HAVING` clause.

A sequencing method (here `StdSeq`) maps an input stream to a sequence of ABoxes according to a grouping criterion defined after the `CREATE SEQUENCE` statement and can be accessed in the query by the `HAVING` clause (see next section).

The `safeHavingClause` as well as the `WHERE` clause share free variables (indicated in parentheses as $\vec{x}$), which can be referred to in the query head `SELECT` or `CONSTRUCT` clause.

A select or graph pattern output can be further defined by aggregators and groupings. It is either directly expressed in the `SELECT` clause and returned in the result binding list, or the aggregation is only used as a constraint in the `HAVING AGGREGATE` clause. These operators can also be bound to variable groupings, which are defined in the `GROUP BY` by a list of free variables (see Listing 3.8).

As mentioned above, safety criteria are required in the `HAVING` clause and will be discussed in detail throughout the next subsection.

3.3.2. STARQL HAVING Clause Syntax and Safety Criteria

As described in the last section, STARQL makes use of an embedded conditional language for intra state data evaluations. Since the ECL has no dependencies on time itself or respectively dependencies on the created temporal state sequence, it can be evaluated for each state separately.

Per state evaluation is used if we define graph patterns in the `WHERE` clause, which holds at any time of the sequence, or by defining a graph template in the `HAVING` clause, evaluated on each ABox state separately for generating the output.

On the other hand, we have also chosen to evaluate the STARQL state sequence for inter state or time based dependencies using an ABox sequence in the HAVING clause, where STARQL allows first order logic with references to states for specifying conditions on ABox sequences as shown for example in the monotonicity example (see Listing 3.9).

The semantics of a HAVING clause (as given in Listing 3.9) is based on a structure of interpretations $\mathcal{I}_t$ for each ABox $\mathcal{A}_t$, where the domain $\Delta^{\mathcal{I}_t}$ does not only consist of the individuals or value constants and the so called *active domain* according to database terminology as used in [4], but also of the whole set *Dom*, which also includes state indexes that are produced by the sequencing operator.

For transforming from STARQL to another for database query language it must be guaranteed that the evaluation of the `HAVING` clause and the ABox sequence is only evaluated on the actual active domain of the streaming data base. In that case we can say that the `HAVING` clause is domain independent (d.i. for short). To realize this domain independence, we introduce a safety mechanism for STARQL HAVING clauses.

Definition 9. ***Domain independence*** *A query q is domain independent, if and only if for two interpretations $\mathcal{I}_1, \mathcal{I}_2$, having two different domains $\Delta^{\mathcal{I}_1}, \Delta^{\mathcal{I}_2} \subseteq Dom$ and identical denotation functions $(\cdot)^{\mathcal{I}_1} = (\cdot)^{\mathcal{I}_2}$, the answers for q in interpretation $\mathcal{I}_1$ are the same as the answers for q in interpretation $\mathcal{I}_2$.*

Without any safety mechanism, a `HAVING` clause of the form $?y > 3$, where $?y$ is a free concrete domain variable, would be allowed in the query, but the result set of bindings for $?y$ would be infinite ($?y$ would match all real numbers bigger than 3). We can also say that the range of $?y$ is not restricted and therefore, $?y > 3$ cannot be domain independent because the filter condition does actually not depend on the active domain of the database in the case of the free variable $?y$ as it is required. On the other hand a formula $?y = 3$ evaluates y to a value of 3 only, and in this case we say that the formula is a *safe range* formula or *range restricted*.

A simple safety mechanism can be given by a direct binding of the concrete domain variable to values of the active domain in the query. Therefore, a graph pattern is added for each unrestricted domain variable in the `HAVING` clause, as, e.g., shown for the following `HAVING` fragment: *GRAPH ?i { ?sens :hasVal ?y } AND ?y > 3*.

While the query fragment is safe with respect to the active domain, the complete `HAVING` clause used in the specific query is not necessarily a safe range formula. It could include other unrestricted free variables or even consist of further fragments that make the variable $?y$ of the originally safe fragment unsafe again in the complete query, e.g., if we have disjunctions of a safe and an unsafe variably $?y$.

Thus, safety conditions ensure that all free variables of a STARQL query are also range restricted. A detailed definition of range restriction (*rr* for short) and domain independence can be found in [4].

The authors of [4] assume that the safe range formula is already in an advanced form that they call *safe range normal form*, but as we define the HAVING clause grammar in the context of a grammar for user-defined STARQL queries, we can not assume a *safe range normal form* in general. Additionally, we also have to take care of other free `WHERE` clause variables $\vec{x}_{wcl}$, which are not bound, but referenced in the `HAVING` clause. This leads to many sub-cases in our grammar rules.

One strategy for checking the safety criteria is to go through each subformula of the `HAVING` clause and connect each of its free variables to variable guards g_i, while declaring a specific status for each guard variable in each subformula and then combine the status of each subformula through operators by specific rules, which results in one safety status for each variable $\vec{z}$ in each HAVING clause.

The `HAVING` clause grammar is constructed by several rules shown in Figure 3.3. Its main rule (Eq. 3.2) $safeHavingClause(\vec{z}) \longrightarrow hCl(\vec{z}^{+})$ defines that every safe `HAVING` clause is a `HAVING` clause, where all free variables in $\vec{z}$ are safe (indicated by a plus at the right hand side of the rule). The further rules define safety conditions for free variables in each `HAVING` clause fragment. The lowest level of the `HAVING` clause fragments are constructed by state atoms and arithmetic expressions (see Eq. 3.4 to 3.6), which are combined to first order logic formulas. Each combination of atoms propagates or changes the safety status of the underlying variables in the complete formula. The safety status is expressed in so-called *adornments*.

Definition 10. ***Adornments*** *The adornments $\vec{g} = g_1, \ldots, g_n$ are defined as a list of guard status g_i (g-status for short) for the vector of free variables $\vec{z}$, where $g_i \in \{+, -_+, -, \emptyset\}$. We use $\vec{z}^{\vec{g}}$ as an abbreviation for $z_1^{g_1}, \ldots, z_n^{g_n}$ where $\vec{z} = z_1, \ldots, z_n$ and $\vec{g} = g_1, \ldots, g_n$.*

$$safeHavingClause(\vec{z}) \longrightarrow hCl(\vec{z}^{+}) \quad (3.2)$$

$$stateAtom(\vec{x}^{+}) \longrightarrow \texttt{GRAPH i } \mathrm{ECL}(\vec{x}) \quad (3.3)$$
$$arithEqAtom(x^{+}) \longrightarrow x = a \mid a = x \quad (\text{for } a \in \vec{x}_{wcl} \cup Const) \quad (3.4)$$
$$arithEqAtom(x^{-}) \longrightarrow x = a \mid a = x \quad (\text{for } a \notin \vec{x}_{wcl} \cup Const) \quad (3.5)$$
$$arithInEqAtom(x^{-}) \longrightarrow x \ op \ a \quad (\text{for } op \in \{\texttt{<,<=, >=, >}\}) \quad (3.6)$$
$$hCl(\vec{z}^{\vec{g}}) \longrightarrow stateAtom(\vec{z}^{\vec{g}}) \mid arith(In)EqAtom(\vec{z}^{\vec{g}}) \quad (3.7)$$
$$hCl(\vec{z}^{\vec{g}^1 \vee \vec{g}^2}) \longrightarrow hCl(\vec{z}^{\vec{g}^1}) \texttt{ OR } hCl(\vec{z}^{\vec{g}^2}) \quad (3.8)$$
$$hCl(\vec{z}^{\vec{g}^1 \wedge \vec{g}^2}) \longrightarrow hCl(\vec{z}^{\vec{g}^1}) \texttt{ AND } hCl(\vec{z}^{\vec{g}^2}) \quad (\text{both conjuncts are not arithEqAtoms}) \quad (3.9)$$
$$hCl(z_1^{g_{max}}, z_2^{g_{max}}, \vec{z_3}^{\vec{g_3}}) \longrightarrow hCl(z_1^{g_1}, z_2^{g_2}, \vec{z_3}^{\vec{g_3}}) \texttt{ AND } z_1^{g_1} = z_2^{g_2} \quad (\text{for } g_{max} = max\{g_1, g_2\}) \quad (3.10)$$
$$hCl(\vec{z}^{\neg\vec{g}}) \longrightarrow \texttt{NOT } hCl(\vec{z}^{\vec{g}}) \quad (3.11)$$
$$hCl(\vec{z}^{\vec{g}^1 \rightarrow \vec{g}^2}) \longrightarrow \texttt{IF } hCl(\vec{z}^{\vec{g}^1}) \texttt{ THEN } hCl(\vec{z}^{\vec{g}^2}) \quad (3.12)$$
$$hCl(\vec{z}^{\vec{g}^1 \wedge \vec{g}^2}) \longrightarrow \texttt{EXISTS } y \ hCl(\vec{z}^{\vec{g}^1}, y^{+}) \texttt{ AND } hCl(\vec{z}^{\vec{g}^2}, y^{g}) \quad (3.13)$$
$$hCl(\vec{z}^{\vec{g}^1 \rightarrow \vec{g}^2}) \longrightarrow \texttt{FORALL } y \texttt{ IF } hCl(\vec{z}^{\vec{g}^1}, \vec{y}^{+}) \texttt{ THEN } hCl(\vec{z}^{\vec{g}^2}, y^{g}) \quad (3.14)$$

Figure 3.3.: Grammar for STARQL `HAVING` clauses

Figure 3.3 contains the grammar of the complete `HAVING` clause with its safety mechanism for each fragment as described by the variable guards. Each rule is denoted with possible free variables and a resulting safety status shown in the rule head for the given subformula in the rule body.

We illustrate the meaning of the `HAVING` clause grammar with two example rules from the grammar, the safeHCL rule (see Rule 3.2) as well as Rule 3.8) for the `OR` case in Figure 3.3 and then go into more detail to explain the adornments.

The resulting safe `HAVING` clause (denoted by the start symbol *safeHavingClause* in rule 3.2) is only allowed to contain free variables that have a safe guard status +, all free variables with a different guard status are defined as non safe.

Furthermore, the two example rules 3.2 and 3.8 say the following: if during the derivation of a formula starting at the term $safeHavingClause(\vec{z})$ one reaches a

term of the form $hCl(\vec{z}^{\vec{g}^1})$, then one may infer a disjunction of two `HAVING` clause fragments under some conditions on the variables $\vec{z}$ that occur in them.

More concretely: For a clause $hCl(\vec{z}^{\vec{g}})$ with variables $\vec{z}$ and some adornment $\vec{g}$, Rule (3.8) justifies the production of $hCl(\vec{z}^{\vec{g}^1})$ `OR` $hCl(\vec{z}^{\vec{g}^2})$, if the adornment $\vec{g}$ can be represented as $\vec{g} = \vec{g}^1 \vee \vec{g}^2$, i.e., $\vec{g}$ is the result of applying a function $\vee$ on the adornment lists $\vec{g}^1, \vec{g}^2$.

We have to distinguish between two kinds of variables that occur in the `HAVING` clause. The first kind of variables describes entities, which are either individuals (such as a $?turbine$ in "$sensor123$ is attached to some $?turbine$") or literal values (such as $?x$ in "$sensor123$ measured temperature $?x$") in the dataset or for the second kind we have variables that refer to temporal states or more concrete to an ABox that occurs in the state sequence (such as $?i$ in "at point $?i$ in time $sensor123$ shows value 99").

Therefore, we investigate guard status only in the former case as we say that state variables cannot occur as free variable, but must be bound by some `FORALL` or `EXISTS` quantor and thus, only need to check variables declaring individuals or values for range restriction, which simplifies or `HAVING` syntax.

The functions $\neg, \vee, \wedge, \rightarrow$ over g-status vectors are defined in Figure 3.4. Combinations with the g-status $\emptyset$ are handled in an extra table, namely Table 3.4b.

Furthermore, we assume the ordering $\emptyset \preceq - \preceq -_+ \preceq +$ on the guard values. This ordering is relevant for the calculation of g_{max} in the rule of Fig. 3.3, where the clause is constructed from an arbitrary clause hCl and an identity atom. In the following example we show the evaluation of functions over g-status vectors.

Example 3. *Assume that one has produced a* `HAVING` *clause* $HCL(x_1^-, x_2^+, x_3^{-+})$*, where* x_1 *has g-status "$-$",* x_2 *has g-status "$+$", and* x_3 *has g-status "$-_+$". Then rule (3.8) and the tables allow, e.g., the production of* $HCL_1(x_1^-,\ x_2^+,\ x_3^{-+})$ `OR` $HCL_2(x_1^+,\ x_2^+,\ x_3^{\emptyset})$.

Let us verify this for the variable x_1*: Its g-status "$-$" in* HCL_1 *and its g-status "$+$" in* HCL_2 *combine to the g-status* $- = - \vee +$ *in HCL according to the entry for the pair* $(-, +)$ *in the table of* $\vee$ *(see 3.4a).*

The special case of $g_i = \emptyset$ is a convenience notation meaning for $x^{\emptyset}$ that x does not occur at all in the formula. In between of *not occurring* and *safe state* we see two forms of unsafety. The regular unsafe variable state (represented by "$-$") and an immediate state, which are actually negated safe variables (represented by "$-_+$") and thus, can be returned into a safe state again by adding a negation (represented by "$\neg$"). Regular unsafe variables stay unsafe also in the case of a negation.

g_1	g_2	$\neg g_1$	$g_1 \wedge g_2$	$g_1 \vee g_2$	$g_1 \rightarrow g_2$
$-$	$-$	$-$	$-$	$-$	$-$
$-$	$-_+$	$-$	$-$	$-_+$	$-_+$
$-$	$+$	$-$	$+$	$-$	$-$
$-_+$	$-$	$+$	$-$	$-_+$	$-$
$-_+$	$-_+$	$+$	$-_+$	$-_+$	$-_+$
$-_+$	$+$	$+$	$+$	$-$	$+$
$+$	$-$	$-_+$	$+$	$-$	$-_+$
$+$	$-_+$	$-_+$	$+$	$-_+$	$-_+$
$+$	$+$	$-_+$	$+$	$+$	$-_+$

(a) Variables existent in both subformulas

g_1	g_2	$g_1 \wedge g_2$	$g_1 \vee g_2$	$g_1 \rightarrow g_2$
$-$	$\emptyset$	$-$	$-$	$-$
$-_+$	$\emptyset$	$-_+$	$-_+$	$-$
$+$	$\emptyset$	$+$	$-$	$-_+$
$\emptyset$	$-$	$-$	$-$	$-$
$\emptyset$	$-_+$	$-_+$	$-_+$	$-_+$
$\emptyset$	$+$	$+$	$-$	$-$

(b) Variable missing in one subformula

Figure 3.4.: Combination of Guards

We also require the additional guard status $-_+$. This status is required as we do allow negations in arbitrary places of the formula, while in safe range normal form it is required to be in front of atoms or quantors.

We also allow double negations, which can possibly turn a safe range variable into the immediate non safe status $-_+$ and back again. Additionally, we allow implications, which basically leads to more subcases for the same reason, as we can rewrite implications of $A \rightarrow B$ into $\neg A \vee B$ using negations.

Domain Independence of STARQL HAVING Clauses

In the following we will show that safe STARQL `HAVING` clauses are indeed domain independent and therefore, transformable into relational algebra.

A well-known theorem from the literature states that every FOL-formula in domain relational calculus corresponds to a relational algebra expression, which is known to be domain independent [4, p.86]. Hence, to prove that also every STARQL `HAVING` clause is domain independent, (according to [4]) we have to show that each STARQL `HAVING` clause is already in or transformable into a *safe range normal form* and at the same time *range restricted.*

Safe Range Normal Form A formula that is in Safe Range Normal Form (SRNF) results from a transformation, which ensures that *(i)* no variable is bound and free or bound by different quantifiers, *(ii)* $F \rightarrow G$ is rewritten to $\neg F \vee G$, *(iii)* $\forall z$ is rewritten to $\neg\exists\neg z$, *(iv)* negation only occurs in front of an exists quantifier or atomic element.

1. $rr(t_1, \ldots, t_n) =$ variables in $t_1, \ldots, t_n$.
2. $rr(x\ op\ y) = \emptyset$ for $x, y \in Var_{val}, op \in \{<, >, \leq, \geq\}$
3. $rr(x\ op\ v) = rr(v\ op\ x) = \emptyset$ for $x \in Var_{val}, v \in Const_{val}, op \in \{<, >, \leq, \geq\}$
4. $rr(x = a) = rr(a = x) = \{x\}$ (for $x \in Var$, $a \in Const$)
5. $rr(F\ \texttt{AND}\ G) = rr(F) \cup rr(G)$
6. $rr(F\ \texttt{AND}\ (x = y)) = \begin{cases} rr(F) \cup \{x, y\} & \text{if } rr(F) \cap \{x, y\} \neq \emptyset \\ rr(F) & \text{else} \end{cases}$
7. $rr(F\ \texttt{OR}\ G) = rr(F) \cap rr(G)$
8. $rr(\texttt{NOT}\ F) = \emptyset$
9. $rr(\texttt{EXISTS}\ \vec{x}F\ \texttt{IN <Seq>}) = \begin{cases} rr(F) \setminus \vec{x} & \text{if } \vec{x} \subseteq rr(F) \\ \perp \text{ otherwise} \end{cases}$

Figure 3.5.: Rule set for checking a formula in SRNF for range restriction from [4]

This transformation into SRNF can be achieved by several transformation steps applied to STARQL `HAVING` clauses. The concrete steps are described in 4.2.1.

Range Restriction In the second step it has to be checked that the formula resulting from STARQL `HAVING` in SRNF is also *range restricted*, which means that every free variable of the formula is range restricted and thus, all possible answer sets are finite. This can be checked by a function rr given in Figure 3.5 [4].

Having defined a function $rr()$, we can further define range restriction for a given formula F as follows.

Definition 11. ***range restricted.*** *A formula F in SRNF is called* range restricted *iff $free(F) = rr(F)$ and no subformula returns $\perp$.*

For a transformation of STARQL `HAVING` clauses in safe range normal form and for the realization safe range queries, we have introduced the concept of safe `HAVING` clauses in the `HAVING` clause grammar above.

Theorem 1. *All safe `HAVING` clauses are range restricted.*

Let *safeHCl*($\vec{u}^+$) be a safe `HAVING` clause and *safeHClNF*($\vec{u}$) be a `HAVING` clause in safe range normal form. We see that for all subformulas $G(\vec{x}^+, \vec{y}^{-+}, \vec{z}^-)$ in $safeHClNF(\vec{u})$ we have

(*) $rr(G) = \vec{x}^+$ (= all variables in G with g-status +)

The proof of (∗) is by structural induction on construction of the formula *safeHClNF*($\vec{u}$).

Proof. We check (∗) for each fragment case of *safeHCl*($\vec{u}^+$):

- Let $G(\vec{x}^+, \vec{y}^{-+}, \vec{z}^-)$ be an **atomic clause**. Then $rr(G) = \vec{x}$ follows from looking at the adornments of the atomic clauses F in Fig. 3.3 and checking that only those with g-status + are in $rr(G)$. Hereby, variables $\vec{x}_{wcl}$ are treated as constants in the definition of $rr(\cdot)$.
- The case of **conjunction** is clear as any + g-status combines with any other g-status to +.
- The claim also holds for a **disjunction**,because a positive g-status for a variable in a disjunction is identified only if both variables are existing in the disjuncts and are labelled with +.
- Now take **negation** $G =$ `NOT` $F(\vec{x}^+, \vec{y}^{-+}, \vec{z}^-)$. Per definition of the STARQL `HAVING` grammar we know that $\neg(\vec{x}^+) = \vec{x}^-$. Additionally, we know that in all SRNF formulas a negation on F can only exist if F is an atomic formula. Therefore, by looking at the `HAVING` grammar, we see that no one of the atomic formulas returns a g-status $\vec{y}^{-+}$, hence actually $\vec{y} = \emptyset$ for atomic formulas and we only have $G(\vec{x}^-, \vec{z}^-)$. Thus, there exists no variable in G with g-status + and we get indeed that $rr(G) = \emptyset$ (see rule number 8 in Figure [4]), which is equal to the variables in G with g-status +.
- The final case is that of quantifiers and especially the **existential quantifier**, where we have that $G =$ `EXISTS` $\vec{x}$ $F(\vec{x}^+, \vec{y}^{-+}, \vec{z}^-)$. According to our induction assumption $rr(F) = \vec{x}$ and our STARQL grammar given in Figure 3.3, we assume that G may result from a transformation of an exists subformula `EXISTS` x $hCl(x^+, \dots)$ `AND` F' in *safeHCl*($\vec{u}^+$).

  So, the variable x is by definition in the set $\vec{x}$ of variables in F with g-status +, hence $rr(G) = rr(F) \setminus \{x\}$. However, x does not occur as free variable in G, hence the set of variables in G with g-status + is actually $\vec{x}$ without x, which proves the induction claim.

  As we allow also non SRNF formulas, we have to consider a `FORALL` quantifier. Here, G can be directly transformed by applying the rule `FORALL` $\equiv$

NOT EXISTS NOT such that there is a formula with variables that have g-status + and are bounded by the exists quantifier. Finally, one gets again a formula of the form EXISTS x $hCl(x^+, \dots)$ AND F'.

□

Relating our set of g-status with the set of g-status used in [4] for the definition of range restriction leads to the desired theorem.

Theorem 2. *All safe HAVING clauses (considered as queries on the DB $\mathcal{I}_t$ of certain answers within the actual ABox sequence at t) are domain independent.*

3.3.3. STARQL Semantics

In general we can instantiate our STARQL framework with different parameters regarding the ontology language and the embedded conditional language (ECL) that is used for UCQ graph patterns. For the proposed OBDA view, presented in the last sections, we use the standard W3C OBDA ontology language OWL 2 QL and unions of conjunctive queries, which are generally known to be domain independent [4].

Furthermore, STARQL was designed as a framework to also solve problems without query transformations and ontology based data access on triple stores, e.g., for using more expressive ontologies in the case of ABDEO (see Section 2.4.3). Therefore, to define the semantics regarding instantiations of STARQL (OL,ECL), we require the parameters for OL (ontology language) and ECL (embedded conditional language) to be defined in such a way that they provide a notion of certain answers for the ECL w.r.t. an ontology.

To explain the semantics of STARQL, we analyze the schema of a STARQL stream S_{out} in the following. We specify the denotation $[\![S_{out}]\!]$ of S_{out} recursively by defining the denotations of the components.

$$\begin{aligned} S_{out} \quad = \quad & \texttt{CONSTRUCT GRAPH } \mathit{timeExpCons}\ \Theta(\vec{a}_{wcl}, \vec{y}) \\ & \texttt{FROM } S_1\ \mathit{winExp}_1, \dots, S_m\ \mathit{winExp}_m, \mathcal{A}_{st}, \mathcal{T} \\ & \texttt{WHERE } \psi(\vec{a}_{wcl}) \texttt{ SEQUENCE BY } \mathit{seqMeth} \texttt{ HAVING } \phi(\vec{a}_{wcl}, \vec{y}) \\ & \texttt{GROUP BY } \xi(\vec{a}_{wcl}, \vec{y}) \texttt{ HAVING AGGREGATE } \Gamma(\vec{a}_{wcl}, \vec{y}) \end{aligned}$$

For ease of exposition we also assume that a query S_{out} specifies only one output sub-graph pattern and that there is exactly one static ABox $\mathcal{A}_{st}$ and one TBox $\mathcal{T}$ mentioned. Similar to the approach of LTL in [49], the TBox is assumed to be non-temporal in the sense that there are no special temporal or stream constructors.

Windowing

Let $[\![S_i]\!]$ for $i \in [m]$ be the streams of timestamped ABox assertions as defined in Section 2.4.5. The denotation of the windowed stream ws_i equals the evaluation of stream S_i by the STARQL window operator with window expression $WinExpr = lag_i$`<-[`$timeExp_i^1, timeExp_i^2$`]->`sl_i and is defined by specifying a window operator function F^{winExp_i} s.t.:

$$[\![ws_i]\!] = F^{winExp_i}([\![S_i]\!])$$

$[\![ws_i]\!]$ is a windowed stream with timestamps for each window from the set $T' \subseteq T$, where $T' = (t_j)_{j\in\mathbb{N}}$ is fixed by the pulse declaration with t_0 being the starting timepoint of the pulse. The domain D of the resulting window stream $[\![ws_i]\!]$ is a temporal ABox.

For each windowed stream ws_i having timestamp t_j, we define the temporal ABox $\tilde{\mathcal{A}}_i\langle t_j\rangle \in [\![ws_i]\!]$. Assume that $\lambda t.g_i^1(t) = [\![timeExp_1^i]\!]$ and $\lambda t.g_i^2(t) = [\![timeExp_2^i]\!]$ are the unary functions of time denoted by the window expressions of each windowed stream.

If $t_j < sl_i - 1$, then $\tilde{\mathcal{A}}_i\langle t_j\rangle = \emptyset$. Otherwise first set $t_i^{start} = \lfloor t_j/sl_i \rfloor \times sl_i$ and $t_i^{end} = max\{t^{start} - (g_i^2(t) - g_i^1(t)), 0\}$. On that basis we define the joint ABox of each windowed stream ws_i and timepoint t_j as $\tilde{\mathcal{A}}\langle t_j\rangle = \{ax\langle t\rangle \mid ax\langle t\rangle \in [\![S]\!] \text{ and } t_{end}^i \leq t \leq t_{start}^i\}$. Finally, the STARQL window operator joins the denotations of all windowed streams in a joined stream js w.r.t. the timestamps in T':

$$js([\![ws_1]\!], \dots, [\![ws_m]\!]) := \{ \bigcup_{i\in[m]} \tilde{\mathcal{A}}_i\langle t\rangle \mid t \in T' \text{ and } \tilde{\mathcal{A}}_i\langle t\rangle \in [\![ws_i]\!]\}$$

Sequencing

The stream $js([\![ws_1]\!], \dots, [\![ws_m]\!])$ is processed according to the sequencing method that is specified in the query. The output stream has timestamps from T' and its domain D now consists of finite sequences of ABoxes.

The sequencing methods used in STARQL refer to an equivalence relation $\sim$ to specify which assertions go into the same ABox. The relation $\sim$ is required to respect time ordering, i.e., it has to be a congruence over T. The equivalence classes are referred to as states and are denoted by variables i, j etc.

Let $\tilde{\mathcal{A}}\langle t\rangle$ be the temporal ABox of the joined stream js at t. Let $T'' = \{t_1, \dots, t_l\}$ be the time points occurring in $\tilde{\mathcal{A}}\langle t\rangle$ and let k' be the number of equivalence classes generated by the time points in T''.

Then define the sequence at t as $(\mathcal{A}_1, \ldots, \mathcal{A}_{k'})$ where for every $i \in [k']$ the ABox $\mathcal{A}_i$ is $\mathcal{A}_i = \{ax\langle t' \rangle \mid ax\langle t' \rangle \in \tilde{\mathcal{A}}_t$ and t' in i^{th} equiv. class$\}$. The standard sequencing method `StdSeq` is just $seqMeth(=)$. Let $F^{seqMeth}$ be the function realizing the sequencing.

WHERE Clause

In the `WHERE` clause only $\mathcal{A}_{st}$ and $\mathcal{T}$ are represented. So, purely static conditions (e.g. asking for sensor types as in the example above) are evaluated only on $\mathcal{A}_{st} \cup \mathcal{T}$.

As the `WHERE` clause is directly evaluated over graph patterns of the ECL, its results are defined by bindings $\vec{a}_{wcl} \in cert(ECL(\vec{x}_{wcl}), \langle \mathcal{A}_{st}, \mathcal{T} \rangle)$. This set of bindings is used in the `HAVING` clause $\phi(\vec{a}_{wcl}, \vec{y})$.

HAVING Clause

STARQL's semantics for the `HAVING` clauses is based on the certain answer semantics of the embedded ECL. We have to define the semantics of $\phi(\vec{a}_{wcl}, \vec{y})$ for every binding $\vec{a}_{wcl}$ from the evaluation of the `WHERE` clause. For every t we define how to get bindings for $\vec{y}$. Assume that the sequence of ABoxes at t is $seq = (\mathcal{A}_1, \ldots, \mathcal{A}_k)$. The set of bindings for $\vec{y}$ is what we call the *separation-based* certain answers, denoted $cert_{sep}$:

$$cert_{sep}(\phi(\vec{a}_{wcl}, \vec{y}), \langle \mathcal{A}_i \cup \mathcal{A}_{st}, \mathcal{T} \rangle)$$

We distinguish two cases: If for any $i \in Seq$ the pure ontology $\langle \mathcal{A}_i \cup \mathcal{A}_{st}, \mathcal{T} \rangle$ is inconsistent, then we set $cert_{sep} = \text{NIL}$, where NIL is a new constant not contained in the signature. In the other case, the bindings are defined as follows. For t one constructs a sorted first order logic structure $\mathcal{I}_t$: The domain of $\mathcal{I}_t$ consists of the index set $\{1, \ldots, k\}$ as well as the set of all individual constants of the signature. For every *stateAtom* `GRAPH i` $ECL(\vec{z})$ in $\phi(\vec{a}_{wh}, \vec{y})$ with a vector of free variables $\vec{z}$ having length l, say, introduce an $(l+1)$-ary symbol R and replace `GRAPH i` $ECL(\vec{z})$ by $R(\vec{z}, i)$. The denotation of R in $\mathcal{I}_t$ is then just stipulated as the set of certain answers of the embedded condition $ECL(\vec{z})$ w.r.t. the ith ABox $\mathcal{A}_i$:

$$R^{\mathcal{I}_t} = \{(\vec{b}, i) \mid \vec{b} \in cert(ECL(\vec{z}), \langle \mathcal{A}_i \cup \mathcal{A}_{st}, \mathcal{T} \rangle)\}$$

Constants denote themselves in $\mathcal{I}_t$. This fixes a structure $\mathcal{I}_t$ with finite denotations of its relation symbols. The evaluation of the `HAVING` clause is then nothing more than evaluating the FOL formula (after substitutions) on the structure $\mathcal{I}_t$.

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

Aggregation

The grouping and aggregation operators are applied as modifiers on the resulting binding list of values for free variables from the `HAVING` and `WHERE` clause. The binding list is grouped according to one or more free and safe variables related to their ordering in the `GROUP BY` clause.

On each group, an aggregate function is applied with one result per group. If no grouping is defined, the aggregation is done with respect to the complete result binding list.

Aggregation operators can be applied on free and safe variables either in the `SELECT` or `HAVING AGGREGATE` clause, but only in the later logical expression and evaluations on their results are possible.

Additionally, if any aggregate function is used, every variable that appears in the `SELECT` clause is also required to appear as a grouping variable.

Finally, let Θ_{agg} be a function that additionally maps the set of result bindings from the `HAVING` clause to aggregated and grouped binding lists, then by summing up we get the following denotational decomposition:

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

Properties of STARQL

The motivation for the STARQL semantics is a strict separation of the semantics provided by the embedded condition language ECL and the semantics used on top of it. This allows for embedding any ECL without repeatedly redefining the semantics of STARQL.

The separation-based semantics has an immediate consequence for perfect rewritability, which is adapted to the sequenced setting as follows. Let $\mathcal{O} = \langle(\mathcal{A}_i)_{i\in[n]}, \mathcal{T}\rangle$ be a sequenced ontology SO. Let the canonical model $DB((\mathcal{A}_i)_{i\in[n]})$ of a sequence of ABoxes be defined as the sequence of minimal Herbrand models $DB(\mathcal{A}_i)$ of the ABoxes $\mathcal{A}_i$. Let QL_1 and QL_2 be two query languages over the same signature of an SO, and OL be a language for the sequenced ontologies SO.

Definition 12. *QL_1 allows for QL_2-rewriting of query answering w.r.tthe ontology language OL iff for all queries ϕ in QL_1 and TBoxes $\mathcal{T}$ in OL there exists a query $\phi_\mathcal{T}$ in QL_2 such that for all $n \in \mathbb{N}$ and all sequences of ABoxes $(\mathcal{A}_i)_{i\in[n]}$ it holds that: $cert(\phi, \langle(\mathcal{A}_i)_{i\in[n]}, \mathcal{T}\rangle) = ans(\phi_\mathcal{T}, DB((\mathcal{A}_i)_{i\in[n]}))$*

Assume that the ECL allows for perfect rewriting w.r.tȮL such that the rewritten formula is again an ECL condition. We call such an ECL language *rewritability closed* w.r.t. the OL. Then, an immediate consequence of the separated semantics is the following proposition.

Proposition 1. *Let ECL be a rewritability-closed condition language and consider the instantiation of a **HAVING** clause language called QL_1. Then QL_1 allows for QL_2 rewriting for separation-based certain query answering w.r.t. the OL.*

3.3.4. Comparison of STARQL to SPARQL Syntax and Semantics

SPARQL, as presented in Section 2.3.3, is a W3C[5] recommendation since January 2008 and therefore well established as a query language for accessing on RDF data in triple stores or virtual access to other backend data bases by query transformation.

SPARQL itself does not directly support access to streaming data, nor to historical time stamped data, because of missing temporal operators. Nonetheless, as a standard query language for static data it sets the ground for a lot of RDF based query languages, which rely on SPARQL operators. Also STARQL was designed with the goal in mind that a SPARQL user has no difficulties when using its grammar and operators. Hence, we designed the STARQL syntax and semantics as close as possible related to SPARQL properties, although it is only a proof of concept design, not covering all possible features of SPARQL 1.1.

In the following we compare similar operators of STARQL as well as SPARQL and their usage in both languages.

Query Registration. SPARQL queries are one time queries in general, where a single query is sent to an endpoint and a query answer is retrieved as response. STARQL queries in comparison are continuous queries, they are continuously querying the dataset on a regular basis and therefore also have regular answer

[5]http://www.w3.org/

sets changing from time to time. For the registration process at the backend system STARQL provides a `CREATE` clause. The `CREATE` clause registers a stream with a specific name, allowing it to be referenced by other streams in the `FROM` clause.

Additionally to stream registration, the `CREATE` clause also allows a registration of pulse or sequencing functions, which supports the streaming topology in managing the timestamps and timebased input sources.

Query Forms. SPARQL offers four different kinds of query forms, namely `SELECT`, `CONSTRUCT`, `ASK` and `DESCRIBE` queries (see Section 2.3.3). The current version of STARQL supports two query forms, namely the `SELECT` and `CONSTRUCT` form. As a stream querying language its purpose is basically monitoring temporal or life data. Users are expected to know what kind of events they are looking for and therefore providing a describe clause, which is also not clearly defined and would be very implementation specific, is out of scope for this work. A support of an `ASK` query form is also a possible STARQL extension in the future. However, the `ASK` query form can always be replaced by a `SELECT` or `CONSTRUCT` form and therefor is not part of the prototypical implementation.

Data Input. The input of SPARQL is based on RDF Graph sources that are available for querying. One datasource can contain different named graphs and each graph can contain different triples. By using graph names indicating a temporal order, simulating temporal data is possible for graph input.

But querying with temporal relations and operators is only provided by language extensions. Additionally to static graph input, STARQL provides continuous access on data streams that can be mixed with static graphs by window operators. The use of several input streams is possible as well as several static graphs in the `FROM` clause.

Accessing Static Data. Access to static data is provided in the SPARQL `WHERE` clause by basic graph patterns combined with filter conditions.

STARQL as a framework also uses a where clause for static data. Its where clause is bound to a transformable query language for ontology based data access (ECL) to possibly providing perfect rewriting and unfolding. The static filter conditions restrict all states in the time-based `HAVING` clause, but also variables that are already bound in the `WHERE` clause.

Accessing Dynamic data. While SPARQL does not offer any inter graph comparison for state sequences, STARQL can reference different state graphs with filter conditions and manage intrastate (between different states) and interstate (in a single state) variables as well as index variables for different states in a first order formula. Although filter conditions are possible in inter state fashion used in the `HAVING` Clause, intra state filter conditions are restricted to UCQs and `OPTIONAL`, but can be extended in the future, e.g., using `FILTER` and `BIND` operators from SPARQL.

SPARQL 1.1. Operators. The are many extensions offered with version 1.1 for SPARQL. Next to new query language operators it comes along with federation of data sources, updating the dataset within the query language, transfer protocols and output in xml, json or csv language. While transferring to backend systems, not all of these additions can be provided by an ontology-based data access approach, but STARQL provides at least some of them, such as aggregations. Interesting for an RDF stream query language are especially the following extensions from SPARQL 1.1:

`Aggregation:` One of the new features in SPARQL 1.1 are aggregations. This is actually a very important feature for data monitoring and analysis and already implemented in the current version of STARQL as described in the syntax and semantics sections above.

`Negation:` As a further operator negation was introduced for SPARQL, extending the conditions in the `WHERE` clause by negated filters using the keyword `FILTER NOT EXISTS`. STARQL already implements negations naturally within its `HAVING` clause, as negated concepts are integrated in first order formulas, which are used for describing inter state relations in the ABox sequence.

`Subqueries:` In STARQL we design a topology approach instead of supporting subqueries, which allows for registration of substreams, and ensure that query results can be used as new data stream input for further registered continuous queries.

After comparing STARQL to the static query language for RDF data SPARQL, we explain how STARQL differs in accessing time in temporal states compared to other languages in the next section.

3.3.5. Expressing Temporal States with STARQL HAVING Clauses

The way how a query language handles time has an enormous influence on its use and complexity with respect to ontologies and reasoning. As an example for

different temporal access methods it is shown in [240] that reasoning on interval-based time models is generally more complex compared to reasoning over a point-based one.

However, also other ontological properties with respect to time have to be considered in parallel. Time can be integrated in different ways. We can integrate time in just another attribute, like many other RDF-Stream query languages do (see, e.g., Section 2.5.7, 2.5.4 or 2.5.2). Or on the other hand, we can directly integrate a temporal dimension in the semantics as we already discussed for non streamed data in Section 2.4.4.

Reified vs. Non Reified Time

In STARQL we have decided to choose a so-called *non reified* approach, which integrates time directly into the semantics. It basically opens a fourth dimension for time on each triple or graph. Thus, we can say that triples, which do exist at the same point in time belong to the same temporal graph. Furthermore, each graph formulates a temporal state that can be related in time to other states or graphs (see also Section 2.4.4).

A reified approach integrates time into data using a certain specific attribute. We argue that this approach has two major disadvantages. First, it blows up the data. Considering the measurement scenario, for adding temporal attributes, we would also have to add the concept of measurements. So finally, we end up with four more triples for the reified case:

{measurementA a Measurement; hasSensor ?sensX; hasValue ?valY; hasTime ?t}

Instead of a single quadruple in the case of non reification:

{?sensX hasValue ?valY <t>}

The difference in required memory is easy to see.

Our second argument against reification is about inference. Reification directly leads to possible combinations of inconsistencies that can not be prevented by a functional restriction in the ontology. We assume that one considers a reified approach of temporal measurements for RDF ontologies. Now, we would like to

formulate that only one measurement value can exist for a single sensor at a given point in time. As we need at least three triples to express a measurement, we can not prevent different measurements of a single sensor at the same time by a functional property $hasValue$.

Therefore, the challenge in dealing with temporal related inconsistencies, which is handled by functional properties in the non-reified case, can be seen as a disadvantage for the reified approach. Many other solutions for RDF stream querying languages try a reified approach or even omit all timestamps inside windows (e.g. SparqlStream, see Section 2.5.4), which can not be a solution either to prevent temporal inconsistencies.

On the other hand, the non reification method used in STARQL also supports expressing general temporal states for the OBDA approach, as used by LTL based languages.

Subsuming State Based Temporal Languages with STARQL HAVING Clauses

With temporal logic as introduced in Section 2.4.4 we investigate LTL-like languages for handling state based time dimensions in query languages. We also mentioned the most important recent query language for ontology based access on temporal states called TCQ [25].

As far as this language deals with ontology based data, it does not deal with relational backend data sources, but uses materialized data in some triple format. And thus, it neither uses the classical OBDA approach, nor is it restricted to a DL-Lite ontology language.

In [182] we showed how STARQL can be used to embed a particular TCQ fragment, which still holds for domain independence, in a `HAVING` clause and thus provide a classical OBDA approach with the help of STARQL also for TCQ.

Hence, we proved that STARQL allows for a classical OBDA approach with TCQ operators by two steps. We introduced a second semantics for `HAVING` clauses called *holistic* semantics, which no longer uses the separation based semantics that we have introduced in this work. And in a second step proved that the domain independent fragment of TCQ can be directly transformed into `HAVING` clauses by following the non separated approach.

As TCQ provides some operators without safety restrictions, STARQL is unable to embed all of them. This safetyness is a necessary feature of STARQL to be applicable for strict OBDA scenarios, where the background query languages are SQL like.

On the other hand, we can show that STARQL is still more expressive than the embedded fragment of TCQs. It offers user defined methods for creating ABox sequences, where as TCQ assumes that a sequence is given in advance without mentioning its creation.

Moreover, TCQ only allows intra state quantifiers within embedded CQs, but is unable to use inter state quantifiers, which are needed to express several kinds of queries, e.g., the STARQL monotonicity condition as shown in 3.9.

3.4. Concluding Remarks

Within this chapter we have presented a new query language called STARQL. The design of this language is grounded on the observations that we have described in Section 3.1 and especially on the state of the art technologies in Chapter 2, which each lack important features needed in use cases as given in the Optique project.

We further introduced syntax and semantics of STARQL, including two operators for handling temporal sequences. The sequencing operator itself and a `HAVING` clause for accessing the defined temporal states that can be used to subsume other standard temporal language in an OBDA approach (see Section 3.3.5). The query language additionally provides operators for combined access to streaming and temporal data, including a pulse function that synchronizes possible asynchronous input streams.

In the next chapter we further discuss the STARQL operators and explain how we can transform them from an ontology based view into relational systems.

4. Transformation of STARQL into Queries for Relational Systems

In the last chapter we have introduced the syntax and semantics of STARQL. This chapter discusses a so-called direct transformation strategy that allows us to compile STARQL queries based on a DL-Lite ontology, e.g., the Semantic Sensor Network ontology from Section 2.3.2, into the query language of a chosen relational backend streaming system (source). The proposed transformation strategy extents the classical OBDA approach in several important directions to *Ontology-Based Stream-Static Data Integration* and can be seen as a first step towards a fully fledged analytical, and cost aware OBDA system [135].

The idea behind our transformation steps that realize ontology-based temporal operators on relational backend systems is to extend the approach for static query transformations from Section 2.4.2 with additional query compilation for the STARQL window operator and `HAVING` clause aligning basic graph patterns with temporal information. The general transformation strategy is sketched in Figure 4.1.

As we consider a virtual ontology-based approach the figure consists of two layers, the virtual upper ontology-based layer with a query Q_{STARQL} and the actual database layer underneath evaluating the transformed query Q_{QL}, where QL is the the query language of the employed backend system.

The left hand side of the figure describes the query transformation process used by the STARQL engine. It consist of two main parts: (i) a transformation of the window operator and (ii) a transformation of the STARQL `HAVING` clause into relational algebra. Both transformations are applied for itself and its results are combined afterwards in Q_{QL}.

We can implement window operators in a direct way as relational DSMS are generally equipped with additional stream operators. A well-known query language for DSMS, e.g., is CQL [17] (see Section 2.2.6). In fact, for the implementation of a STARQL prototype in the Optique use case we propose as backend a stream-extended version of the Exareme system [227], which provides window operators for real time streaming in the spirit of CQL.

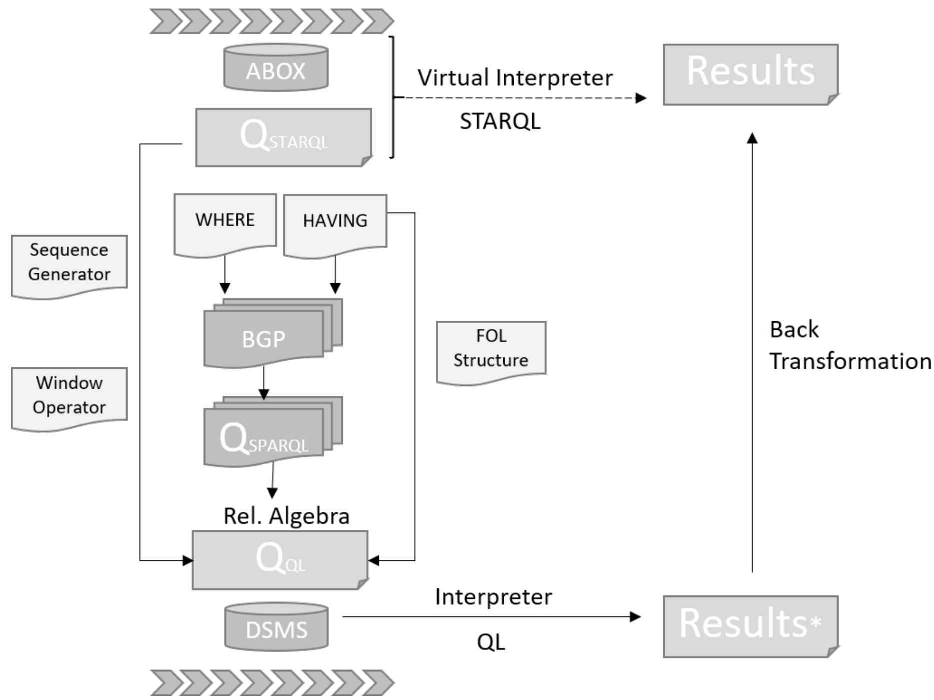

Figure 4.1.: Schematic Transformation of STARQL queries

Having shown that a STARQL `HAVING` clause is indeed domain independent is the main step towards using STARQL for OBDA in the classical sense according to which queries at an ontology level are rewritten and unfolded into queries over a data source.

The overall evaluation of the temporal sequence can then be summarized in two steps. First, we map the graph patterns of each state to the database views of the underlying defined source by the window operator. This is possible by creating simple SPARQL queries out of every single graph pattern and transforming them into relational queries with the standard algorithm for static ontology-based queries. Furthermore, we relate the generated relational subqueries based on first order logic formulas and temporal conditions in the `HAVING` sequence.

As (backend) data source candidates we can consider a data stream management system (DSMS) providing a SQL like streaming language (several of these systems are mentioned in Section 2.2) or in the case of historical data even simple database systems providing a declarative language such as SQL. Note that a database system does not necessarily mean a limitation in comparison with streaming approaches (in particular our own implementations [180, 181]), which rely on relational data

stream management systems (DSMS) as data sources.

The rest of the chapter describes details on the transformation process for an OBDA approach based on STARQL queries. We first explain how we transform the STARQL window operator into its representative form on the DSMS (Section 4.1). In a second step we transform the `WHERE` and `HAVING` clause, as given by its syntactical description from Section 3.3.1, into a relational algebra normal form (RANF). Since we require the `HAVING` clause to be in SRNF, we additionally have to add two normalization steps in preparation for the temporal sequence transformation that will be described in details in Section 4.2. After normalizing and transforming the `HAVING` clause, we have to consider several additional operations to complete the transformation result. Those steps include, e.g., the transformation of aggregators, `GROUP BY` clauses, as well as stream joins. Those are finally described in Section 4.3

4.1. Transformation of Window and Sequencing Operators

The general idea for an implementation of the STARQL window operator is a transformation of a view of timestamped tuples into a data base view that represents the sliding window and sequence information. Or more formally, according to Chapter 2 and Chapter 3, we have that a stream S is a possibly infinite bag (multiset) of elements $\langle s, \tau \rangle$, where s is a tuple belonging to the schema of S, and $\tau \in \Gamma$ is the timestamp. Then, let $[\![S_i]\!]$ for $i \in [m]$ be the streams of timestamped ABox assertions.

According to Section 3.3.3, we see that applying the window and sequencing operators can be expressed by three functions and window or sequencing parameters respectively, as given below.

$$F^{seqMeth}(js(F^{winExp_1}([\![S_1]\!]), \ldots, F^{winExp_m}([\![S_m]\!])))$$

Implementing those three steps on a relational database as data streaming system means: *(i)* creating a view that adds a column *windowID* to the rows of timestamped data, depending on *slide* and *width* of the related window parameters for each stream, *(ii)* creating a second view. Joining all input stream windows based on the given pulse function and *(iii)* creating a third view, which adds another column *ABoxID* for managing the sequencing strategy and merging each timestamp into the appropriate temporal state.

Table 4.1.: Example for incoming data in the measurement scenario

Timestamp	**Sensor**	**Value**
00:00	sens1	90
00:01	sens1	91
00:02	sens1	93
00:03	sens1	94

Let the data in Table 4.1 be incoming data of a measurement stream. A corresponding example view for a sliding window view with measured data is shown in Table 4.2. It includes measurements for a single sensor and describes two windows with *WindowID 1* and *2*, while the window width is three minutes and a slide of one minute is used.

The STARQL window generator provides a virtual view that represents time tagged data with a time series representation of the sliding window in two additional columns, where the schema consists of *WindowID*, *ABoxID*, *Timestamp* and [Datacolumns] for every temporal table.

Every single ABox in the sequence can then be accessed by an *ABoxID* in the second column.

Table 4.2.: Example for a sliding window view in the measurement scenario

WindowID	***ABoxID***	***Timestamp***	**Sensor**	**Value**
1	1	00:00	sens1	90
1	2	00:01	sens1	91
1	3	00:02	sens1	93
2	1	00:01	sens1	91
2	2	00:02	sens1	93
2	3	00:03	sens1	94

4.1.1. Window Transformation for Historical Queries

Now, we would like to show how a sliding window view can be generated for answering historical STARQL queries on relational databases such as PostgreSQL or others. The complete function naturally depends on the underlying backend

system. Thus, we only sketch the general transformation in this section and give concrete implementation examples for different systems in the next chapter.

WindowFunction. For F^{winExp_i} we propose a generation of *windowIDs* by two applied functions, namely $F^{dataJoin}(F^{borderGen})$. In the first step ($F^{borderGen}$) we generate temporal borders for each window, based on the window parameters *width* and *slide*. The resulting view is a sequence of *windowIDs* with additional columns for the left and right borders. The input data can be used for signaling the start and endpoint of the window sequence, where start and end is given by $t_i^{start} = \lfloor t_j/sl_i \rfloor \times sl_i$ and $t_i^{end} = max\{t^{start} - (g_i^2(t) - g_i^1(t)), 0\}$ (see Section 3.3.3).

Second, we join this view with the actual input-data of the incoming stream ($F^{dataJoin}$). A schematic example of the implementation is shown in Listing 4.1.

Listing 4.1: Example for generating a windowed sequence

```
1   CREATE VIEW Stream_wid AS
2   SELECT wid, timestamp
3   FROM inputData in, (
4      //subquery for empty window generation
5      SELECT wid, leftBorder, rightBorder
6      [...]
7      FROM inputData in
8   ) w
9   WHERE in.timestamp >= w.leftBorder
10     AND in.timestamp <= w.rightBorder;
```

JoinStream. In general the implementation of function js can be seen as a simple join of all input streams, but as we have different input parameters, each window or stream can be constructed by a different size and different temporal borders and as such, they cannot be simply joined on the $windowID$. We already introduced a pulse function in the last chapter for managing the join of input streams into joined windows. Furthermore, we also need a function that chooses at each *pulse* in time the related $windowID$ of each input stream. This can be done by adopting the windowing function from Section 3.3.3 with pulse extensions. Therefore, we join all input stream data based on the recalculation of the $windowID$ into a joined pulse with name $pWindowID$ as follows:

$$pWindowID = \left\lfloor \frac{WindowID \times sl_i}{pulse_{freq}} \right\rfloor.$$

SequencingFunction. The third function ($F^{seqMeth}$) that we have to apply adds an additional $ABoxID$ column to the sliding window view. In our example this can be seen as a simple recalculation of the windowed timestamps based on a granularity parameter of the sequencing method. Based on the parameter the time is rounded to seconds (or minutes) and entries with the same timestamp are then merged into respective ABoxes afterwards.

Finally, we arrive at the desired sliding window view from Table 4.2. This transformation strategy for the window and sequencing operator was implemented on the relational data base system PostgreSQL (see Section 5.3.1).

4.1.2. Window Transformation for Continuous / Real Time Queries

The transformation described for the STARQL window operator was meant for temporal reasoning on historical data that is stored in a RDBMS (e.g. PostgreSQL). And so the window table generation as part of the whole transformation above is a one-step generation for the whole dataset and not processed incrementally as in the case of real time streaming.

In order to cope with streaming data, the transformation process has to be slightly adapted for real time processing and thus, the window table now is assumed to be incrementally updated by some function. Apart from that, a similar transformation as for temporal reasoning can be applied to realize continuous OBDA querying with STARQL. In fact, the implementation of the transformation that we evaluate in the following only evaluates one window at a time, and hence it can be directly adapted for non-historic databases with dynamically updated entries.

Those ideas are often implementation-specific and encapsulated into evaluation functions of the backend streaming system, which means that many of these systems already provide their own window operators. An advantage is that the details can be managed by the underlying system that simply reduces the problem into a translation for passing the required input parameters of the operators to the specific window function.

The implementation of a state based temporal sequence is still new to these systems. Therefore, we use DSMS's that incorporates also functions for generating sequences based on different methods. One such system is Exareme, which includes ExaStream as a streaming extension (see Section 2.2.9) and comes with an expendable set of additional user-defined functions that simulate window operations and sequencing strategies in real time. Furthermore, ExaStream already includes scalability and distributed processing probabilities as well as several other optimizations.

4.2. Rewriting and Unfolding of STARQL HAVING Clauses

The described transformation process generally depends on mappings for grounding ontology level entities (constants, roles, concepts) in relational and streaming data. The `HAVING` clause of a STARQL query may involve static concepts such as *Assembly* or roles such as *hasSensor* and *mountedAt*, which can be handled by classical mappings in the sense that they define instances of roles and concepts on the basis of classical SQL queries over static tables. Thus, these static concepts or fragments can be transformed by standard query transformation algorithms as already described in Section 2.4.2

Furthermore, we can create mappings for the dynamic vocabulary, with names for roles such as *hasValue* that are mapped into streaming data during the unfolding process. Dynamic roles are different in a way that they are constructed from a general static mapping schema, but also from window and sequencing parameters of a given STARQL query as well, while composed from mappings for attributes and schemata of STARQL query clauses and constructs.

Attributes with dynamically changing values are declared in the `HAVING` clause. The schemata for temporal mappings are then dynamically generated during query processing and involve window as well as sequencing parameters that are specified in a certain STARQL query, which makes them dependent on time-based relations and temporal states. Note that the latter kind of mappings are not supported by traditional OBDA systems for static data.

In general the unfolding of a STARQL `HAVING` clause is organized in two layers. On the ground layer we consider each graph pattern in isolation for itself. A graph pattern is expressed in an embedded conditional language (in our case UCQ plus `OPTIONAL`) that is known to be domain independent (see Section 3.3.1) and can be mapped according to the rewriting and unfolding steps described for the static data.

From the top layer we see each graph pattern as a description for a single time point (ABox) based on the time windows generated by the window operator. Therefore, we embed the unfolding for each graph pattern in a larger structure of the transformed STARQL `HAVING` clause, based on the given relations between different points in time, described in the first order formula.

Schematic Mapping of Temporal Axioms in STARQL

For a better understanding, we show the mapping schema for the dynamic attribute `hasVal`, which would have in a static case the form:

$$\text{?sens hasVal ?y} \quad \leftarrow \quad \begin{array}{l}\text{SELECT sId as ?sens, val as ?y} \\ \quad \text{FROM Measurement}\end{array} \tag{4.1}$$

For the actual description of the mapping schema we use the same notation as for classical mappings, but additionally note that the right hand side now specifies a source stream together with data that is lifted to RDF triples. The actual time based assembled mapping is given as

$$\text{GRAPH i \{ ?sens hasVal ?y \}} \quad \leftarrow \quad \begin{array}{l}\text{SELECT sId as ?sens, val as ?y} \\ \text{FROM Slice(Measurement,i,r,sl,st).}\end{array}$$

The left hand side contains an indexed graph triple pattern, while the right hand side provides results from the mapping schema by applying a function *Slice*, which has to be implemented in each target system for the STARQL transformation approach. The *Slice* implementation is a parametrized function that describes the relevant finite slice of the stream $Msmt$ from which the triples in the ith RDF graph of the sequence are produced. As parameters we use the window range r, the slide sl, the sequencing strategy st and the index i.

This described approach of mapping streaming data with window parameters is due to the fact that, in the case of STARQL, we do not unfold a query to relational data only, but we unfold each mentioned graph pattern in the `HAVING` clause to temporal ABoxes of the ABox sequence by using an index variable. Thus, we use index variables and windowIDs to directly map the temporal states into window data.

As the generated state sequence is individually generated by the window and sequencing operator, we can not directly include it into the provided mappings. Thus, our approach rewrites the index variable connected to the unfolded graph pattern dynamically into each unfolding of the query and delegates the generation of the state sequence to the processing environment, as shown in the following examples.

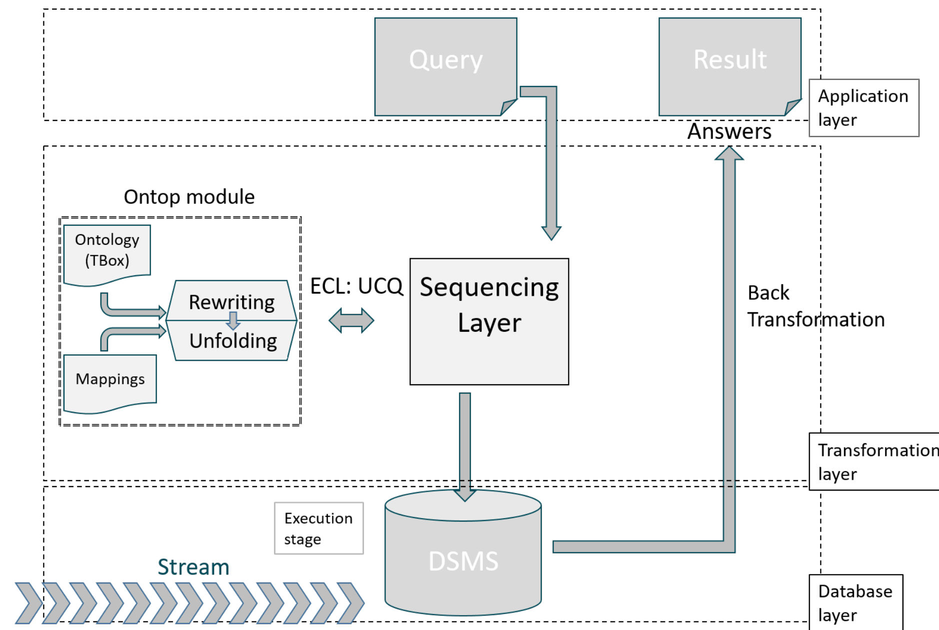

Figure 4.2.: STARQL transformation architecture

In comparison to the architectural description for static OBDA from Section 2.4.2, we extend the transformation of temporal data by a new sequencing layer (see Figure 4.2). This top layer manages the rewriting and unfolding tasks (executed by the Ontop module) for each graph pattern in ECL language and connects them by adding further conditions that are constructed by input parameters of the *slice* function (a detailed view on its implementation can be found in Section 5.1). For a better understanding of this advanced unfolding technique, we describe the processing of the `HAVING` clause in detail.

4.2.1. An Example Transformation for STARQL Having Clauses

To describe the transformation process in detail, we first consider once again our example from Listing 4.2.

Listing 4.2: Example query in STARQL

```
1   CREATE STREAM S_out_MonInc AS
2   CONSTRUCT GRAPH NOW { ?s rdf:type MonInc }
3   FROM STREAM S_Msmt [NOW-9s, NOW]->"1S"^^xsd:duration
4   WHERE {?sens sie:mountedAt ?ass}
5   SEQUENCE BY StdSeq AS seq
```

```
6    HAVING EXISTS ?k IN seq, ?m(
7          GRAPH ?k { ?ass sie:showsMessage ?m . ?m a ErrorMessage})
8       AND FORALL ?i < ?j  IN seq, ?x, ?y(
9          IF  ?j < ?k  AND GRAPH ?i {?sens :hasVal ?x}
10            AND GRAPH ?j {?sens :hasVal ?y}
11         THEN ?x <= ?y)
```

The query generally reports sensors, which are showing monotonically increasing values for the last ten seconds and are mounted on an assembly that additionally shows an error message at that particular time.

Furthermore, a specification for an error message and monotonically increasing values are defined in the `HAVING` clause. It states that within sequence *seq* there is no timepoint j before k (where the error message is recorded) and after a timepoint i, where the measured value of a sensor is lower than before—meaning that there is a monotonic increase of values for the particular sensor on the complete sequence.

In the following, we use the view of first order logic for a more formal description of the query processing. We transform the given `HAVING` clause into a pure FOL formula, by substituting each *stateAtom* GRAPH i GP($\vec{z}$), where $\vec{z}$ is the vector of free variables in the graph pattern GP with length l, having an appropriate $(l+1)$-ary relation $R(\vec{z}, i)$ for representing the set of certain answers of the embedded graph pattern in the ith ABox $\mathcal{A}_i$.

In the example from Listing 4.2 three different `HAVING` clause graph patterns are defined, which can be represented in FOL formulas accordingly by three relations R_1 (representing the message state), R_2 (representing the former sensor value) and R_3 (representing the later sensor value). The resulting FOL formula for the given query in Listing 4.2 is shown below.

$$\exists k, m(R_1(sens, ass, m, k)) \land \\ \forall i, j, x, y((R_2(sens, x, i) \land R_3(sens, y, j) \land i < j) \rightarrow (x \leq y)) \tag{4.2}$$

Making these assumptions, the given `HAVING` clause of STARQL can be transformed by following the algorithm below to transform from Safe Range Normal Form into Relational Algebra Normal Form (for detailed informations and proofs, see [4]). It basically includes three steps. *(i)* We show that the particular query is indeed domain independent. Therefore, we first transform it into the so called Safe Range Normal Form (SRNF) and check whether every free variable is range restricted. *(ii)* We reformulate the query in Relational Algebraic Normal Form, which allows for the algebra transformation. *(iii)* We apply the algebra transformation as given in

[4]), but use additional constraints concerning the window operator parameters for a fourth temporal dimension.

Safe Range Normal Form

In the following we will use the Safe Range Normal Form (SRNF) to show that the FOL formula (see Formula 4.2) for increasing sensor values is indeed domain independent (what was already proven for the general case in Section 3.3.2). It can be applied as follows.

For a formula F let $SRNF(F)$ be the formula in Safe Range Normal Form (SRNF) [4, S.85] resulting from applying the following normalization steps:

- Rename variables such that no variable symbol occurrence is bound by different quantifiers and such that no variable occurs bound and free;
- rewrite `IF` F `THEN` G to `NOT` F `OR` G; eliminate double negations;
- rewrite `FORALL`z with `NOT EXISTS z NOT`;
- push `NOT` through using de Morgan rules.

These steps are applied in some order until they cannot be applied anymore. A formula F is said to be in SRNF iff $F = SRNF(F)$.

A formula F in SRNF is called *range restricted* iff $free(F) = rr(F)$ and no subformula returns $\perp$. A well-known theorem states that range restricted formulas in SRNF are exactly as expressive as relational algebra—which is known to be d.i. Hence, it is well-known that safe range formulas are d.i. (in particular all sets of answers are finite).

After applying the four transformation steps, we result in Formula **??**.

One can directly see that the only existing free variable is *sens* and as it only appears in graph patterns, we say that it is bound to the finite answers set of the database and thus is range restricted. Therefore, $free(F) = rr(F)$ yields true and our desired formula is indeed domain independent.

Relational Algebraic Normal Form

In the second step, we normalize again for the Relational Algebraic Normal Form by three additional rules:

1. **Push into or:** We push every *or* upwards until there is no *or* as a child of an *and*.

2. **Push into quantifier:** if ψ is of the form $\psi_1 \wedge ... \wedge \psi_n \wedge \exists \vec{x} \xi$ and not all free variables of ξ are range restricted, we **push** a subset of $\psi_1 \wedge ... \wedge \psi_n$ as conjunct into ξ until all free variables of ξ are range restricted.

3. **Push into negated quantifier:** if ψ is of the form $\psi_1 \wedge ... \wedge \psi_n \wedge \neg\exists \vec{x} \xi$ and not all free variables of ξ are range restricted, **copy** a subset of $\psi_1 \wedge ... \wedge \psi_n$ as conjunct into ξ until all free variables of ξ are range restricted.

We can prove the correctness of the given transformation into Relational Algebraic Normal Form by two steps.

Proof. It can be shown by a case analysis that there exists no SRNF that is not in RANF and none of the given rewrite rules can be applied.

- The first rule is automatically applicable for any conjunctive query that has an *or* as a child of an *and*.

- The second and third rule are automatically applicable for any SRNF subformula ψ, because any free variables of ξ has to be range restricted in at least one subformula of $\psi_1 \wedge ... \wedge \psi_n$ by definition and therefore in any given STARQL query.

Second, the termination of the given algorithm is given because each applied transformation rule reduces the number of given subformulas in ψ. $\square$

In fact, these rules do not change the meaning of our given example formula and therefore, after both normalization steps the `HAVING` clause example from listing 4.2 can now be transformed from RANF into algebra as shown below.

Algorithm 2: Translation from RANF to Algebra

input : A formula ψ in modified RANF

output: An algebra query E_ψ equivalent to ψ

$$
\begin{array}{rcl}
R(\tilde{e}) & \longrightarrow & \delta_f(\pi_{wID,x_1,\ldots,x_n}(\sigma_F(R))) \\
x = a & \longrightarrow & \{\langle x : a \rangle\} \\
\psi \wedge \xi & \longrightarrow & \text{if } \xi \text{ is } x = x\text{, then } E_\psi \\
 & & \text{if } \xi \text{ is } x = y\text{, (with x, y distinct) then} \\
 & & \quad \sigma_{x=y}(E_\psi)\text{, if } \{x, y\} \subseteq free(\psi) \\
 & & \quad \sigma_{x=y}(E_\psi \bowtie \delta_{x \to y} E_\psi)\text{, if } x \in free(\psi) \text{ and } y \notin free(\psi) \\
 & & \quad \sigma_{x=y}(E_\psi \bowtie \delta_{y \to x} E_\psi)\text{, if } y \in free(\psi) \text{ and } x \notin free(\psi) \\
 & & \text{if } \xi \text{ is } x \neq y\text{, then } \sigma_{x \neq y}(E_\psi) \\
 & & \text{if } \xi = \neg\xi'\text{, then} \\
 & & \quad E_\psi - (E_\psi \bowtie E_{\xi'})\text{, if } free(\xi') \subset free(\psi) \\
 & & \quad E_\psi - E_{\xi'}\text{, if } free(\xi') = free(\psi) \\
 & & \quad \text{otherwise, } E_\psi \bowtie E_\xi \\
\neg\psi & \longrightarrow & \{\langle\rangle\} - E_\psi \text{ (if } \neg\psi \text{ does not have ``and'' as parent)} \\
\psi_1 \vee \ldots \vee \psi_n & \longrightarrow & E_\psi \cup \ldots \cup E_\psi \\
\exists x_1,\ldots,x_n \psi(x_1,\ldots,x_n,\, y_1,\ldots,y_m) & \longrightarrow & \pi_{wid,y_1,\ldots,y_n}(E_\psi)
\end{array}
$$

Transformation into Relational Algebra

For transforming a formula ψ from Relational Algebraic Normal Form (RANF) into an algebraic query E_ψ, we adopt the Algorithm from [4] and arrange it by projections of the window parameters for each relation by introducing a new and free variable wID, which connects each subresult to a windowID (see Algorithm 2).

To apply Algorithm 2, it must be shown that $q(\psi, \mathcal{O})$ and E_ψ are equivalent. In fact, STARQL `HAVING` clauses are by definition domain independent and therefore it is already guaranteed that every STARQL query q is *FOL-rewritabile*:

$$cert(q(\psi), \mathcal{O}) = E_\psi.$$

Additionally, a proof of the correctness of the algorithm can be given by straightforward induction for each fragment case as shown in [4, p. 89].

$$\pi_{wID,sens}(R_1) - \pi_{wID,sens}(\sigma_{x>y}(\sigma_{i<j}(\pi_{wID,sens,x,i}(R_2) \bowtie \pi_{wID,sens,y,j}(R_3)))) \tag{4.3}$$

By following the given algorithm for the STARQL `HAVING` clause in Formula 4.2, we arrive at an algebraic query shown in Formula 4.3, which can now directly be written in SQL. For writing the shown FOL formula in SQL, we have to unfold each graph pattern from static mappings (as described in Section 2.4.2) and insert the results for the relation term with R_1, R_2 and R_3.

The unfoldings $\mathcal{U}$ of R_2 and R_3 (referring to *hasValue*) can be generated according to the described mapping in Formula 4.1. For relation R_1 (*showsMessage, type ErrorMessage*) as well as the static *mountedAt* property from the `WHERE` clause, we give three simple mappings below in Formula 4.4, 4.5, 4.6. Please note that we need to join the static information for the location of each sensor into each temporal state as it was previously defined for STARQL in Section 3.3.3.

$$\text{?ass showsMessage ?m} \quad \leftarrow \quad \begin{array}{l}\text{SELECT cName as ?ass, mId as ?m} \\ \text{FROM Messages}\end{array} \tag{4.4}$$

$$\text{?m a ErrorMessage} \quad \leftarrow \quad \begin{array}{l}\text{SELECT DISTINCT mId as ?m} \\ \text{FROM Messages}\end{array} \tag{4.5}$$

$$\text{?sens mountedAt ?ass} \quad \leftarrow \quad \begin{array}{l}\text{SELECT sId as ?sens, cName as ?ass} \\ \text{FROM Sensor}\end{array} \tag{4.6}$$

By replacing the placeholders for temporal "states" in the algebra (R_1, R_2, R_3), we result in a complete unfolding $\mathcal{U}_{having}$ for the `HAVING` clause as given in Listing 4.2. The final algebra query is shown according to the unfolding $\mathcal{U}_{having}$ in Formula 4.7.

$$
\begin{aligned}
\mathcal{U}_{having(wID,sens)} \quad = \quad & \pi_{wID,sens}(\\
& \qquad \delta_{sId \to sens, cName \to ass} \pi_{sId,cName}(Sensor) \\
& \qquad \bowtie \delta_{cName \to ass} \pi_{wID,cName,mId}(Messages) \\
&) \\
& - \pi_{wID,sens}(\sigma_{x>y}(\sigma_{i<j}(\\
& \qquad \pi_{wID,sens,x,i}(\\
& \qquad\qquad \delta_{sId \to sens, cName \to ass} \pi_{sId,cName}(Sensor) \\
& \qquad\qquad \delta_{sId \to sens, val \to x} \pi_{wID,sId,val,i}(Measurements) \\
& \qquad) \bowtie \pi_{wID,sens,y,j}(\\
& \qquad\qquad \delta_{sId \to sens, cName \to ass} \pi_{sId,cName}(Sensor) \\
& \qquad\qquad \delta_{sId \to sens, val \to y} \pi_{wID,sId,val,j}(Measurements) \\
& \qquad) \\
&)))
\end{aligned} \tag{4.7}
$$

■

4.3. Additional Transformation of STARQL Operators

The complete transformation process of STARQL has to consider several additional operators that have been introduced in Chapter 3.

Aggregations

As described in the last section, all states are finally temporally related in the unfolding according to the defined FOL formula of the `HAVING` clause, which results in a returned vector $\vec{x}$ of free variables that are restricted by the temporal and static constraints of the query.

Mentioning the free variables, we see that a number of aggregations, as directly defined in the STARQL query, can now be unfolded into queries for the backend system. The aggregations are not defined in the ontology, but in the query itself and can be directly applied, while created dynamically using the aggregation clause.

Let $\mathsf{agg}(\vec{z})$ be an aggregate function on $\vec{z}$, $\vec{z} \circ r$ an arithmetic expression on $\vec{z}$ and E_{agg} an aggregate mapping resulting from STARQL queries. Then, we can define

$SQL_{E_{agg}}$ of the mapping $E_{agg} \leftarrow SQL_{E_{agg}}$ based on the free `HAVING` clause variables and aggregation operators as follows:

$$
\begin{aligned}
E_{agg} \quad \leftarrow \quad & \text{SELECT } x, \, \mathsf{agg}(y) \text{ FROM } \mathcal{U}_{having(x,y,z)} \\
& \text{GROUP BY } wID, \, x \\
& \text{HAVING } \mathsf{agg}(z) \circ r
\end{aligned}
\tag{4.8}
$$

Additionally to the already unfolded `HAVING` clause subview $\mathcal{U}_{having(x,y,z)}$, we have to consider an evaluation of the aggregation functions defined in the `SELECT` or `HAVING AGGREGATE` clause according to the complete temporal window using the $windowID$ as a grouping variable.

STARQL is designed to support advanced aggregation functions, such as multi column aggregation and more. This actually requires the same functionalities also in the backend system, as STARQL only references by internal mappings to functions of the backend language. A system that provides these functionalities is Exareme (see Section 2.2.9). Its engine allows us to use many different implemented user defined functions (UDFs), extending the globally used standard aggregations such as `MIN`, `MAX` and `AVG`. One popular UDF investigates on correlations between different sensor signals with the $Pearson$ function, which delivers results between one and zero (close to one means more correlated, close to zero less correlated). We conclude by giving the `HAVING AGGREGATE` unfolding for a sensor example using the pearsonCorrelation function of Exareme in the following statement:

$$
\begin{aligned}
& \text{SELECT } sensor \text{ FROM } \mathcal{U}_{having(sensor,y,z)} \\
& \text{GROUP BY } wid, \, sensor \\
& \text{HAVING } pearsonCorrelation(y, z) > 0.75;
\end{aligned}
$$

Furthermore, we cannot guarantee a correct evaluation of multi column aggregation in other systems beside Exareme, as it is not included in the current SQL standard *SQL2011*.

Orthogonality

We already discussed the orthogonality feature of STARQL in Section 3.2.1. For making sure that STARQL streams can be cascaded and reused within a query, we also have to consider mappings for managing the transformation of STARQL base streams to STARQL top level streams.

A common scenario for using orthogonality is the preprocessing of values for easier computation in a user friendly way. Let be given the STARQL query from Listing 4.3. A preprocessing step is done here in the S_{out_movavg} stream by calculating a moving average over sensor values from the input stream. It helps to eliminate outliers, when evaluating a possible monotonic increase in a succeeding substream (see also the example from Listing 3.9). Let incoming sensor values arrive every second, then in S_{out_movavg} the window with width of one minute provides a smooth curve as input for the following monotonic stream.

Listing 4.3: An example for STARQL orthogonality

```
1   CREATE STREAM S_out_movavg AS
2
3   CONSTRUCT GRAPH NOW { ?sens :hasAvg AVG(?y) }
4   FROM S_input [NOW - 1m, NOW]->1s
5   SEQUENCE BY StdSeq AS SEQ1
6   HAVING EXISTS i IN SEQ1 (
7       { ?sens :hasVal ?y . )
8   GROUP BY ?sens
9
10  CREATE STREAM S_out_moninc AS
11
12  CONSTRUCT GRAPH NOW { ?sens rdf:type RecentMonInc }
13  FROM S_out_movavg [NOW-1m, NOW]->1s
14  SEQUENCE BY StdSeq AS SEQ1
15  HAVING FORALL i,j IN SEQ1,?x,?y (
16     IF GRAPH i { ?sens :hasAvg ?x }
17        AND GRAPH j { ?sens  :hasAvg ?y } AND i < j
18     THEN ?x <= ?y )
```

For connecting both streams in the backend, we calculate a dynamic mapping that can be generated on the fly during query processing. This mapping is actually constructed by two cascading mappings. One reverse mapping for building an intermediate tuple view represents the output tuples of the first stream (output mapping) and an additional mapping vice versa, which is used by the second stream for identifying the incoming tuples from the substream (input mapping).

Let E_{out} be an reverse output mapping from the triple based stream S_{out_movavg} (as given in Listing 4.3) to an intermediate tuple view. Then, we can compute $SQL_{E_{out}}$ of the mapping $E_{out} \rightarrow SQL_{E_{out}}$ as follows:

$$
\begin{aligned}
E_{out} \leftarrow \; & \text{CREATE VIEW S_out_movavg_trples AS} \\
& \text{SELECT timestamp, ?sens AS subject, ':hasAvg' AS predicate,} \\
& \qquad \text{?avg AS object} \\
& \text{FROM S_out_movavg_having}
\end{aligned}
\tag{4.9}
$$

After having defined the intermediate view, let E_{in} be a mapping from an intermediate view to the triple based stream S_{out_moninc}. Then, we can formulate $SQL_{E_{in}}$ of the mapping $E_{in} \leftarrow SQL_{E_{in}}$ as follows:

$$E_{in} \quad \leftarrow \quad \begin{array}{l} \text{SELECT timestamp, subject, predicate, object} \\ \text{FROM S_out_movavg_trples} \\ \text{WHERE predicate = ':hasAvg'} \end{array} \tag{4.10}$$

This procedure can generally be applied for connecting two STARQL streams. Nevertheless, the transformation requires a backend that provides dynamically updated views as it is the case for the Exareme[1] system. The strategy is also restricted to be used together with the `CONSTRUCT` query form, because STARQL input streams are lifted through the OBDA approach from the backend tuple view to a triple view, which should not be contradicted.

After being mapped from a STARQL input, the stream can be handled as any other stream considering the given window and sequence based functions.

4.4. Concluding Remarks

The past chapter has explained how the overall translation process of the STARQL query language can be realized in an algorithm. We started with window and sequencing operators for live and historical data in Section 4.1, looked inside the classical translation for static data as also done for the STARQL `WHERE` clause, and finally, we described a new approach for translating temporal sequences based on classical algorithm for transforming safe range normal forms into relational algebra. We concluded by giving further examples on translations with respect to aggregation operators and cascaded STARQL streams.

The presented transformation strategy has been implemented in the stream query answering prototype of the EU Optique project. A detailed description and evaluation of several examples will be presented in the following chapters.

[1] www.exareme.org

5. Querying Relational Streaming Engines with STARQL

The last chapter has shown a detailed view on query transformation techniques for connecting the STARQL framework to relational based (streaming) backend databases. In this section we will show an overall description of the system, which transforms input queries and communicates to the backend. Furthermore, we would like to practically show that we are able to compile STARQL queries based on common ontologies in a proof-of-concept implementation into relational database queries. Therefore, we further define our test data in Section 5.2, which we already sketched during the last chapter to align it concretely with the semantic sensor network ontology. Furthermore, we define standardized queries to test the general STARQL features within specific test scenarios that are based on mappings to the underlying data model. Finally, in Section 5.3 we discuss transformation results for different implemented backend systems according to given test queries for historical and live streaming scenarios.

5.1. Implementation of a STARQL Streaming Engine

We can further combine each part of the STARQL transformation process in an architecture, by a prototypical implementation of the transformation given in the last chapter. Our architecture is related to the original architecture for an static OBDA approach in [57] as it is constructed at the top-level by four generally similar modules shown in Figure 5.1.

A module on the application level, which formulates the query and shows results, communicates to the transformation layer. Then, in the direction from application to data stream layer the STARQL transformation component translates the time based `HAVING` clause into the desired target language, while it rewrites and unfolds each temporal graph separately and without temporal specification as a static SPARQL query using the Ontop module for static query transformations. It finally translates everything into the target system using a specific query adapter that is implemented individually for each engine.

The final query can than be transfered to the backend system, for example using REST API, depending on the underlying system. We will later describe an example for a REST API based on the Exareme system, but note that not every underlying system or query language supports the same functionalities or features and therefore, different APIs are used in each case. Further, it is not even guaranteed that all features of the STARQL query language can be applied in a particular backend system. We show examples for different data stream management systems in the following parts of this chapter.

The third important module of the implementation is the actual DSMS backend, which is connected to the transformation component using a query handler. It usually organizes the communication process to the backend using a REST API, which registers continuous queries in the streaming case (or one-time SQL queries in the historical case) and provides an interface to retrieve results from the DSMS later.

For the backward direction from data stream to application layer a fourth module, namely the data serialization module, transforms streamed results, which are pulled through the REST API, into a ontology format. Its results are triples or tuples, based on the query form that is either a `CONSTRUCT` (triples) or `SELECT` (tuple bindings) query form.

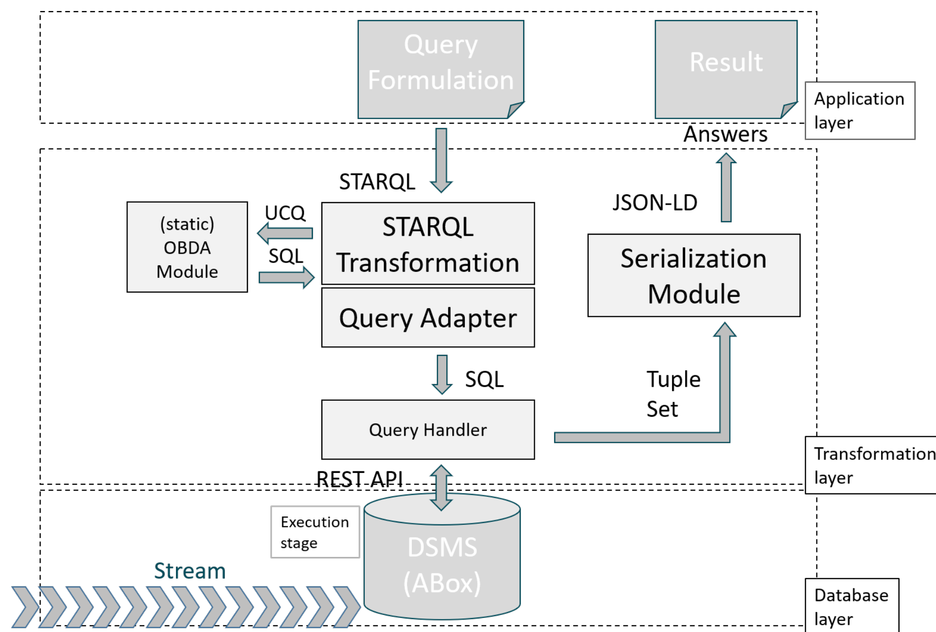

Figure 5.1.: Schematic implementation of the STARQL prototype

5.1.1. Transformation Module

We present the implementation of the transformation module in Figure 5.2. It shows the combined approach of a general transformation and the query adapter module. Besides parsing and providing the query data structure, the general module handles the transformation of UCQs into SQL fragments with respect to mappings handled by the external Ontop module. According to the process described in Chapter 4, each `GRAPH` i {...} form referring to state (i) is rewritten and unfolded locally. Results are joined internally into a new query in the query adapter, which guarantees a combined unfolding based on the syntactical requirements of the target system. Therefore, we call its output an *X-SQL* query, because it could be a dialect in SPARK, ExaStream, PipelineDB or other relational languages.

Additionally, mapping and ontology information have to be available independently for each use case. Also the specific backend DSMS has to be known in advance and a specific query adapter has to be chosen that initiates additional steps in the query unfolding or execution process and allows for running queries in a flexible way on different systems such as SPARK or Exareme.

In practice, information on the query adapter have to be added in a preprocessing step before the transformation starts. Thus, we have chosen a data source encapsulation for the STARQL framework. In this *data source* we can save further required parameters, such as the used mapping and ontology combination or information on the available backend streams and data, which have a reference name on the application layer, but might be referenced in the backend by an IP/port configuration.

Furthermore, the abstract *data source* view allows to expose different virtual abstractions from the same streaming source, while using different sets of mappings or ontologies for the transformation. An abstract *data source* can also expose the desired federation approach, where several different sources are combined to one streaming source, for example to provide a combination of streaming and static sources as it is proposed in the Optique project[1].

5.1.2. Query Processing

In Section 2.2 we presented several data stream management systems as possible stream backends for the STARQL framework. For our prototype we have chosen systems that provide declarative languages, where most current implementations use standards such as SQL or closely related streaming languages as CQL. The

[1] www.optique-project.eu

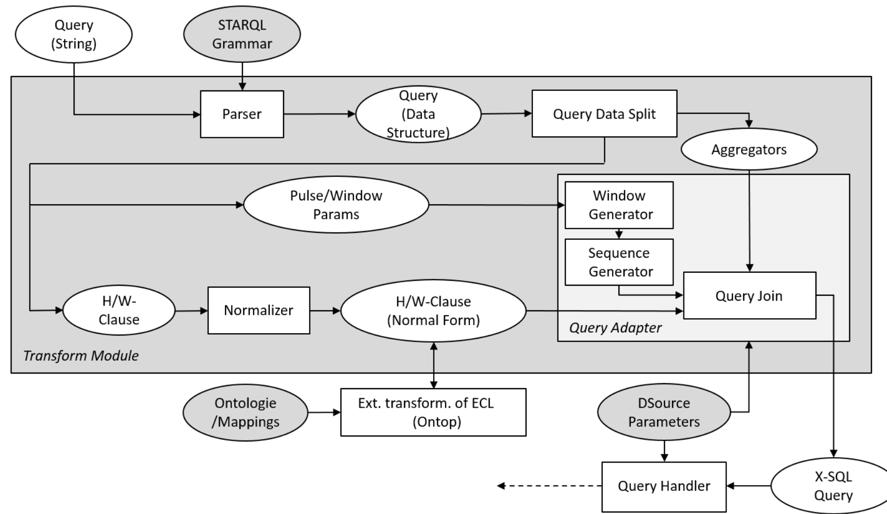

Figure 5.2.: Processing pipeline of the STARQL transformation

query processing between the STARQL framework and the backend system in these cases significantly depends on the implementation of the backend. In most cases the STARQL framework uses a specific REST API for registering continuous queries or receiving results via pull delivery.

Furthermore, a DSMS, which supports continuous queries, typically tracks all registered queries via a reference identifier. This identifier can either be delivered by the STARQL framework or received from the backend system after registering a query, depending on the implemented query adapter and the backend system itself. This reference can later be used to retrieve results or to delete a specific query.

A REST-API for transferring streaming data can be designed from two perspectives. Each of these models has one active and one passive side.

In the *pull model* the sender remains passive until it receives a request from the consumer node. It then provides the queried content to the requesting consumer node. An advantage of the scenario is that the receiving node is able to choose the exact amount and the content of the data it receives. However, the main disadvantage appears to be a case where the sender might receive too many requests (intended or unintended) that it is unable to serve. Additionally the server must have data in large endpoint buffers, which is largely impractically.

On the other hand, in a *push model* the sender is active, while the receiver remains passive. The sender starts sending its content as soon as it is ready to be published to the receiver. Thus, it does not have to wait for the receiver, while sending or

multicasting its content. The scenario has a clear advantage, as the sender does not get interrupted and a minimum of communication is ensured. A disadvantage could be that a receiver might get content, which is no longer interesting and lacks control on incoming data, until the receiver unsubscribe from the source.

We decided to implement a pull model in our prototype, because we suppose to have a secure use case environment and prefere the receiver to have control on the incoming data.

The prototypical query adapter is implemented in a simple setup case for connecting the STARQL framework to the popular DSMS of Exareme[2]. In the following we give a short description of the used API and refer the reader to [42] for more details.

Method	Type	URL	Param	Return Value
Register Stream	Post	http://HOST/{Method}/{Name}	register_query = {Query}	200 OK, 409 Conflict Error, 429 Too Many Requests
Delete Stream	Delete	http://HOST/{Method}/{Name}	-	200 OK, 404 Not Found
Get Stream Info	Get	http://HOST/	-	200 OK
Get Stream Results	Get	http://HOST/{Method}/{Name}	last={N}, startTimestamp={S}, endTimestamp={E}	200 OK, 404 Not Found
Historical Query	Post	http://HOST/{Method}/{Name}	-	200 OK
Read Table	Post	http://HOST/{Method}/{Name}	-	200 OK

Table 5.1.: Exareme REST API Functionality Overview

The REST service of Exareme supports two main functionalities. First, a registration of continuous queries together with pull facilities for retrieving its results, and second, the possibility to make one time historical queries on previously recorded streams and retrieving the answer set from a calculated table. We now briefly describe both processes.

Continuous Queries. A continuous query can be registered as given in Table 5.1. The used URL consists of the hostname, the method description and a unique identifier name, which represents the query reference name. Using this reference,in a second we retrieve step results of the specific query by using either the *last* parameter and a number (e.g. 5 for the last five tuples) or a combination of *start* and *end* timestamps for receiving results that are evaluated over a specific time

[2] www.Exareme.org

period (see row four of Table 5.1). On the other hand, the reference name allows for a deletion of the named stream (second row of Table 5.1).

Historical Queries. Historical queries can also be handled by the Exareme API. Instead of mentioning a reference identifier for such queries, we have to transfer a parametric name for a temporal output table. All query results are saved in the table and can be received by users (see row six of Table 5.1).

The implementation of the API above as prototype is used for evaluation purposes, and depending on the backend system, we can also think of other implementations such as the deployment of a server side JAR file in the case of Spark. We also provide a simple transformation prototype for transforming STARQL queries into the required backend format for running them manually on the desired system.

5.1.3. Serialization

The idea of serialization in computer science is the translation of data structures or objects into a storable or transferable format to be reconstructed later on the same or another machine. The operation of serializing (also called marshalling) is often used for transforming complex object structures into linear streams and turning them back again into the original structure.

Our proof of concept implementation is based on a serialization strategy for translating streamed results from the backend system into a storable and transferable format with respect to the underlying ontology, which should remain unchanged (e.g. for storing it on a triple store). In the case of RDF data, many different kinds of storage formats exist that can be transformed into or reassembled from streams. A first de facto standard was *RDF/XML* [40], an XML based syntax originally introduced to define RDF and the first standard RDF format. Much more compact and readable is *Notation*3 (N3) [44], together with the *Turtle* format [39], while N3 is similar to a SPARQL dialect (see Section 2.3.3). Much less complex than N3 or RDF/XML are line-based formats such as N-Triples [38] or N-Quads [86], which simply present an easy parsable format with one triple or quad per line.

However, we decided to use another format that has recently become popular, namely JSON-LD [153]. On the one hand it is easy for humans to read, write and understand and on the other hand it can easily be parsed, generated and streamed by machines. Developers who are familiar with the JSON standard can easily adopt its syntax and the large number of JSON parsers can also be used for JSON-LD. Further, it is a W3C recommendation since January 2014 [214]. It is commonly used in many projects such as the Google Knowledge Graph [212],

Microsoft Cortana [164] or BBC online content [33]. More professional JSON-LD users can be found on its own official Github page [55].

Therefore, we implemented a serialization component, for receiving answers in tuple-wise way from the relational data stream management system and for transforming them into a serialized JSON-LD format. An example JSON-LD query answer from the mentioned component can be found in Listing 5.1.

Listing 5.1: Example answer set for Query1 in JSON-LD format

```
1   [{"@graph":[{"_sens":"siemens:sensorinst-4","_val":"707.500"},
2   {"_sens":"siemens:sensorinst-5","_val":"690.000"},
3   {"_sens":"siemens:sensorinst-6","_val":"683.700"},
4   {"_sens":"siemens:sensorinst-7","_val":"720.200"}],
5      "@id":"2016-06-16T14:24:02+00:00"},
6   {"@graph":[{"_sens":"siemens:sensorinst-4","_val":"708.300"},
7   {"_sens":"siemens:sensorinst-7","_val":"719.600"},
8   {"_sens":"siemens:sensorinst-8","_val":"690.300"},
9   {"_sens":"siemens:sensorinst-11","_val":"831.900"},
10     "@id":"2016-06-16T14:24:03+00:00"}]
```

At pulse rate or window slide rate we receive one linked data graph in *jsonLD* format. The example in Listing 5.1 shows two succeeding graphs at timestamps *14:24:02* and *14:24:03* that deliver four different sensor measurements each. The given test data will be further described in the next section.

5.2. Test Dataset

It is important for our experiments that the test dataset, consisting of an ontology, a mapping, some relational data and interesting queries, is clearly defined to be reproducible and comparable within different sensor network scenarios. Therefore, we clearly define the used experimental dataset in detail and compare it to standard ontologies such as the Semantic Sensor Network ontology (see Section 2.3.2).

5.2.1. Data Schema and Example Data

We have already introduced the database schema together with STARQL examples in Chapter 3, which are based on data sets provided for the Optique[3] project. It basically consists of two types of input streams (*sensor measurement* and *event* streams) and a table with static information about the system and its devices. The

[3] www.optique-project.eu

database scheme are listed below in a simplified way, where sensors are named as *sensorIDs* (SID) and Assemblies by *assemblyIDs* (AID).

```
MEASUREMENT(TimeStamp, SID, Value)
EVENT(TimeStamp, AID, Category, Eventtext)
SENSOR(SID, Name)
Assembly(AID, Name)
SENSORMETADATA(SID, Property, Unit, RangeMin, RangeMax, MonitoredPart,
     Location)
```

Furthermore, the *sensormetadata* table provides static information for each sensor. It describes the specific type, e.g. temperature, pressure or speed, its measurement unit and location at the turbine. Additionally, we give example entries for each category in Table 5.2 below.

MEASUREMENT

TimeStamp	SID	Value
2015-11-20 00:00:01	25921	6.300

SENSOR

SID	Name
25921	TC260

Assembly

AID	Name
1	GasTurbine2103/01

SENSORMETADATA

SID	Property	Unit	RangeMin	RangeMax	MonitoredPart	Location
25921	Temperature	Degrees celsius	-40	1000	BurnerTip	GasTurbine2103/01

Table 5.2.: Example Data as used for query experiments

Along with the data schema we provide a dataset for further tests with measurement streams. The synthetic data provides sensor measurements of 19 sensors with a sampling rate of one minute per sensor. Further, the complete data has a temporal volume of three years, which is enough to represent a big data use case, as we can adjust the window parameter in any direction. The dataset roughly contains 30 million tuples. Therefore, if we use a sliding window approach with slide one

minute and width three years, we already deal with millions of windows and billions of tuples.

We continue with the ontological representation queries for our experimental evaluation (see Chapter 6) in the next sections.

5.2.2. Ontology

The presented semantic sensor net (SSN) ontology in Section 2.3.2 can be seen as the standard representation for ontology based sensor measurement scenarios. Based on the given dataset described in the last section, we concentrate our representation on the core module of the SSN ontology, the so called Stimulus-Sensor-Observation (SSO) Pattern (also described as *skeleton* module in Figure 2.7, see also [76] or directly for a stimuli description of the SSN Ontology [126]).

As an example, we consider the observation of temperature sensors deployed on a specific turbine. We describe the sensor as an instance of *ssn:Sensor*, while using the *Deployment* and *PlatformSite* module of the SSN ontology to define its topology (see Listing 5.2). Here, wee see the description of two different sensors of type *ssn:Sensor*, they are mounted on *Gasturbine2103/01* and measure either the temperature or rotation speed of the turbine.

Listing 5.2: Representation of two Siemens sensors used in a turbine installation

```
1    sie:TC260
2    rdf:type ssn:Sensor;
3    ssn:onPlatform sie:GasTurbine2103/01;
4    ssn:observes sie:Temperature;
5
6    sie:R117
7    rdf:type ssn:Sensor;
8    ssn:onPlatform sie:GasTurbine2103/01;
9    ssn:observes sie:RotationSpeed.
```

Besides that, according to the processing of external stimuli, we can define representations of observations and observation results as given in Listing 5.3. In this case an observation is shown which is observed by temperature sensor *TC260* and shows a value of 91.

Listing 5.3: Representation of a sensor observation

```
1    ssn:Observation1
2    rdf:type ssn:Observation;
3    ssn:observedBy sie:TC260;
```

```
4    ssn:observationResult ssn:Result1.
5    ssn:Result1 ssn:hasValue ‘‘91’’.
```

5.2.3. Mappings

We have already shown the relational dataset as well as the describing ontology for sensors and observations of our OBDA example dataset. Now, we would like to explain how both layers are connected by mappings of a R2RML file (see Section 2.4.2).

Listing 5.4: Mapping of sensor meta data based on a turbine installation

```
1    @prefix rr : <http://www.w3.org/ns/r2rml#>
2    @prefix ssn : <http://purl.oclc.org/NET/ssnx/ssn#>
3
4    ssn:SensorMap
5       a rr:TriplesMap;
6       rr:logicalTable [ rr:tableName "SensorMetaData" ];
7       rr:subjectMap [
8          rr:template "http://www.sensor.net/{Sensor}";
9          rr:class ssn:Sensor;
10      ];
11      rr:predicateObjectMap [
12         rr:predicate ssn:onPlatform
13         rr:objectMap [ rr:column "Location" ];
14      ];
15      rr:predicateObjectMap [
16         rr:predicate ssn:observes
17         rr:objectMap [ rr:column "Property" ];
18      ].
```

First, the mapping of the sensor topology, as shown in Listing 5.4, describes the mapping of sensors, e.g., those that are given in Listing 5.2, to the columns of the *SensorMetaData* table. While second, we create dynamic mappings for changing sensor values and observations to a relational measurement table that are shown in Listing 5.5.

Listing 5.5: Mapping of sensor observations based on the SSN ontology

```
1    @prefix rr : <http://www.w3.org/ns/r2rml#>
2    @prefix ssn : <http://purl.oclc.org/NET/ssnx/ssn#>
3
4    ssn:ResultMap
5       a rr:TriplesMap;
6       rr:logicalTable [ rr:tableName "Measurement" ];
7       rr:subjectMap [
```

```
8           rr:template "http://www.sensor.net/result/{Sensor}/{Timestamp}/{
                Value}";
9           rr:class ssn:Result;
10       ];
11       rr:predicateObjectMap [
12          rr:predicate ssn:hasValue
13          rr:objectMap [ rr:template "http://www.sensor.net/{Value}"; ];
14       ].
15
16    ssn:ObservationMap
17       a rr:TriplesMap;
18       rr:logicalTable [ rr:tableName "Measurement" ];
19       rr:subjectMap [
20          rr:template "http://www.sensor.net/observation/{Sensor}/{Timestamp
                }/{Value}";
21          rr:class ssn:Observation;
22       ];
23       rr:predicateObjectMap [
24          rr:predicate ssn:observedBy
25          rr:objectMap [ rr:parentTriplesMap <ssn:SensorMap> ];
26       ];
27       rr:predicateObjectMap [
28          rr:predicate ssn:observationResult
29          rr:objectMap [ rr:parentTriplesMap <ssn:ResultMap> ];
30       ].
```

5.2.4. Queries

Next, we are going to define four standardized queries in STARQL, which are used for testing the most important functionalities of our framework. For the following queries we assume that a *measurement* backend stream, as well as the mentioned static data for sensor descriptions are provided.

The first query (Listing 5.6) evaluates temperature sensor observations with a specific value higher than 41 degrees. Here, values are evaluated on a Window, which has a width of only a single minute as the query does not need to compare more than one measurement at a time. Additionally, the temporal stream is combined with static data, saying that observed sensors are mounted on *GasTurbine2103/01* and measure temperature data.

Listing 5.6: STARQL Query **Q1** (Threshold and static data)

```
1    PREFIX : <http://www.siemens.com/Optique/OptiquePattern#>
2    PREFIX ssn : <http://purl.oclc.org/NET/ssnx/ssn#>
3
4    CREATE
5    PULSE example_pls WITH FREQUENCY = "PT2M"^^XSD:DURATION
6
```

```
7    CREATE STREAM S_out AS
8
9    CONSTRUCT GRAPH NOW { ?sens :hasVal ?x }
10   FROM STREAM Measurement [ NOW - "PT1M"^^XSD:DURATION, NOW ]-> "PT1M"^^XSD
         :DURATION
11   USING PULSE example_pls
12   WHERE { ?sens a ssn:Sensor; ssn:onPlatform sie:GasTurbine2103/01;
13         ssn:observes sie:Temperature .}
14   SEQUENCE BY StdSeq AS SEQ1
15   HAVING
16   EXISTS i in SEQ1, ?val ( GRAPH i { ?obs a ssn:Observation;
17      ssn:observedBy ?sens;
18      ssn:observationResult ?val;} AND ?val > 41)
```

The second query (in Listing 5.7) is a query already known from Section 3.2.2. It shows the sequencing capabilities of the STARQL framework and queries for sensors, which have been showing monotonically increasing values in the last hour.

Listing 5.7: STARQL Query **Q2** (Sequencing)

```
1    PREFIX : <http://www.siemens.com/Optique/OptiquePattern#>
2    PREFIX ssn : <http://purl.oclc.org/NET/ssnx/ssn#>
3
4    CREATE PULSE example_pls WITH FREQUENCY = "PT1H"^^XSD:DURATION
5
6    CREATE STREAM S_out AS
7
8    CONSTRUCT GRAPH NOW { ?c2 a :RecentMonInc }
9    FROM STREAM Measurement [ NOW - "PT1H"^^XSD:DURATION, NOW ]-> "PT1H"^^XSD
         :DURATION
10   USING PULSE example_pls
11   SEQUENCE BY StdSeq AS SEQ1
12   HAVING FORALL i, j IN SEQ1, ?x,?y(
13   IF  GRAPH i { ?obs a ssn:Observation;
14         ssn:observedBy ?sens;
15         ssn:observationResult ?x;}
16      AND GRAPH j { ?obs a ssn:Observation;
17         ssn:observedBy ?sens;
18         ssn:observationResult ?y; }
19      AND i < j
20   THEN ?x <= ?y)
```

The next query in Listing 5.8 tests the ability to use different kinds of aggregations, which are used directly in the `SELECT` header or in an additional `HAVING AGGREGATE` clause. Here, we filter query results by a constraint that says no value is allowed to differ more from the average value than s6 units, which is a basic query for dropping outliers from the signal. In the answer set we show its maximum, its average and how much they differ from each other.

Listing 5.8: STARQL Query **Q3** (Aggregation)

```
1   PREFIX : <http://www.siemens.com/Optique/OptiquePattern#>
2   PREFIX ssn : <http://purl.oclc.org/NET/ssnx/ssn#>
3
4   CREATE PULSE example_pls WITH FREQUENCY = "PT1M"^^XSD:DURATION
5
6   CREATE STREAM S_out AS
7
8   SELECT ?sens MAX(?z) AS ?max, AVG(?z) AS ?avg, (MAX(?z)-AVG(?z)) AS ?diff
9   FROM STREAM Measurement [ NOW - "PT5M"^^XSD:DURATION, NOW ]
10     -> "PT1M"^^XSD:DURATION
11  USING PULSE example_pls
12  SEQUENCE BY StdSeq AS SEQ1
13  HAVING EXISTS i in SEQ1 (
14     GRAPH i { ?obs a ssn:Observation;
15        ssn:observedBy ?sens;
16        ssn:observationResult ?z;})
17  GROUP BY ?sens
18  HAVING AGGREGATE AVG(?z) + 6 < MAX(?z)
19 \
```

In the final query (Listing 5.9) we again use a technique for eliminating outliers or noise. To accomplish this, we have foreseen two cascaded STARQL streams. The first one is a moving average over sensor values, evaluated for a window of five seconds each. While the second evaluates the output of this stream and sends only the maximum values of the moving average for the last one minute to the output.

Listing 5.9: STARQL Query **Q4** (Orthogonality)

```
1   PREFIX : <http://www.siemens.com/Optique/OptiquePattern#>
2   PREFIX ssn : <http://purl.oclc.org/NET/ssnx/ssn#>
3
4   CREATE PULSE example_pls WITH FREQUENCY = "PT1M"^^XSD:DURATION
5
6   CREATE STREAM S_out_avg AS
7
8   CONSTRUCT GRAPH NOW { ?sens :hasAvg AVG(?z)}
9   FROM STREAM Measurement [ NOW - "PT2M"^^XSD:DURATION, NOW ]
10     -> "PT1M"^^XSD:DURATION
11  USING PULSE example_pls
12  SEQUENCE BY StdSeq AS SEQ1
13  HAVING
14  EXISTS i in SEQ1 (
15     GRAPH i { ?obs a ssn:Observation;
16        ssn:observedBy ?sens;
17        ssn:observationResult ?z;})
18  GROUP BY ?sens
19
20  CREATE STREAM S_out_max AS
21
22  SELECT ?sens, MAX(?z)
```

```
23    FROM STREAM S_out_avg [ NOW - "PT1H"^^XSD:DURATION , NOW ]-> "PT1M"^^XSD:
          DURATION
24    USING PULSE example_pls
25    SEQUENCE BY StdSeq AS SEQ1
26    HAVING
27    EXISTS i in SEQ1 ( GRAPH i { ?sens :hasAvg ?z})
28    GROUP BY ?sens
```

All of the mentioned queries above can either be evaluated on a live stream or on temporal data if the underlying system supports this functionality. The examples are designed for a real time streaming case. For the evaluation of archived temporal data the pulse function has to be extended by *start* and *end* parameter. An example pulse for historical data can be found in Section 3.2.2. Further, we have listed explicit transformation results for each query in Appendix A.

5.3. Implementation of the Ontology Based Streaming Back End Adapter

We already discussed several general aspects of the unfolding strategy into relational algebra and handling of streaming and temporal query structures in the general case, e.g., in Section 4.1 and 4.1.2, where we described the transformation of STARQL window operators.

However, data stream management systems often use their own syntax, in particular when it comes to the evaluation of historically recorded data, because many DSMSs are only optimized for real time streams. Therefore, we tried to simulate window operators that are closely related to standard SQL in Section 4.1 to support access on both types of input. PostgreSQL has no internal window operator implementation suitable for the management of time-based windows and thus, requires a complex alignment of different SQL operators to achieve a similar universal implementation of windows, which is also used for a implementation of a STARQL engine based on historic data in systems that are optimized for real time input.

Thus, the syntax as well as other specific execution properties of different backend systems have to be considered in each query adapter, for rewriting an SQL query to different backends. As shown in Figure 5.1, the Query Adapter is integrated in the transformation module.

In the following, with examples we describe all necessary implementation steps to transform STARQL queries for several backends. We use the threshold query from Section 3.2.1 with an additional pulse function (shown in Listing 5.6) as an example to give a proof of concept for each example backend system. In the historical case,

Table 5.3.: Comparison of implemented backend examples

	PostgreSQL	**Exareme**	**Spark**	**PipelineDB**
Live Streams	No	Yes	Yes	Yes
Static Data	Yes	Yes	Yes	Yes
Historic Streams	Yes	Yes	Yes	No
API	JDBC	REST API	REST API / built in	JDBC

we define the dataset to an absolute time series of one hour, which is referred to the pulse function (see Section 3.2.2). If the standard sequence method of STARQL can not be offered by a certain backend system, we choose an appropriate sequencing replacement strategy.

In fact, we show four different implementations. While two of them (SPARK and Exareme) support historical as well as live stream processing, PipelineDB supports only streamed data and PostgreSQL only historic processing.

5.3.1. Experiments on PostgreSQL Back End

The first possible backend system we consider is PostgreSQL, which can be seen as a standard SQL database, being really close to the SQL standard, but not optimized for stream processing. Therefore, it can not be configured as a service for continuous queries on a real time stream. PostgreSQL lacks the ability to handle continuously (possibly infinite) incoming data streams and queries as well as a generation of temporal windows on the fly.

Nevertheless, a PostgreSQL server still handles static data, which in the case of historically recorded input could also mean tuples with an additional timestamp column. Thus, our strategy for evaluating STARQL queries in the case of PostgreSQL leads into a system for answering historical queries that rely on recorded data and evaluate all necessary temporal windows in a single precalculation step. The STARQL framework considers temporal queries by direct use of a pulse function with *start* and *end* timepoint (see Section 3.2.2).

Temporal Querying

We have already sketched the general transformation strategy for the historical window generation in Section 4.1.1. Here, we basically introduce 5 additional steps as an implementation for the PostgreSQL backend system: *(i)* a definiton of the pulse in SQL, *(ii)* definition of windows on the series of timestamps, *(iii)* creating

a sampling on these windows according to the pulse function *(iv)* join of the data into the sampling view and *(v)* a generation of temporal ABox sequences for each window. The process can be formulated in five succeeding views for PostgreSQL, as shown in Listing 5.10. For the example transformation a pulse *start* of '2010-05-12 01:00:00' and *end* parameter of '2010-05-12 02:00:00' is chosen.

Listing 5.10: Example SQL Code for Query Example 1 in PostgreSQL

```
1
2   -- 1) Generate Pulse
3   CREATE VIEW pulse_example_pls AS
4   SELECT row_number() OVER ( ORDER BY pulse ) - 1 AS wid, pulse AS time
5   FROM ( SELECT generate_series ( '2010-05-12 01:00:00'::timestamp,
6   '2010-05-12 02:00:00'::timestamp, '0 seconds'::interval ) AS pulse)
        series;
7
8   -- 2) Window borders of Input Stream
9   CREATE VIEW measurement_public_window_range AS
10  SELECT row_number() OVER ( ORDER BY time ) - 1 AS wid,
11  time - ('60 seconds'::interval) AS left, time AS right,
12  lead(time, 1) over(order by time) as next
13  FROM ( SELECT generate_series ( '2010-05-12 01:00:00'::timestamp,
14  '2010-05-12 02:00:00'::timestamp, '60 seconds'::interval) AS time
15  ) series;
16
17  -- 3) Input Window sampling by pulse function
18  CREATE VIEW measurement_public_window_pulse_join AS
19  SELECT p.wid, m.left, m.right FROM
20  measurement_public_window_range m RIGHT JOIN pulse_example_pls p
21  ON p.time BETWEEN m.right AND m.next AND m.next > p.time;
22
23  -- 4) join data and timestamps into new WindowIDs
24  CREATE VIEW measurement_public_data AS
25  SELECT DISTINCT wid, tble.*
26  FROM measurement_public_window_pulse_join pj
27  LEFT OUTER JOIN measurement_public tble
28  ON tble.timestamp BETWEEN pj.left AND pj.right;
29
30  -- 5) Create ABoxIDs for each window
31  CREATE VIEW measurement_public_stream AS
32  SELECT dense_rank() OVER ( PARTITION BY wid ORDER BY timestamp ASC )
33     AS abox, * FROM measurement_public_data;
```

In addition to the general SQL schema shown in Section 4.1, several Postgres-specific functionalities are used. We give an overview for each of the five steps in the following list.

1. The pulse is generated by a Postgres-specific function *generate_series*(start, end, interval) that uses the pulse frequency as interval and its *start* and *end* parameters. On top a function *row_number()* is used to turn row numbers of

the generated series into *windowIDs*.

2. Similar to the pulse, we generate stream windows for each input stream, but have to consider the window *width*. Thus, we select a *right* and *left* border for each window instead of a single time point.

3. In a third step the pulse view is used as a sampling function on the generated input windows and joined into the previous view.

4. We join the windows once again with the incoming data based on their timestamps in the fourth step.

5. Finally, we add a ABox numbering by a dense rank function, which is similar to a normal rank function, but does not omit any numbers.

Furthermore, the PostgreSQL framework strongly conforms to the ANSI-SQL:2008 standard regarding its SQL implementation [193]. That makes an evaluation of the described STARQL transformation result from Chapter 4 fairly simple as no further adjustments are necessary. The transformed SQL query, based on query **Q1**, is shown in Listing 5.11.

Listing 5.11: Transformed HAVING clause in PostgreSQL for example **Q1**

```
1   CREATE VIEW S_out_having AS
2   SELECT wid, _sens, _z
3   FROM
4    ( SELECT * FROM
5      (
6            SELECT sens AS _sens, ass AS _ass FROM (
7            [...WHERE CLAUSE transformation...]
8            ) SUB_QVIEW
9      ) SUB_WHERE
10     NATURAL JOIN
11      (    SELECT wid, _z, _sens FROM (
12               SELECT * FROM(
13               -------------------modified ontop result begin
14                  SELECT wid, abox AS i, z AS _z, sens AS _sens FROM (
15                  SELECT DISTINCT  qview1.wid, qview1.abox, qview1."value"
                        AS "z",
16                  ('http://www.siemens.com/Optique/OptiquePattern#' ||
                        qview2."name") AS "sens"
17                   FROM
18                  measurement_stream qview1,
19                  sensor qview2
20                  WHERE
21                  (qview1."sensor" = qview2."id") AND
22                  qview2."name" IS NOT NULL AND
23                  qview1."value" IS NOT NULL
24                  ) SUB_QVIEW
25               -----------------modified ontop result end
26               ) SUB_TRIPLE0
```

```
27                ) SUB_QVIEW
28          ) SUB_HAVING
29    ) SUB_FROM;
```

One can see that the HAVING clause transformation is a join of a static and temporal part of the query (HAVING and WHERE clause respectively) that each selects the unbound variables from both clauses. At the heart of the temporal part are its ABox states (only one state is referred in query **Q1**), whose SQL structure is directly derived from the Ontop [60] module in each case. A postprocess of the STARQL framework joins each state and allows direct access to the previously generated window structure with a particular selection of its $windowID$ and $ABox$ variables.

Finally, all bindings are derived by the global selection of free variables and $windowID$s in the SUB HAVING SQL view, which are further modified by groupings and aggregation views (not shown here) or directly send to the output.

Scalabity of PostgreSQL with Multiple Session

What PostgreSQL servers lack, and thus also our backend implementation for PostgreSQL, is a feature for horizontal scalability or parallelization, which is naturally not supported by the query engine. Its whole implementation is process-based [110] (not threaded) and uses a single operating system process for each database session. Hence, each database connection (or query) only utilizes a maximum of one CPU core. Basically, the resulting idea for a better scalable approach on PostgreSQL is a split of the STARQL query into smaller queries or sessions, which can finally be automatically spread across all available cores of the operating system.

A streaming system such as our STARQL framework, typically supports such a split for temporal data, as the generated windows are evaluable separately with dedicated queries. For that purpose, we propose two different approaches.

The first idea is to split the dataset of computed windows into smaller parts, such that each part of windows can be evaluated in its own database session on its own CPU core with different queries aiming for one of the smaller parts each. However, this approach needs a manual data split and additional manual formulating of new queries, where the window generating would suffer at the start and end border of each data part.

Besides this rather naive version, we also implemented another approach that has been tested during this work. It automatically distributes a STARQL query over several PostgreSQL sessions or hosts using the procedural language *pl/pgsql* as an

extension. The idea here is to evaluate each window automatically on the fly, while they are distributed over different SQL sessions that run on one or more machines. Therefore, the query is not connected to an underlying window table or view, but to a client function that directly computes a single data window based on raw temporal sensor values and evaluates this window separately on the backend system. A simplified example for query **Q1** and multiple pl/pgsql sessions is shown in Listing 5.12.

Listing 5.12: Transformed HAVING clause in pl/pgsql for query example1

```
CREATE FUNCTION measurement_window(BigInt)
RETURNS SETOF my_window
AS $$
BEGIN
RETURN QUERY
SELECT $1 as WID, rank() OVER (ORDER BY timestamp ASC) as ABOX, *
FROM measurement_public where timestamp between
(SELECT * FROM time_start($1)) and (SELECT * FROM time_end($1));
END
$$
LANGUAGE plpgsql;

CREATE FUNCTION s_out_having(BigInt)
RETURNS SETOF my_hasval AS
$$
BEGIN
RETURN QUERY
SELECT wid, _sens, _z
FROM
 ( SELECT * FROM
   (
         SELECT sens AS _sens, ass AS _ass FROM (
         [...WHERE CLAUSE transformation...]
         ) SUB_QVIEW
   ) SUB_WHERE
   NATURAL JOIN
    (    SELECT wid, _z, _sens FROM (
            SELECT * FROM(
            -------------------modified ontop result begin
               SELECT wid, abox AS i, z AS _z, sens AS _sens FROM (
               SELECT DISTINCT  qview1.wid, qview1.abox, qview1."value"
                   AS "z",
               ('http://www.siemens.com/Optique/OptiquePattern#' ||
                   qview2."name") AS "sens"
                FROM
               measurement_window($1) qview1,
               sensor qview2
               WHERE
               (qview1."sensor" = qview2."id") AND
               qview2."name" IS NOT NULL AND
               qview1."value" IS NOT NULL
               ) SUB_QVIEW
            -----------------modified ontop result end
            ) SUB_TRIPLE0
```

```
43            ) SUB_QVIEW
44        ) SUB_HAVING
45    ) SUB_FROM;
46    END
47    $$
48    LANGUAGE plpgsql;
```

In the pl/pgsql example the transformed `HAVING` clause is no longer a view, but a function defined in a procedural language. The previous selection of the *measurement_stream* is now the call of a procedural window function. As parameter we simply take the $windowID$, which is further used to compute temporal borders of each window on the fly in the particular $time_{start}$ and $time_{end}$ function (not explicitly shown here). The advantage is that there is no precalculation of any windows required, everything is done per session and each session could be run on a different core or even separated machines, if the pl/pgsql code was deployed there. The complete server and client implementation for example **Q2** in pl/pgsql can be found in Appendix B.

Additionally, a client side management of the process is required to distribute all desired sessions on the server and collect each answer set as soon as a window result is ready. The process was implemented on the client side by a distributed round robin structure and cursors that can split answer sets in result batches when they become large. We have evaluated this implementation in Section 6.3.2.

5.3.2. Experiments on Exareme

Compared to the previous PostgreSQL implementation, Exareme[4] is a scalable system that was originally designed for cost-aware distributed processing and data streaming as well (see Section 2.2.9). Recently it has been recently extended with user defined python functions for streaming and window processing. Thus, a formulation of the window stream is not necessarily done directly in SQL (as shown in Listing 5.10 for PostgeSQL), but can be encapsulated in python functions.

We show an excerpt of the transformation result for Exareme in Listing 5.13. This query reads an input stream directly from TCP port inside the *newTimesliding-Window* function (see line four). The remainder of the query then is again a list of streams, while the *measurement* stream is used as a subquery inside of the *having* sub-stream (line 8).

Listing 5.13: Simplyfied transformation in Exareme for query **Q1**

[4] www.Exareme.org

```
CREATE STREAM measurement AS WCACHE SELECT * FROM
(newtimeslidingwindow timecolumn:0 timewindow:60 frequency:0 granularity
    :1 equivalence:floor
SELECT cast(strftime('%s', timestamp) as float) as epoch, * FROM
(file dialect:json 'http://192.168.11.37:8989/measurement'));

[...]

CREATE STREAM S_out_having AS WCACHE
SELECT DISTINCT wid, _sens, _z
FROM
 ( SELECT * FROM(
      (SELECT * FROM S_out_sub_ontop2) SUB_TRIPLE1
      NATURAL JOIN
      (SELECT * FROM S_out_sub_ontop3) SUB_TRIPLE2
      NATURAL JOIN
      (SELECT * FROM S_out_sub_ontop1) SUB_TRIPLE3
   ) SUB_WHERE
   NATURAL JOIN
    (    SELECT wid, _z, _sens FROM(
            (SELECT * FROM S_out_sub_ontop0) SUB_TRIPLE4
         ) SUB_QVIEW
   ) SUB_HAVING
) SUB_FROM;

[...]
```

Furthermore, Exareme is able to read from different sources, while in the example we read from TCP/port, streaming data could also be read from files or data bases.

The Exareme system requires a syntax that differs from the PostgreSQL case, as it does not directly support functionalities such as `LEFT JOIN`, `UNION` or `EXCEPT` in the streaming case. To realize these features, it requires a combination of the Exareme-specific `WCACHE` operator and Python functions to call substreams as in the following example:

$$\text{SELECT * FROM (streamexcept 'S_out_sub1, S_out_sub2') SUB_EXCEPT} \tag{5.1}$$

However, Exareme can not support the given semantics of STARQL completely. Its *newTimeslidingWindow* function generates a grid for temporal states, which means that also empty ABoxes are included in its result, which is not part of the STARQL semantics (see Section 3.3.3).

Finally, Exareme is especially optimized for distributing subqueries and thus, each query is divided into many substreams for optimization purposes (e.g. see SUB_TRIPLE1 to SUB_TRIPLE4 in Listing 5.13) STARQL and Exareme have been developed

very closely together in a prototypical implementation during the Optique project, which is presented in [130] and [131].

5.3.3. Experiments on Spark

Compared to PostgreSQL and Exareme, which are both based on a SQL like query language, Apache Spark provides an application programming interface centered on RDD data structures for Java, Python, Scala and R (see also Section 2.2.11). Furthermore, for our tests we implemented a Java program that can be deployed on the master node of a Spark cluster, instead of using a single SQL query as previously explained. The Java program has to define which transformations and actions are performed on the RDD. Thus, as transformations we can define the generation of temporal views, e.g., such as *sparkSession.sql("select value from measurement where value < 70").createOrReplaceTempView("low_values");*, while an action starts the evaluation of the program, such as in the case of *sparkSession.sql("select * from low_values).show;*.

Therefore, the idea is to provide a query adapter for the STARQL framework that provides a transformation result, which can be used in a single deployable Java program to generate all necessary RDD transformations and actions on the Spark cluster.

Real Time Evaluation with SPARK Streaming

As it is already optimized for window generation and processing, SPARK Streaming is the first choice when implementing real time data evaluation on Spark. Though it is not enough to pass a single query directly to spark, the engine naturally supports the generation of windows through its programming API, which is highly optimized for partitioning on the Spark cluster.

Thus, we have to compile a program that includes the complete transformation process in advance. The STARQL application is then executed on the spark Cluster, while it manages the evaluation of windows as well as its RDD transformations and actions directly on the master node.

Listing 5.14: Transformation result in Spark Streaming for Query Example 1

```
1  WINDOW Measurement_stream 60 60 stdsequence;
2
3  [Transformed SQL result for HAVING clause without WID]
```

An example transformation result for Spark Streaming is shown in Listing 5.14. The Listing can be divided into two parts: parameter for window generation and an SQL query, which is very close to the PostgreSQL result in Listing 5.11, but without any additional *windowID* variable, as the engine already guarantees that the query is executed separately on each window (RDD).

While the SQL result in the second part can just be transformed into transformations and actions for SQL statements, as given above, the previous lines have to be parsed for configuring each inputstream on the Spark cluster. We assume that the data is derived as strings via TCP port, which requires the following transformations internally by the deployed Java code to transform a DStream into window structures of tables. *(i)* Each tuple (timestamp, sensor, value) of the input stream (DStream) is added into an RDD array list, *(ii)* the string array is transformed into a preconfigured DStream of measurement table objects, *(iii)* we transform the object stream into a stream of windows with respect to the input parameters and finally *(iv)* the windowed DStream is transformed into an SQL view with added ABox information depending on the sequencing method. A simplification of the Spark Java code is shown in Listing 5.15.

Listing 5.15: Simplified Java code for Spark windows

```
1   JavaDStream<String> in = ssc.socketTextStream(host, port);
2   JavaDStream<String> list =
3      in.flatMap(Arrays.asList(line.split("\n")).iterator();
4   JavaDStream<Row> window =
5      win.transform([<String> -> <measurementObjRDD>]);
6   JavaDStream<Row> win = list.window(60,60);
7   Dataset<Measurement_Public> objTable =
8      sparkSession.createDataset(objRDD.rdd(),
9      Encoders.bean(Measurement.class));
10  objTable.createOrReplaceTempView("window");
11  Dataset<Row> out = sparkSession.sql(
12  "SELECT rank() OVER ( ORDER BY timestamp ASC ) AS abox, * FROM WINDOW;");
13  out.createOrReplaceTempView("Measurement_stream");
```

The integrated window structures of Spark Streaming already guarantee a window per window evaluation. Thus, no explicit window column is necessary and the provided SQL query can directly be evaluated by the transformer (see the PostgreSQL query without window generations for an example).

Temporal Evaluation with SPARK SQL

While real time queries are handled pretty well by the window generation of Spark Streaming, this is not the case for already recorded temporal data. The window

operator of Spark is not designed to use synthetic timestamps and such, it is not possible to evaluate timestamped data naturally with windows. The API would rather try to use real time timestamps on the incoming data and not the recorded ones. Even a replay of the data with synchronized time would not guarantee a correct result as recorded dataset would not be evaluated as fast as possible, but either too fast for the system or too slow.

Therefore, our idea for temporal data on Spark is once again to generate the time windows ourselves by using SPARK SQL. We show the changed transformer result with window generation for SPARK SQL in Listing 5.16.

Listing 5.16: Transformation result for SPARK SQL

```
1   VIEW time AS
2   select distinct unix_timestamp(timestamp) as value from measurement order
         by timestamp;
3   VIEW time_win AS
4   select collect_list(*) over (order by value range between *window width*
        preceding and 0 following) as array, value from time;
5   VIEW time_wid AS
6   select dense_rank() over (order by value) as wid, explode(array) as
        timestamp from time_win;
7   VIEW win AS
8   select wid - 1 as wid, from_unixtime(timestamp) as timestamp from
        time_wid where pmod(wid - 1, " + slide + ") = 0;
9   VIEW measurement_wid AS
10  select ceil(wid / *window slide*) as wid, timestamp from win;
11
12  [Transformed SQL result for HAVING clause with WID]
```

As already discussed, in the temporal case we bypass the streaming API of Spark and directly transform into SQL code for the window generation, which is then parsed and transformed statement per statement into SQL views on the spark cluster. As a side effect of the window generation, we require again a *windowID* column to guarantee the desired evaluation per window.

5.3.4. Experiments on PipelineDB

By an implementation of a query adapter for PipelineDB we face three major problems compared to other systems: *(i)* we need an additional client implementation that pushes the data into streams (streams can not be read or pulled from an external source in PipelineDB), *(ii)* no real window operator exists that could define a sliding parameter (width only), thus we cannot implement any pulse function, and *(iii)* a join of two data streams in one query is not possible.

Once we have arranged ourselves with these disadvantages, the implementation appears rather simple. First, we implement a client module that pushes our measurement data into PipelineDB streams, e.g., by using a copy command:

COPY stream(t timestamp, sens integer, val numeric(12,3)) FROM 'msmnt.csv'.

And second, we build the transformation result that consists of `STREAM`s and `CONTINUOUS VIEW`s as shown in Figure 5.17.

Listing 5.17: Transformation result for example1 in PipelineDB

```
1   CREATE STREAM measurement_stream(t timestamp, sensor integer, value
        numeric(12,3))
2
3   CREATE CONTINUOUS VIEW measurement WITH (max_age = '1 minute') AS
4   SELECT dense_rank() OVER ( PARTITION BY wid ORDER BY timestamp ASC ) AS
        abox,
5   * FROM measurement_stream;
6
7   CREATE CONTINUOUS VIEW measurement_having AS
8   SELECT _sens, _z
9   FROM
10   ( SELECT * FROM
11     (
12           SELECT sens AS _sens, ass AS _ass FROM (
13           [...WHERE CLAUSE transformation...]
14           ) SUB_QVIEW
15     ) SUB_WHERE
16     NATURAL JOIN
17      (     SELECT _z, _sens FROM (
18               SELECT * FROM(
19               -------------------modified ontop result begin
20                  SELECT abox AS i, z AS _z, sens AS _sens FROM (
21                  SELECT DISTINCT  qview1.abox, qview1."value" AS "z",
22                  ('http://www.siemens.com/Optique/OptiquePattern#' ||
                        qview2."name") AS "sens"
23                   FROM
24                  measurement qview1,
25                  sensor qview2
26                  WHERE
27                  (qview1."sensor" = qview2."id") AND
28                  qview2."name" IS NOT NULL AND
29                  qview1."value" IS NOT NULL
30                  ) SUB_QVIEW
31               -----------------modified ontop result end
32               ) SUB_TRIPLE0
33           ) SUB_QVIEW
34     ) SUB_HAVING
35   ) SUB_FROM;
36
37   select * from measurement_having;
```

After having declared the input streams and continuous views in PipelineDB, 100% of the PostgreSQL code can be reused and therefore, the succeeding views are quite similar to the ones used in the case of PostgreSQL (Listing 5.10 and 5.11). The only main difference can be seen in the formulation of continuous views, windows with *max_age* expression and the omitting of the *windowID*, which is also described in 2.2.10.

5.4. Concluding Remarks

This chapter has set the ground for experiments and evaluations in the upcoming chapter. We described the implementation of our approach in three steps:

We provided significant example queries together with mappings for a small ontological implementation of the Semantic Sensor Network ontology to provide a typical setting and scenario from industrial sensor network use cases in Section 5.2.

We gave an overview of the overall architecture and described the implementation of its components in Section 5.1. The approach is an extension of the classical OBDA approach with an integration of the algorithm for temporal sequences as shown in the previous chapter, and sketch a query adapter that allows transformations to different backend systems.

In Section 5.3 we have analyzed different implementations of the query adapter for several backends. The overview has shown that they all require different syntaxes or even techniques of deployment as in the case of Spark. Finally, they all have their pros and cons. While Exareme has been specifically optimized to fit a specific sensor stream use case as it is given in the Optique project, Spark is a system that allows for streaming and temporal data as well, while PostgreSQL only handles historical data, but can be used for optimized distributed window computations.

We evaluate the execution of the given test queries on different backend systems in the next chapter.

6. Evaluation of Query Processing with STARQL

In Chapter 3 we formulated three research problems that we would like to address in this thesis. They can basically be summarized by: (**P1**) *Can we design a query language that provides all necessary functionalities for an ontology based industrial streaming scenario?*, (**P2**) *Is an OBDA approach with query transformations still feasible?* and (**P3**) *Does the approach support an efficient execution on scalable big data architecture for streaming and temporal data?*.

To answer these questions and to support the more detailed hypotheses that we additionally have stated additionally in Section 3.1.4, we evaluate experiments and measure functionality, feasibility and efficiency of the implemented approach in this chapter.

We finally conclude with an overview and evaluation of each hypothesis in Section 6.4.

6.1. Functionality Evaluation - Comparison of RDF Stream Processing Engines

Our first goal regarding an evaluation of the STARQL framework is to show that its functionalities are at least comparable to other current RDF streaming languages and engines, but also sufficient for expressing complex queries in sensor measurement scenarios.

As described in Section 2.5, several RDF streaming systems have been developed besides STARQL. While all have their advantages and disadvantages, STARQL found its niche with respect to sequence based time handling and evaluation of historic data to make a great step forward in academic research.

We are first going to compare the properties of STARQL to concurrent streaming languages and second, investigate the STARQL language on formulating realistic

Table 6.1.: Comparison of RDF-Stream query languages (Part1)

Name	Data Model	Union, Join, Optional, Filter	IF Expression	Aggregate	Property Paths	Time Windows	Triple Windows
Streaming SPARQL	RDF-Streams	Yes	No	No	No	Yes	Yes
C-SPARQL	RDF-Streams	Yes	Yes	Yes	Yes	Yes	Yes
CQELS	RDF-Streams	Yes	No	Yes	No	Yes	No
SPARQLStream	(virtual) RDF-Streams	Yes	Yes	Yes	Yes	Yes	No
EP-SPARQL	RDF-Streams	Yes	No	Yes	No	No	No
TEF-SPARQL	RDF-Streams	Yes	No	Yes	No	Yes	Yes
STARQL	(virtual) RDF-Streams	Yes	Yes	Yes	No	Yes	No

complex queries in the measurement scenario using the SR and CSR Benchmark (see Section 2.6.2).

6.1.1. Comparing RDF-Stream Query Languages

For the functionality evaluation we have adopted a language overview table from [56], extended it for STARQL and updated the features of all other query languages to the best of our knowledge. Its result is shown in two parts in Table 6.1 and 6.2 respectively.

In part one we compare the implementation of the main SPARQL characteristics and features for each case. While all languages support basic functionalities such as `UNION`, `JOIN`, `OPTIONAL` and `FILTER`, we see that some languages (including STARQL) have already been further developed in the spirit of SPARQL 1.1 with `IF` clauses, aggregations, arithmetic expressions (not listed here) and others.

Furthermore, all of them, excepting EP-SPARQL (which is more based on events than time), support temporal windows, though only three of them support triple windows. Especially in the case of SPARQLStream and STARQL triple windows would lead into wrong results as they both use relational DSMS as backend systems, which do not use triples but tuples and, thus, it could be the case that many triples on the one hand represent a single tuple on the other.

The specific streaming capabilities and operators of each query language are compared in the second table. We can identify two groups of query languages, which differ in the management of time and for temporal operators in general. On the one hand we have a group that allows access to timestamps by operators that access each triple or object within windows. This group includes, e.g., SPARQLStream, C-SPARQL and STARQL.

While SPARQLStream uses reified time with additional axioms and STARQL a non reified version with a semantics of temporal states, C-SPARQL uses something in between and offers temporal functions on objects for retrieving their timestamp,

Table 6.2.: Comparison of RDF-Stream query languages (Part2

Name	W-to-S Operator	Cascading Streams	Intra window time	Sequencing	Synchronized Pulse	Historic data
Streaming SPARQL	RStream	No	No	No	No	No
C-SPARQL	RStream	No	Yes	No	No	No
CQELS	RStream	No	No	No	No	No
SPARQLStream	RStream, IStream, DStream	No	Yes	No	No	No
EP-SPARQL	RStream	No	No	Yes	No	No
TEF-SPARQL	RStream	No	No	Yes	No	No
STARQL	RStream	Yes	Yes	Yes	Yes	Yes

which could lead to inconsistencies if an object occurs several times inside a window in different temporal states.

However, we find a group of languages being developed with respect to temporal sequences and specific sequencing operators, such as in EP-SPARQL, TEF-SPARQL, and STARQL, that build a bridge to event processing. Though EP-SPARQL is different from the two other approaches, as it is more a CEP based language, they all share the possibility to define temporal sequences with operators. While EP-SPARQL extends SPARQL by four new binary operators: SEQ, EQUALS, OPTIONALSEQ and EQUALSOPTIONAL, TEF-SPARQL defines temporal facts and STARQL makes use of its special `HAVING` clause.

Finally, STARQL offers several new operators with functionalities that have not been included in previous systems. Next to cascaded streams, which can be seen as temporal sub queries, it offers the possibility of querying historically recorded data or even comparing it with a current live stream (see Section 3.2.2). Those different kinds of input streams (possibly using different kinds of window widths and slides) can additionally be synchronized in STARQL by one or more pulse functions, which allow a regularly query output for possibly asynchronous input.

The given overview has shown that STARQL subsumes most of the other languages and offers several novel features.

6.1.2. Comparing RDF based Streaming Systems

We presented a prototypical implementation of the STARQL framework for ontology based data access on temporal and streaming data in Chapter 5. This approach is based on R2RML mappings and query rewriting techniques as additionally presented in Chapter 4.

For a proof-of-concept implementation we proposed four different backend transformations and implementations showing different strengths of the overall system: *(i)* PostgreSQL as a standard relational database, *(ii)* PipelineDB as a standard

Table 6.3.: Comparison of RDF stream query languages

Language	**Input**	**Execution**	**Query Optimization**	**Stored Data**	**Reasoning**
Streaming SPARQL	RDF-Streams	physical stream algebra	Static plan optimization	Yes	No
C-SPARQL	RDF-Streams	DSMS based evaluation with triple store	Static plan optimization	Internal triple store	RDF entailment
CQELS	RDF-Streams	RDF stream processor	Adaptive query processing operators	Stored linked data	No
SPARQLStream	Relational streams	external query processing	Static algebra optimizations, host evaluator specific	Data source dependent	No
EP-SPARQL	RDF-Streams	logic programming, backward chaining rules	No	No	RDFS, Prolog equivalent
TEF-SPARQL	RDF-Streams	Yes	No	Yes	Yes
STARQL	Relational streams	external query processing	Static algebra optimizations, host evaluator specific	Datasource dependent	Yes (DL-Lite$_{\mathcal{A}}$)

streaming extension for PostgreSQL, *(iii)* Exareme as a scalable relational database on different nodes with integrated python functions that manage aggregations, and *(iv)* a standard and heavily used big data system, namely Spark, which is highly scalable and provides built-in streaming as well as graph and data mining libraries for specific additional purposes.

Our system can be compared to other existing RDF-streaming engines. We summarize the main facts of these architectures and compare them to the STARQL framework in Table 6.3.

Most of the streaming engines rely on native implementations of query processors. CQELS, for example, reimplements functionalities which do already exist in DSMS and therefore can be seen as a standalone engine. EP-SPARQL is based on logical programming and backward chaining, but is also implemented from scratch. Finally, C-SPARQL relies on an internal DSMS, but has no flexibility for mappings or rewritings.

Therefore, the only two systems supporting an ontology-based data access approach with mappings and a flexible backend are SPARQLStream and STARQL. As they both rely on external DSMS, they also both suffer from respective disadvantages. Query rewriting and translation of results can be expensive, also the expressiveness of the underlying system restricts the input of the RDF-Streaming queries (see also Section 2.4.1).

On the other hand, the query rewriting process also offers possibilities for query optimization, e.g., in the rewriting/unfolding process itself or on the backend, which might execute the query in a more efficient way compared to a directly implemented query processor.

6.1.3. Evaluating Functionalities in a Benchmark

When talking about a functionality benchmark for RDF-Stream query languages, probably the best choice is the SRBenchmark (see Section 2.6.2). It offers 17 well chosen queries in natural language and tests functionalities of SPARQL 1.0 and

Table 6.4.: SRBench result table

Language	**Q1**	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**	**Q7**	**Q8**	**Q9**	**Q10**	**Q11**	**Q12**	**Q13**	**Q14**	**Q15**	**Q16**	**Q17**
SPARQLStream	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
C-SPARQL	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
CQELS	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✗	✗	✗	✗	✗	✗
STARQL	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗

SPARQL 1.1 in a realistic scenario with respect to data from the linked open data cloud [85].

The tested features of the benchmark can be characterized by seven categories as listed below.

Graph Pattern Matching: included are basic operations on RDF tripel such as `UNION` or `OPTIONAL`.

Solution Modifier: covered are only projection and distinction, as the other modifications are generally covered by time or window properties.

Query Form: the queries of the SRBench are using `SELECT`, `CONSTRUCT` and `ASK` query forms.

SPARQL 1.1: several operators of the SPARQL extension are tested, including: aggregates, subqueries, negations, arithmetic expressions, property paths and assignments.

Reasoning: the queries currently involve reasoning over the following properties: *owl:subClass*, *owl:subProperty* and *owl:sameAs*.

Streaming Feature: the covered features are time-based windows with parameters and streaming operators (RStream, DStream, IStream) .

Data Access: the benchmark includes four different data sets from the linked open data cloud.

In its initial paper [248] three different RDF-Stream query engines where compared, namely C-SPARQL, CQELS and SPARQLStream. Recently, these languages have been further extended and their results for the SRBench improved. Their new results that give the expressable queries for each language, are shown in Table 6.4 together with a result for the STARQL framework. All updated queries of the benchmark together with their latest representations in the respective query language can also be found on the web [215].

While in 2012 SPARQLStream failed at 14 and CQELS/C-SPARQL at ten query formulation tests each, it now seems that SPARQLStream and C-SPARQL have been extended to express most of the test queries. However, the current implementation of STARQL was not able to express seven queries out of 17 and therefore, lies somewhere between the first results from 2012 and the latest updates of the

concurrent systems (except CQELS, which still has not been updated since 2012). The ten SRBench queries, which can be expressed in STARQL, are found in Appendix C.

We list the major functionalities that prevent STARQL from expressing the final queries:

ASK queries: currently the STARQL framework supports no `ASK` query form, which is required for query **Q3**. This query form also was not available in the original 2012 version of SPARQLStream and has been added recently. As this feature can always be replaced by a SELECT or CONSTRUCT form, we see it less important for a simple proof of concept implementation, but still is a possible candidate for future updates.

IF clauses: the IF clauses used in STARQL are not identical to those used in SPARQL or SPARQL extensions (as required in query **Q9**). While STARQL uses them in a declarative way and returns only a boolean value, which is similar to "if A is correct, then B also has to be correct", in SPARQL (and its stream extensions) `IF` clauses are used as function with return values, say, "if A is correct, then return B, else return C". These differences do not allow a direct implementation of **Q9** in STARQL. Nevertheless, we are able to express the query by using a substream for every `IF` clause (see Appendix).

Property Paths: the feature with the strongest impact on the SRBench results are *property paths*. As originally no RDF-Stream query language supported them in 2012, at least SPARQLStream and C-SPARQL claim to do this in their latest version. Furthermore, this feature is required in incomprehensibly many queries and therefore, all test queries between **Q12** and **Q17** can neither be expressed in CQELS, nor in STARQL. Though, only two different kinds of property path modifiers are used (i.e. "|" and "+") from the large operator set [9].

Besides the three listed functionalities above, which could be implemented in future versions of STARQL, there are no further issues influencing the results of the SRBenchmark or make queries of these kinds not expressible.

6.1.4. Discussion of the Functionality Evaluation

We can say that the basic functionalities of handling graph patterns are covered in the STARQL query language according to Table 6.1, which includes several features of SPARQL 1.1 (e.g., aggregates, arithmetic expressions, and negations) that also support hypothesis **H1**. Only very few features regarding graph patterns, which recently have been integrated in SPARQLStream and C-SPARQL, such as

property paths, have not yet been implemented. Nevertheless, an integration in future versions is possible.

Nevertheless, STARQL integrates a lot of functionalities for streaming and time handling that have not found their way into any or just partially into very few other RDF-Stream query languages yet. As shown in Table 6.1, those features are based on stream operators such as the sequencing method together with its `HAVING` clause or the pulse function that allows an evaluation of historically recorded time series and synchronizes stream outputs for non-synchronized input streams and especially in a mixed setting of live and recorded data (see Section 3.2.2).

Furthermore, we show evidence for supporting hypothesis **H2** and **H3**, which exactly state the support of historic and streamed data in one query.

The system most comparable to STARQL might be SPARQLStream because it also relies on an OBDA approach, which finally restricts the expressiveness of a query to a maximum of DL-Lite$_{\mathcal{A}}$, but nevertheless also supports hypothesis **H4**. Additionally, both systems use query transformations with respect to perfect rewriting from the RDF-Stream level to a relational level, where other languages directly implement their desired features in their own query processor (in a non-relational way) and reach a possible higher level of expressiveness.

Furthermore, STARQL shows features that also SPARQLStream supports and vice versa, as apparent in the SRBenchmark. The differences between the expressiveness of STARQL queries and SPARQLStream for the SRBenchmark are based on their differences in the rewriting techniques. While the STARQL implementation uses the Ontop framework for the translation of graph patterns (that currently does not support property paths), the implementation of SPARQLStream benefits from the Kyrie2 [171] rewriting module. The Kyrie approach tries to reduce the rewriting by existential constraints in a so called *EBox* inspired by Prexto [205].

For providing features such as property paths, Kyrie also needs to extend the expressiveness to $\mathcal{ELHIO}$, which is no longer in the range of perfect rewriting, but is based on Datalog rewriting as it requires recursive Datalog. Hence, it is also no longer ensured that the output can be reduced to a (finite) UCQ query and therefore, the result can be incomplete or even include wrong answers (see also [144]).

This discussion shows that, although features such as property paths can be added to a rewriting approach, they come at a cost of probably incorrect results, which still makes adding property paths questionable.

6.2. Evaluation of Rewriting and Transformation

In this section we review hypothesis **H5**, **H6**, **H7** and thus, evaluate the query rewriting and transformation process of STARQL itself.

6.2.1. (Non) Reification of Direct Mapping and Time

In Section 3.3.5 we already mentioned the problematic situation of handling timestamps as part of the underlying semantics or as part of the dataset. We now also would like to address this issue from a mapping and transformation perspective and explain how different ontologies or temporal semantics may or may not change the transformation result in the case of the STARQL framework.

Mapping of the Semantic Sensor Network Ontology

We recap the sensor observation description from Section 5.2.2 for the SSN Ontology and investigate its transformation result.

In Chapter 3 we used a very minimalistic model with axioms such as *?sens :hasVal ?x*, but in practice, ontologies can be more complex, like in the case of the SSN ontology. Here we have a reified data model in such a way that each measurement is modeled with a specific sensor observation object which is then part of further axioms for declaring the sensor, result values, the observed property, and other features (see Listing 6.1).

Listing 6.1: Representation of a sensor observation

```
1    ?obs a ssn:Observation;
2    ssn:observedBy ?sens;
3    ssn:observationResult ?r.
4    ?r ssn:hasValue ?x.
```

Transformation of graph patterns have been addressed in Chapter 2. According to the results in Section 2.4.2, we would expect that more axioms in a graph pattern also result in (probably unnecessary) joins in the transformation result, but tests with query rewriting and unfolding tools show that these (self) joins can be eliminated in an optimization step.

We show a transformation result of the observation model in Listing 6.1, tested in the Ontop transformation module with respect to R2RML mappings from the

implementation chapter in Listing 6.2 (names for result and observation are abbreviated).

Listing 6.2: Example transformation for observations and results (abbreviated)

```
SELECT  r AS _r, x AS _x, sens AS _sens, obs AS _obs FROM (
   SELECT qview1."value" AS "x",
   ('http://www.sensor.net/' || qview1."sid") AS "sens",
   ('http://www.sensor.net/result/' || qview1."sid") AS "r",
   ('http://www.sensor.net/observation/' || qview1."sid") AS "obs"
   FROM measurement_stream qview1
) SUB_QVIEW
```

The transformation results show that, although we have a join of four triples on the RDF layer, the unfolding can be optimized by optimization algorithms that eliminate all (self) joins, which makes us confident that the complex structures of the SSN ontology can be rewritten and unfolded with respect to mappings, but without any additional joins in the unfolding and thus, without any performance loss or increased complexity for the SQL query.

Reified vs. Non-Reified Time Transformation

The SSN ontology does not prescribe how timestamps should be handled. However, as explained in Section 3.3.5, we decided to choose a non reified time approach and integrate it directly into the semantics of the ontology, while most other approaches (e.g. SparqlStream) have chosen a reified approach and additional temporal axioms for each object.

Listing 6.3: Representation of a sensor observation (reified)

```
?obs1 a ssn:Observation;
ssn:observedBy ?sens;
ssn:observationTime ?t1.

?obs2 a ssn:Observation;
ssn:observedBy ?sens;
ssn:observationTime ?t2.
FILTER(?t1 < ?t2)
```

Therefore, it is a reasonable question, if a non-reified approach can be implemented and transformed in the same way as the classical reified one. To shed light on this, we have made two further experiments. First, we have evaluated a reified temporal approach regarding its transformation with respect to the SSN ontology

Listing 6.4: Transformation of a sensor observation (reified)

```
1  SELECT t2 AS _t2, obs1 AS _obs1, t1 AS _t1, sens AS _sens, obs2 AS _obs2
      FROM (
2     SELECT DISTINCT
3        ('http://www.sensor.net/' || qview1."timestamp") AS "t1",
4        ('http://www.sensor.net/' || qview2."timestamp") AS "t2",
5        ('http://www.sensor.net/' || qview1."sid") AS "sens",
6        ('http://www.sensor.net/observation/' || qview1."sid") AS "obs1",
7        ('http://www.sensor.net/observation/' || qview2."sid") AS "obs2"
8      FROM
9     measurement_stream qview1,
10    measurement_stream qview2
11    WHERE
12       (('http://www.sensor.net/' || qview1."timestamp")
13       < ('http://www.sensor.net/' || qview2."timestamp"))
14    ) SUB_QVIEW
```

and mappings. Thus, we handle timestamps simply as additional assertions, which can be seen in the example from Listing 6.3.

We model two observation with two different timestamps and add a filter constraint, which declares that *observation*1 happened before *observation*2.

A look at the transformed query in Listing 6.4 shows that the additional observation object and timestamp of the RDF layer results in a self join of the measurement table in the SQL result. In comparison to the previous example, this can not be prevented by an optimization algorithm, as we try to combine two different temporal states.

Listing 6.5: Representation of a sensor observation (reified)

```
1  HAVING EXISTS i,j in SEQ1 ( GRAPH i {
2     ?obs1 a ssn:Observation;
3     ssn:observedBy ?sens }
4  AND GRAPH j {
5     ?obs2 a ssn:Observation;
6     ssn:observedBy ?sens.}
7  AND i < j )
```

We now would like to compare these results to our non reified approach that is used in the STARQL framework. Thus, we formulate the same observation model of two observations, but with different timestamps in a STARQL `HAVING` clause (see Listing 6.5) and evaluate its transformation result in Listing 6.6.

Listing 6.6: Transformation of a sensor observation (non-reified)

```
1  SELECT * FROM(
2        ( SELECT DISTINCT  qview1."abox" AS "i",
3           ('http://www.sensor.net/' || qview1."sid") AS "sens",
4           ('http://www.sensor.net/observation/' || qview1."sid") AS "obs2"
5        FROM
6        measurement_stream qview1
7     ) SUB_QVIEW
8     NATURAL JOIN
9        ( SELECT DISTINCT  qview1."abox" AS "j",
10          ('http://www.sensor.net/' || qview1."sid") AS "sens",
11          ('http://www.sensor.net/observation/' || qview1."sid") AS "obs1"
12       FROM
13       measurement_stream qview1
14    ) SUB_QVIEW
15 )SUB WHERE i < j
```

While noticing that the timestamps $t1$ and $t2$ now have changed to temporal state ABox variables i and j, the experiment shows a transformation result, which is similar to the result of the reified approach before. We again have a self join of the measurement table, though each instance is encapsulated in a subquery. as we basically have the same join and selection.

Furthermore, the approach for using non-reified time in temporal states could provide cases, which would be even more efficient. If we thought about further constraints for a specific sensor, say $sensor1$, then this sensor would be selected in the reified case after a self join, but for the non reified version even before, which can reduce the cost of joins a lot, depending on the specific constraints we choose.

Finally, we showed that a rewriting of the SSN ontology with non reified temporal semantics does not have any negative effect on the resulting SQL transformation, but for some cases it can even be more efficient.

6.2.2. Evaluation of Transformation and Delays

Our prototypical implementation of a STARQL engine provides four different query adapters, each with its own characteristics. While Exareme and Spark are scalable systems for historical and live streaming data, we also support PostgreSQL as an example for the SQL standard with historical processing and PipelineDB, which is an extension of PostgreSQL for live streaming only.

Before we focus on a comparison between these systems, we first concentrate on evaluating the general rewriting and translation performance, which is the ground-

ing for each backend connection. Thus, we would like to measure the additional overhead that these techniques add to the query processing in real scenarios.

In order to do that we evaluate the temporal costs of the back translation process, which is necessary each time we retrieve a tuple from the backend system and translate it into a JSON-LD format by the STARQL serialization module (see Section 5.1). As the query is rewritten only once (with translation times usually below 1 second), but the query results are translated regularly for each resulting tuple, we concentrate on the translation of query results, but also show the query transformation time afterwards.

We evaluate query translation times of the STARQL framework for different tuple rates and different window sizes, using a simple query as given in Listing 5.6 and compare it to the results for SPARQLStream from [56]. The translation process was evaluated on a single machine with an Intel i7 2.8 GHz processor and 8 GBs of RAM.

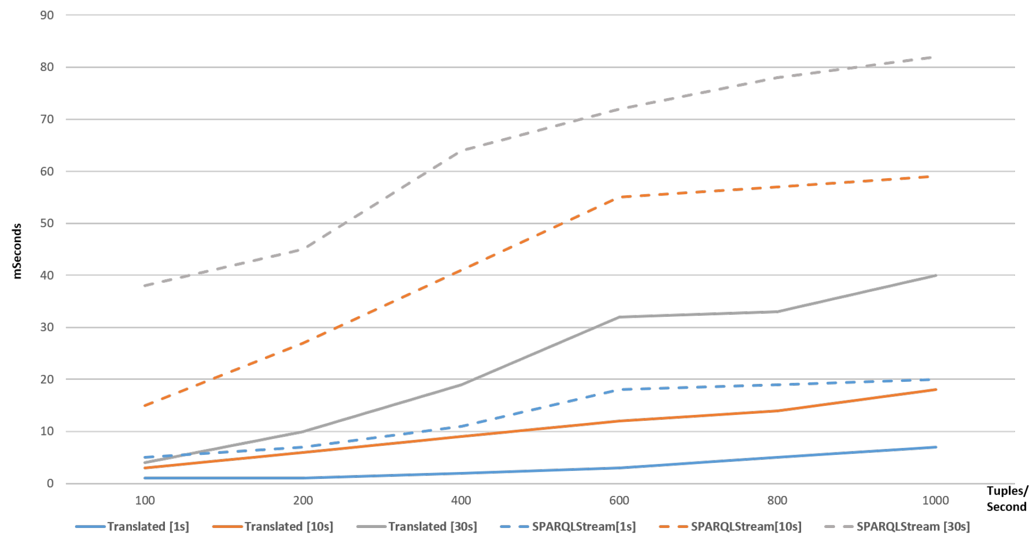

Figure 6.1.: Comparison of tuple translation delay

First, one can see that the results between STARQL and SPARQLstream are pretty close, although STARQL results are a little bit less delayed. As we expected, the delay increases according to the streamed tuples to be transfered per second and the window size of the query. The STARQL engine already transfers some transformation steps into the backend, which allows for efficient processing and shorter delays on the client side (see also Chapter 4).

Table 6.5.: Query rewriting delays for different query examples

System	**Example1**	**Example2**	**Example3**	**Example4**
PostgreSQL	251ms	336ms	307ms	295ms
Spark	317ms	337ms	300ms	307ms

Second, we show the query transformation times for PostgreSQL and Spark. Transformation has to be done once before registering each query in the backend. We list the delay in milliseconds for each of our four example queries from Section 5.2.4 in Table 6.5.

The table basically shows the feasibility of translating all chosen example queries. Translation Times lie within a range of 250 to 350 milliseconds for PostgreSQL or Spark, which literally explains why query rewriting is not an issue when it comes to estimating the performance of query answering. In particular, query rewriting is only executed once in the process, namely each time a new query is registered on a backend system.

6.2.3. Discussion of the Transformation Process

In this section we answered three different questions, which are important for the overall results of the STARQL framework.

We first had a look at the rewriting process of the standard ontology for semantic sensor data and showed that additional complex axioms in the ontology on the one hand does not necessarily require any additional complexity for queries on the relational data base side. Using an appropriate set of mappings and a primary as well as foreign key structure on the database level, ontology based data access modules are able to optimize the rewriting and unfolding in such a way that no unnecessary joins are produced. These observations also support our hypothesis **H5** from Chapter 3.

Second, we also discussed the representation of time and its impact on query rewriting. The investigation of an example showed that the rewriting and unfolding is not identical, but the comparison of different timepoints comes hand in hand with a join of the time series data on the relational backend. But on the other hand a clear distinction between temporal states also allows for a prefiltering of the join beforehand, which makes it less complex and more efficient at the end. And thus,

we can show that unfolding with the STARQL engine and its non reified temporal semantics is potentially more efficient than other implementations using reified semantics (e.g; SPARQLStream). This finding also supports hypothesis **H6**.

Finally, with hypothesis **H7** we require that both the query rewriting and the translation of results from the backend system is possible with minimal delay. Both translation directions were shown to be efficient, and lie in the range of milliseconds for the given queries and results.

We even were able to outperform SPARQLStream in the case of query result transformations by approximately 50%.

6.3. Evaluation of Query Execution

We already mentioned above that a performance evaluation of the STARQL framework in a competitive benchmark cannot be seen as a desirable task. The performance benchmarks that we have described in Section 2.6.2 aim on systems which rely on their own query engine. STARQL uses an ontology based data access approach for relational backends (similar to SPARQLStream) and thus, can only provide an API for plugging in different query adapters and DSMSs as described in Chapter 5. A performance evaluation of relational systems is out of scope of this work, but see, e.g; [19] for a comparison of Spark to other systems.

Nevertheless, with STARQL we introduce a framework that also supports processing of historically recorded timeseries data. Therefore, we would like to show how the chosen example queries behave with different window parameters in the following section. Additionally, we show how those queries can be optimized for parallel execution in a scalable setting.

6.3.1. Evaluation of Historical Queries

We first evaluate our example queries from Section 5.2.4 on a single machine with different window parameters.

Similarly to our previous experiments, we have chosen a machine with i7 2.8 GHz core and 8 GB of RAM. Further, we have evaluated queries **Q1** to **Q4** on four different synthetic data sets with sensors emitting one value per minute: *(i)* one sensor over three days, *(ii)* 20 sensors over three days, *(iii)* one sensor over three years, *(iv)* 20 sensors over three years (more than 31 million tuples). Their results are shown in Table 6.6.

Table 6.6.: Query times for different examples and parameters

	Postgres		**Spark**	
	3 Days	**3 Years**	**3 Days**	**3 Years**
Q1 (1 Sens.)	0.2 s	22 s	1 m 22 s	8 m 51 s
Q1 (20 Sens.)	3.1 s	6 m 47 s	3 m 5 s	14 m 8 s
Q2 (1 Sens.)	3 s	10 m 51 s	1 m 45 s	17 m 37 s
Q2 (20 Sens.)	1 m 17 s	**out of memory**	2 m 39 s	2 h 41 m 58 s
Q3 (1 Sens.)	0.6 s	2 m 54 s	3 m 10 s	15 m 48 s
Q3 (20 Sens.)	12.4 s	1 h 13 m	6 m 10 s	20 m 16 s
Q4 (1 Sens.)	13 s	N/A	14 m 20 s	N/A
Q4 (20 Sens.)	7 m 47 s	N/A	44 m 5s	N/A

One can see in the results that measured time scales are roughly linear in the PostgreSQL case with respect to the number of sensors and the used window size as well, by a factor of about 20 for sensors and less than 300 for the two time periods as expected. Besides that, there are two special cases. The case **Q4** does not scale by the expected factor of 20, but a factor of 40. This is because of the two cascaded streams that we use in the query, where we have to build the window structure twice in a row, which can not be parallelized by the Postgres engine. In the case of the 3 years dataset **Q4** took too long and we even canceled the execution of experiments for the largest dataset after five hours.

The second observation is about the memory consumption of query **Q2**, which is the only query that compares two different abstract time points that are defined on the temporal sequence. Therefore, the unfolding result tries to join the temporal relational data of both states on the backend system. The join between two tables, with a couple of gigabytes each, was so memory consuming that it could not be handled by the machine used in the experiments.

These initial observations for PostgreSQL show that the set up of the system scales vertically with respect to a growing dataset, but direct horizontal scaling with the PostgreSQL system is not possible, due to the Postgres query execution process, which can only run on a single core.

Furthermore, we compare the Postgres implementation to a STARQL implementation with Spark engine backend on the same machine with four installed worker

Table 6.7.: Query times for distributed window execution

Query	**Postgres** (1 core)	**Postgres** (2 cores)	**Postgres** (3 cores)	**Postgres** (4 cores)
Q2 (1 Sens.) / 3 Days	1.5 s	0.9 s	0.6 s	0.5 s
Q2 (20 Sens.) / 3 Days	12.8 s	7.7 s	6.5 s	5.6 s
Q2 (1 Sens.) / 3 Years	7 m 16 s	3 m 36 s	2 m 48 s	2 m 14 s
Q2 (20 Sens.) / 3 Years	1 h 25 m	57 m	52 m	43 m

units. The execution on Spark shows its design for a huge datasets, while PostgreSQL is found to be fast for small datasets and small joins, Spark shows its power in handling larger data sets and regarding larger joins without any memory problems. Although the execution of a Spark jobs produces overhead itself and additional overhead for the transformation between RDDs and Dataframes (see Section 2.2.11), we see an improved scaling with respect to the increasing dataset that is based on the parallel execution on different cores. In most cases the scaling from one to twenty sensors and three days to three years is just a one digit factor. The overhead for Spark jobs could be even further reduced by fixing different system parameters, e.g; for partitioning and shuffling data between the different worker units as we simply used the standard Spark configuration parameters for our proof of concept implementation.

6.3.2. Scalability of Query Execution

In Chapter 5 we described PostgreSQL as a process based system that cannot execute multi threading. Therefore, although we can scale vertically, we have not shown a possible horizontal scaling with parallel execution of windows in different threads. Additionally, the shown approach already has disadvantages in memory usage if different temporal states are compared in a sequence.

To solve these problems, we also implemented a second approach that is described in Section 5.3.1 and can be seen as a proof-of-concept implementation for the distributed execution of single windows. For our experiments (results shown in Table 6.7) we have chosen once again example **Q2**, because we would like to evaluate its computation time and memory consumption in comparison to the previous experiments.

The resulting table (see Table 6.7) presents values for different numbers of PostgreSQL sessions (each executed on a single core) on the machine with four cores. One can see that the distributed window approach even outperforms the non distributed approach on a single core. The reason is simple, because now every window

is executed for its own on a smaller piece of data and thus, the necessary joins in each case are reduced massively.

Furthermore, the approach proves that STARQL queries are also scalable in a horizontal direction by adding more cores or machines. Although the queries executed on a single multi core machine with shared memory, the execution time is nearly halved by adding a second CPU core (session). The addition of further cores has a measurable, but reduced effect, because the distribution overhead and splitting of windows was also managed by the same machine, and requires a CPU core.

While this experiment shows promising results on a PostgreSQL server, it is not applicable in the same way to a Spark engine because of the overhead for each Spark job. Mentioning that we execute each window in a new session or job, we soon arrive at millions of jobs, which currently cannot be handled by Spark, but are handled well in distributed sessions with PostgreSQL. An implementation of manually distributed Spark processing would require further fine tuning of Spark execution parameters and system variables, which is out of scope for this work.

Finally, we are also able to support hypothesis **H8**, as we showed a horizontal scalability of our general implementation with Postgres and are able to translate queries to big data engines such as Spark (although not efficiently in the case of **Q2**).

6.4. Discussion of Evaluation Results

This chapter presented experimental results for supporting the eight hypothesis from the beginning of our thesis.

We proceed by going through each hypothesis once again and discuss their status with respect to the evaluation results given in this chapter. Therefore, we split the hypothesis into three groups and start by functionalities, succeed with the query transformation and conclude by a discussion on efficient scalability with different backend systems.

6.4.1. Evaluation of Functionalities

An overview on functionalities and operators of STARQL was first presented in Chapter 3. The list has been compared to other RDF-Stream query languages and systems in Table 6.2 and 6.2, While its results are already discussed in Section 6.1.4, they can help us to check the following hypothesis:

H1. *We support the basic functionalities of SPARQL, as well as operators from SPARQL 1.1 in our query language.*

A comparison between STARQL and SPARQL (especially SPARQL 1.1) with respect to their query operators can be found in Section 3.3.4. It shows that all important operators are also provided in STARQL, with a few exceptions: the SRBenchmark has shown that `ASK` queries, `IF` clauses, as well as `Property Paths` are currently not supported or supported in a different way by STARQL. While `ASK` queries can basically be replaced by other query forms and `IF` clauses by cascaded selection streams, the absence of `Property Paths` can be seen as a small disadvantage when it comes to the access of RDF data in industrial measurement scenarios.

Although other competitors provide operators for property paths, we decided to omit them. The expressivity of such queries would be required to be extended to $\mathcal{ELHIO}$, which is no longer in the range of a perfect rewriting approach and therefore, could lead to incomplete and incorrect results (see Section 6.1.4).

H2. *We guarantee stream access on different kinds of temporal and non temporal input, namely, live streamed, historically recorded and static data.*

The STARQL framework provides different (window) operators for accessing all three kinds of data as presented in Section 3.2 and in our experimental results from the current chapter. Additionally, we can formulate exclusive static data access in the `WHERE` clause and temporal access in the `HAVING` clause. Thus, we can state these hypothesis as directly supported. Nevertheless, the access on different data inputs is highly dependent on support in the specific backend implementation, while, e.g., Exareme allows for live and historic data, the implementation of PostgreSQL only processes recorded temporal data.

H3. *Different kinds of streams (live and historic) can be joined and its output synchronized, if different slide parameters are provided.*

All possible inputs can be declared in the `FROM` clause and are automatically joined at the backend. This functionality is also supported by a pulse operator with *start*/*end* parameters, as well as a sampling frequency for synchronizing the output. Further, we provide a *lag* parameter for each window description, which allows a mix of live and historic data (see Section 3.2.2).

H4. *We allow basic reasoning in the range of DL-Lite$_{\mathcal{A}}$ and semantic enrichment for* STARQL.

The heart of our reasoning solution is the extension of rewriting approach of the Ontop module that was rearranged with temporal parameters for our architecture to allow a non-reified temporal layer (see Chapter5). As already mentioned in connection to **H1**, the OBDA approach allows for an expressivity of DL-Lite$_{\mathcal{A}}$ in connection with perfect rewriting. This a general limitation of the OBDA approach and was not to be solved by our implementation. Therefore, the hypothesis can also be seen as supported.

6.4.2. Feasibility of the OBDA Approach

In connection to **H4**, which is already supported, we will now check hypothesis **H5** to **H7** for a more detailed evaluation of the rewriting process (see also Section 6.2).

H5. *We are able to handle our OBDA approach, while using sensor ontologies such as the SSN ontologies.*

The rewriting of the SSN ontology with respect to R2RML mappings and the STARQL framework was shown in Section 6.2. Although we represent single tuples on the relational database side with a large number of triples, its transformation does not lead to unnecessary joins, due to optimization techniques of the transformation module. Thus, this hypothesis can also be stated as supported.

H6. *We are able to rewrite the used the extended temporal sequences and operations without any loss of efficiency on the backend system.*

The rewriting and translation for sequences of temporal states was an important goal of this thesis and is implemented in the STARQL prototype. We evaluate and discuss the transformation in Section 6.2.1 and show that unfolding is not only comparable to an reified temporal approach, but also more efficient in some cases if used together with filtering.

H7. *We are able to translate the continuous relational query results of high throughput streams without delays (or at least not relevant delays).*

We have shown our experimental results for translating query answers with respect to mappings in Figure 6.1, as well as temporal measurements for the query rewriting process in Table 6.5. In both cases we see that the delay is in an acceptable range. While query rewriting for our examples takes only up to 350ms (executed once per query), the translation of query answers took up to 40 ms in the case of 1000 tuples per second, which makes us confident that we are able to handle up to 25,000 tuples transfered to a client per second on a single machine until a user would recognize a one second delay.

6.4.3. Efficiency and Scalability of the Implemented Approach

Finally, we discuss our last hypothesis based on backend efficiency and scalability.

H8. *Our approach can be used in parallel and horizontally scalable with relational data stream management systems as backends in big data scenarios.*

The scalability of STARQL queries for the historical approach was measured on distributed PostgreSQL sessions and the big data engine Spark, and realized by implementing a parallel distributed window execution. Our results show that the addition of further CPU cores scaled nearly linear, while also processing the data distribution on the same machine. These observations make us confident that the currently implemented architecture is also applicable for big data scenarios with even larger volumes of data than the 20 sensors and measurements shown over three years.

7. Conclusion

The presented work in this thesis constitutes a major breakthrough in research on temporal ontology-based data access. We support this claim by designing a domain independent streamed temporal query language based on advanced query transformation techniques.

We started in Chapter 3 by formulating three research problems: *(i)* How do we design a state-based query language for RDF streams and temporal RDF data?, *(ii)* How do we support the OBDA translation?, *(iii)* How do we enable parallel execution for streaming and temporal data for big data solutions?

We addressed those issues throughout the thesis by presenting the STARQL framework. In Chapter 3 we also introduced a query language on RDF data with operators for accessing streams as well as temporal data and explained how we apply an OBDA approach with rewriting and query transformations for full access to different relational DSMSs and DBMSs in Chapter 4. We continued be giving a complete architecture description and implementation of the STARQL prototype together with query, mapping and data examples. Finally, we evaluated our approach in experiments with respect to functionality, feasibility of rewriting and efficiency of translated queries executed on backend systems.

In recent years several concurrent query languages for RDF streams have been developed. Although they all allow access on streams with SPARQL-based languages, those approaches also miss several important features. They either do not provide real temporal semantics , do not provide important operators for describing temporal patterns, or even omit window operators.

Additionally, most of these systems implement their own query engine into the system and do not provide flexible access on external streaming systems, with only one exception. SPARQLstream supports a query rewriting approach on relational streaming backends. Though it benefits from the flexibility of using modern and optimized systems, it provides only a reified temporal approach and no access to historical data.

Therefore, we are confident in adding important research results to the RDF streaming community, by having unified different temporal operators in one single query

language (e.g. sequence operator, window operators and operators from SPARQL 1.1), added temporal semantics with respect to ontologies and mappings, flexible backend access on relational streaming and non streaming systems,comparable to the Lambda Architecture.

Furthermore, we have verified eight research hypotheses, formulated at the beginning of our work, in Section 6.4. Our detailed contributions are given in following section.

7.1. Contributions

We provide contributions in terms of new ways for accessing RDF stream and historic data on relational backends, by a design of the STARQL framework. The detailed contributions associated to the described research problems can be listed in four parts:

- We have formulated the **new query language STARQL** with syntax and semantics. It was first presented in Chapter 3 and can be seen as a combination of elements from SPARQL 1.1, different SPARQL streaming extensions and additional concepts for temporal based analysis. While targeting for flexible access to different relational streaming and non-streaming engines, it also allows for a combined approach on historically recorded and live stream data as also given in lambda architectures (depending on the backend).
- We have extended the classical query rewriting approach by **a rewriting of temporal states and sequences**. The translation is based on mappings for hiding the heterogeneity of different data schemes on backends behind modern ontologies for sensor networks (e.g. the SSN ontology). Further, the translation was enriched by a fourth temporal dimension on top of temporal concepts of the ontology, which is a direct part of the query semantics and was shown in Chapter 4.
- We **provided an architectural design** for accessing historical and stream data by an OBDA approach. The architecture, as shown in Chapter 5, consists basically of two parts: *(i)* the rewriting and transformation module, which parses the query and unfolds the STARQL `HAVING` clause into relational expressions that are representations of the given temporal states, and *(ii)*, an implementation of different query adapters for accessing relational backends and integrating their specific window operators and constructions. The implementation was described for four backends: PostgreSQL, Exareme,

Spark and PipelineDB and was also used in the FP7 EU project Optique with an Exareme engine.

- We **evaluated our implementation** and compared it to other approaches on the market. The evaluation, as given in Chapter 6, has shown that our approach combines many positive features and approaches of other designs, while it adds a real temporal layer on top of the ontology and provides combined access to historic and streamed data. Furthermore, we have shown that the approach scales if executed on modern big data solutions such as Spark and can be massively accelerated by the use of more than one processing core or machine.

7.2. Outlook and Future Work

The approach for RDF stream access to relational sources already contributes to the work in this field and has been one of the major building blocks of the Optique project. However, there are still open issues to be addressed.

The evaluation of the SRBenchmark has shown that still several features are missing in the STARQL framework and query language, those could be at least partially implemented in the future, especially features of SPARQL 1.1 such as different kinds of property paths that do not require recursion (e.g., path alternatives) or additional functions such as `IF` clauses or regular expressions.

Furthermore, we might want to add functionalities that are naturally supported by the attached backend system, but not currently implemented in the query language. The Exareme system provides a large number of analytical Python functions (e.gfor correlation), which can be exposed to an front end RDF query language (i.e; STARQL) as additional aggregation functions. Many of these Exareme-specific functions are already supported by STARQL. However, with respect to the used underlying system, also more advanced functionalities for statistical or data analysis could be supported or even integrated into an extension of the used DL-Lite$_{\mathcal{A}}$ ontology language.

One first approach can be found in [130] and [135], where an extension of the OBDA approach for streams was shown that leads to a more analytics, source and cost-aware system. The authors presented their system based on three components: *(i)* the ontology language $\textit{DL-Lite}_{\mathcal{A}}^{agg}$, which extends DL-Lite$_{\mathcal{A}}$ with aggregations as first class citizens and a corresponding mapping language with STARQL constructs, *(ii)* the STARQL framework for accessing streams and static data and *(iii)*

a highly optimized version of Exareme for streams (i.e., ExaStream) capable of handling complex streaming and static queries with source and cost aware optimization techniques. The architecture can be seen as a first step towards the development of an fully-fledged and analytics aware OBDA systems with access on static relational and streaming data.

Another interesting research field of ontology based approaches considers the field of federation, which is already a part of SPARQL 1.1 and first work has been seen done in the sensor network community (e.g. in [95]). Nevertheless it is not implemented in OBDA streaming systems yet. Say, one would like to refer to a stream that uses data with respect to a static dataset which is distributed over several relational databases. This exposes a need for references to different streaming or static data servers directly within the query language. Furthermore, we could think about an ecosystem of sensing devices, which offers an API for data access through a web interface as also used in the architectural implementation shown in Chapter 5.

Moreover, we can think of an extension of the overall reasoning capabilities of the system in various directions. Current approaches of RDF-streaming engines only allow for inexpressive reasoning. Since OBDA and perfect rewriting approaches are limited to DL-Lite. Also systems based on their own RDF stream processing language use limited reasoning. First work on extending the expressiveness in the case of query rewriting for streams has been presented in [59] for CQELS (see also Section 2.5.3), while the same drawbacks remain as discussed in Section 6.1.4. Other current research has also started investigations on stream reasoning for non-rewriting approaches, but they have focused on a materialization of ontological axioms from streams as, e.g., in [31], [197] or [155].

However, as all these approaches use a reified temporal version of RDF streams, we can hardly say that they use a real temporal OBDA approach in our sense. The authors of [25] propose a query language *TCQ* with built in LTL operators, while [22] also investigates on rewritability problems of temporal queries and ontologies. Nevertheless, we have already mentioned in 3.3.5 that STARQL `HAVING` clauses are able to express rewritable TCQ queries. Therefore, we could invest more effort in the enrichment of the query language with temporal constructs and operators as given above or even try to extend the ontology language with similar constructs for expressing temporal states. However, some ontology standards for temporal concepts already exist (see [236]).

Finally, the different approaches for RDF-streaming query languages create a reasonable demand for a standardized query language. Such a process was started with the query language RSP-QL (see Section 2.5.7), but it is still an ongoing process, and the insight of STARQL offers a lot of opportunities for further research in the community.

Appendices

A. Transformation of Example Queries

Listing A.1: Transformation results - example Q1

```
1  CREATE VIEW S_out_having AS
2  SELECT DISTINCT wid, _sens, _val
3  FROM
4   ( SELECT * FROM
5     ( SELECT * FROM
6        (  SELECT sens AS _sens FROM (
7           SELECT DISTINCT
8              ('http://www.sensor.net/' || qview1."sid") AS "sens"
9            FROM
10          sensormetadata qview1
11          WHERE
12          (qview1."property" = 'Temperature') AND
13          (qview1."location" = 'GasTurbine2103/01') AND
14          qview1."sid" IS NOT NULL
15          ) SUB_QVIEW
16       ) SUB_TRIPLE1
17    ) SUB_WHERE
18    NATURAL JOIN
19     (SELECT wid, _sens, _obs1, _val FROM (
20       SELECT * FROM(
21          SELECT * FROM(
22             SELECT wid, abox AS i, obs1 AS _obs1, val AS _val, z AS _z, sens
                    AS _sens FROM (
23             SELECT DISTINCT  qview1.wid,  qview1.abox,
24                ('http://www.sensor.net/result/[...]qview1."value") AS "z",
25                ('http://www.sensor.net/[...]qview1."sid") AS "sens",
26                ('http://www.sensor.net/observation/[...]qview1."value") AS "
                      obs1",
27                qview1."value" AS "val"
28              FROM
29             Measurement_public_stream qview1
30             WHERE
31             qview1."timestamp" IS NOT NULL AND
32             qview1."sid" IS NOT NULL AND
33             qview1."value" IS NOT NULL
34             ) SUB_QVIEW
35             ) SUB_TRIPLE0
36          )SUB WHERE _val > 41
37       ) SUB_QVIEW
38    ) SUB_HAVING
39 ) SUB_FROM;
```

Listing A.2: Transformation results - example Q2

```
1  CREATE VIEW S_out_having AS
2  SELECT DISTINCT wid, _sens
3  FROM
4   (SELECT * FROM
5      (SELECT wid, _z2, _sens, _z, _obs1, _obs2 FROM
6          (SELECT wid, _z, _z2, _sens, _obs1, _obs2 FROM
7             (SELECT * FROM(
8                SELECT wid, abox AS j, y AS _y, z2 AS _z2, sens AS _sens,
                     obs2 AS _obs2 FROM (
9                SELECT DISTINCT  qview1.wid,  qview1.abox,
10                  qview1."value" AS "y",
11                  ('http://www.sensor.net/result/[...]qview1."value")
12                     AS "z2",
13                  ('http://www.sensor.net/' || qview1."sid") AS "sens",
14                  ('http://www.sensor.net/observation/[...]qview1."value")
15                     AS "obs2"
16                 FROM measurement qview1
17               WHERE
18               qview1."timestamp" IS NOT NULL AND
19               qview1."sid" IS NOT NULL AND
20               qview1."value" IS NOT NULL
21               ) SUB_QVIEW
22               ) SUB_TRIPLE0
23            )SUBJOIN1
24            NATURAL JOIN
25            (SELECT * FROM(
26               SELECT wid, abox AS i, obs1 AS _obs1, z AS _z, x AS _x, sens
                     AS _sens FROM (
27               SELECT DISTINCT  qview1.wid,  qview1.abox,
28                  ('http://www.sensor.net/result/[...]qview1."value")
29                     AS "z",
30                  qview1."value" AS "x",
31                  ('http://www.sensor.net/' || qview1."sid") AS "sens",
32                  ('http://www.sensor.net/observation/[...]qview1."value")
33                     AS "obs1"
34                FROM measurement qview1
35               WHERE
36               qview1."timestamp" IS NOT NULL AND
37               qview1."sid" IS NOT NULL AND
38               qview1."value" IS NOT NULL
39               ) SUB_QVIEW
40               ) SUB_TRIPLE1
41            )SUBJOIN2
42         ) SUB_QVIEW
43         EXCEPT
44         SELECT wid, _z, _z2, _sens, _obs1, _obs2 FROM (
45            SELECT * FROM(
46               SELECT * FROM(
47                  (SELECT * FROM(
48                     SELECT wid, abox AS j, y AS _y, z2 AS _z2, sens AS
                           _sens, obs2 AS _obs2 FROM (
49                     SELECT DISTINCT  qview1.wid,  qview1.abox,
50                        qview1."value" AS "y",
51                        ('http://www.sensor.net/result/[...]qview1."value")
52                           AS "z2",
```

```
53                          ('http://www.sensor.net/'||qview1."sid") AS "sens",
54                          ('http://www.sensor.net/obs/[...]qview1."value")
55                             AS "obs2"
56                       FROM measurement qview1
57                       WHERE
58                       qview1."timestamp" IS NOT NULL AND
59                       qview1."sid" IS NOT NULL AND
60                       qview1."value" IS NOT NULL
61                       ) SUB_QVIEW
62                       ) SUB_TRIPLE2
63                    )SUBJOIN1
64                    NATURAL JOIN
65                    (SELECT * FROM(
66                       SELECT wid, abox AS i, obs1 AS _obs1, z AS _z, x AS _x,
                              sens AS _sens FROM (
67                       SELECT DISTINCT  qview1.wid,  qview1.abox,
68                          ('http://www.sensor.net/result/[...]qview1."value")
69                             AS "z",
70                          qview1."value" AS "x",
71                          ('http://www.sensor.net/'||qview1."sid") AS "sens",
72                          ('http://www.sensor.net/obs/[...]qview1."value")
73                             AS "obs1"
74                       FROM measurement qview1
75                       WHERE
76                       qview1."timestamp" IS NOT NULL AND
77                       qview1."sid" IS NOT NULL AND
78                       qview1."value" IS NOT NULL
79                       ) SUB_QVIEW
80                       ) SUB_TRIPLE3
81                    )SUBJOIN2
82                 )SUB WHERE i < j
83              )SUB WHERE _x > _y
84           ) SUB_QVIEW
85        )SUB_EXCEPT
86     ) SUB_HAVING;
```

Listing A.3: Transformation results - example Q3

```
1  CREATE VIEW S_out_having AS
2  SELECT DISTINCT wid, _sens, _x
3  FROM
4   ( SELECT * FROM
5      (SELECT wid, _z, _x, _sens, _obs1 FROM (
6           SELECT * FROM(
7              SELECT wid, abox AS i, obs1 AS _obs1, z AS _z, x AS _x,
8                 sens AS _sens FROM (
9              SELECT DISTINCT  qview1.wid,  qview1.abox,
10                 ('http://www.sensor.net/result/[...]qview1."value") AS "z",
11                 qview1."value" AS "x",
12                ('http://www.sensor.net/' || qview1."sid") AS "sens",
13                ('http://www.sensor.net/observation/[...]qview1."value")
14                   AS "obs1"
15             FROM
16             Measurement_public_stream qview1
17             WHERE
```

```
18              qview1."timestamp" IS NOT NULL AND
19              qview1."sid" IS NOT NULL AND
20              qview1."value" IS NOT NULL
21              ) SUB_QVIEW
22              ) SUB_TRIPLE0
23           ) SUB_QVIEW
24     ) SUB_HAVING
25 ) SUB_FROM;
26
27 CREATE VIEW S_out_agg_MAX_x AS
28 SELECT wid, MAX(_x::numeric) as _agg_MAX_x, _sens
29 FROM S_out_having
30  GROUP BY wid, _sens;
31
32 CREATE VIEW S_out_agg_AVG_x AS
33 SELECT wid, AVG(_x::numeric) as _agg_AVG_x, _sens
34 FROM S_out_having
35  GROUP BY wid, _sens;
36
37 CREATE VIEW S_out_tJoin AS
38 SELECT DISTINCT *
39 FROM S_out_having NATURAL JOIN S_out_agg_AVG_x
40 NATURAL JOIN S_out_agg_MAX_x;
41
42 CREATE VIEW S_out_starqlout AS
43 SELECT DISTINCT time AS timestamp, _sens, _agg_MAX_x AS _max,
44    _agg_AVG_x AS _avg, (_agg_MAX_x - _agg_AVG_x) AS _diff
45 FROM S_out_tJoin
46 WHERE (_agg_AVG_x + 6) < _agg_MAX_x;
47
48 SELECT * FROM S_out_starqlout;
```

Listing A.4: Transformation results - example Q4

```
1 CREATE VIEW S_out_avg_having AS
2 SELECT DISTINCT wid, _sens, _x
3 FROM
4  ( SELECT * FROM
5     (SELECT wid, _z, _x, _sens, _obs1 FROM (
6          SELECT * FROM(
7             SELECT wid, abox AS i, obs1 AS _obs1, z AS _z, x AS _x,
8                sens AS _sens FROM (
9             SELECT DISTINCT  qview1.wid,  qview1.abox,
10               ('http://www.sensor.net/result/[...]qview1."value") AS "z",
11               qview1."value" AS "x",
12               ('http://www.sensor.net/' || qview1."sid") AS "sens",
13               ('http://www.sensor.net/observation/[...]qview1."value")
14                  AS "obs1"
15              FROM Measurement qview1
16             WHERE
17             qview1."timestamp" IS NOT NULL AND
18             qview1."sid" IS NOT NULL AND
19             qview1."value" IS NOT NULL
20             ) SUB_QVIEW
21             ) SUB_TRIPLE0
```

```
22          ) SUB_QVIEW
23    ) SUB_HAVING
24 ) SUB_FROM;
25
26 CREATE VIEW S_out_avg_agg_AVG_x AS
27 SELECT wid, _sens, AVG(_x::numeric) as _agg_AVG_x
28 FROM S_out_avg_having
29  GROUP BY wid, _sens;
30
31 CREATE VIEW S_out_avg_tJoin AS
32 SELECT DISTINCT *
33 FROM S_out_avg_having NATURAL JOIN S_out_avg_agg_AVG_x;
34
35 CREATE VIEW S_out_avg_starqlout AS
36 SELECT DISTINCT time AS timestamp, _sens AS Subject, ':hasAVG' AS Predicate,
37    _agg_AVG_x AS Object  FROM S_out_avg_tJoin;
38
39 CREATE VIEW S_out_avg_starqlout_stream AS
40 [Window generation of S_out_avg_starqlout]
41
42 CREATE VIEW S_out_max_strminfo AS
43 SELECT * FROM S_out_avg_starqlout_stream;
44
45
46 CREATE VIEW S_out_max_having AS
47 SELECT DISTINCT wid, _sens, _x
48 FROM
49  (SELECT * FROM
50     (SELECT wid, _x, _sens FROM (
51          SELECT * FROM(
52             SELECT * FROM(
53                 SELECT DISTINCT
54                 wid, Object AS _x, Subject AS _sens
55                 FROM S_out_max_strminfo
56                 WHERE (S_out_max_strminfo.Predicate = ':hasAVG')
57             ) SUB_NONMAPPED1
58             ) SUB_TRIPLE0
59          ) SUB_QVIEW
60    ) SUB_HAVING
61 ) SUB_FROM;
62
63 CREATE VIEW S_out_max_agg_MAX_x AS
64 SELECT wid, MAX(_x::numeric) as _agg_MAX_x, _sens
65 FROM S_out_max_having
66  GROUP BY wid, _sens;
67
68
69 CREATE VIEW S_out_max_tJoin AS
70 SELECT DISTINCT *
71 FROM S_out_max_having
72 NATURAL JOIN S_out_max_agg_MAX_x;
73
74 CREATE VIEW S_out_max_starqlout AS
75 SELECT DISTINCT time AS timestamp, _sens, _agg_MAX_x FROM S_out_max_tJoin;
76
77
78 SELECT * FROM S_out_max_starqlout;
```

B. Distributed Window execution with pl/pgSQL

Listing B.1: Distributed window implementation - Client

```
1  --CREATE EXTENSION dblink;
2
3  DROP TYPE my_hasval CASCADE;
4  CREATE TYPE my_hasval AS (
5      WID bigint,
6      ABOX bigint,
7      "timestamp" timestamp,
8      sensor integer,
9      VALUE numeric(12,3)
10    );
11 DROP TYPE my_starqlout CASCADE;
12 CREATE TYPE my_starqlout AS (
13    WID bigint,
14    subject varchar,
15    predicate varchar,
16    Object varchar  );
17
18 DROP TYPE my_sout CASCADE;
19 CREATE TYPE my_sout AS (
20    WID bigint,
21    _sens text
22   );
23
24
25 CREATE OR REPLACE FUNCTION distribute(wid BigInt, con int)
26    RETURNS VOID
27 AS $$
28 DECLARE
29 sql text;
30 conn text;
31 BEGIN
32     conn := 'conn_' || con;
33     sql := 'SELECT dblink_send_query(' || QUOTE_LITERAL(conn) || ',' ||
           QUOTE_LITERAL('SELECT * FROM start_eval(' || wid || ');') || ');';
34     RAISE NOTICE 'SELECT dblink_send_query(%,%);', QUOTE_LITERAL(conn),
           QUOTE_LITERAL('SELECT * FROM start_eval(' || wid || ');');
35     execute sql;
36 END
37 $$
38 LANGUAGE plpgsql;
39
```

```
---connection management
CREATE OR REPLACE FUNCTION stream(wids BigInt, cons Int)
   RETURNS SETOF my_starqlout
AS $$
DECLARE
sout my_starqlout;
conn text;
sql text;
host text;
port text;
db text;
usr text;
pw text;
send BOOLEAN;
status integer;
dispatch_error text;
BEGIN
  host := 'localhost'; --host := '141.83.117.124';
  port := '5432';
  db := 'publicdatadiss'; --- host connections
  usr := 'postgres';
  pw := 'postgres';
  FOR i IN 1..cons LOOP
   conn := 'conn_' || i;
   RAISE NOTICE 'connect %', conn;
   sql := 'SELECT dblink_connect(' || QUOTE_LITERAL(conn) || ',' ||
       QUOTE_LITERAL('host=' || host || ' port=' || port ||  ' dbname=' || db
        || ' user=' || usr || ' password=' || pw) || ');';
   RAISE NOTICE 'connected %', conn;
   execute sql;
  END LOOP;
  send := FALSE;
  FOR i IN 0..wids LOOP --- loop through windows
   RAISE NOTICE 'trying to send wid %', i;
   WHILE send != TRUE LOOP
      FOR con IN 1..cons LOOP
         conn := 'conn_' || con;
         sql := 'SELECT dblink_is_busy(' || QUOTE_LITERAL(conn) || ');';
         execute sql into status;
         RAISE NOTICE 'Received status % from con %', status, con;
         IF status = 0 THEN --- con not busy
                 status := 1; --- set to busy
                 FOR sout IN --- receive results
               SELECT * FROM dblink_get_result( conn ) AS my_starqlout(WID
                   bigint, subject varchar, predicate varchar, Object varchar
                   )
            LOOP     --- gebe results aus
               RAISE NOTICE 'Received % from con %', sout, con;
               RETURN NEXT sout;
            END LOOP;
            RAISE NOTICE 'Con % is ready', con;
            FOR sout IN
               SELECT * FROM dblink_get_result( conn ) AS my_starqlout(WID
                   bigint, subject varchar, predicate varchar, Object varchar
                   )
            LOOP
               RAISE NOTICE 'Received % from con %', sout, con;
               RETURN NEXT sout;
```

```
92             END LOOP;
93             PERFORM distribute(i, con); --- send wid
94             RAISE NOTICE 'i have send wid % to con %', i, con;
95             send := TRUE;
96             EXIT;
97          ELSE
98              RAISE NOTICE 'Connection % not ready. Status is %', conn,
                    status;
99              status := 0;
100             sql := 'SELECT dblink_error_message(' || QUOTE_LITERAL(conn)
                    || ');';
101             execute sql into dispatch_error;
102             if dispatch_error <> 'OK' THEN --- If not 'ok', show error
103               RAISE 'Error: %', dispatch_error;
104            end if;
105         END if;
106      END LOOP;
107      RAISE NOTICE 'next try';
108      --sql := 'select pg_sleep(0.1)';
109      --execute sql;
110   END LOOP;
111   send := FALSE;
112  END LOOP;
113  --- collect results
114  FOR i IN 1..cons LOOP
115    status := 1;
116    RAISE NOTICE 'Waiting till connection % has finished', i;
117    conn := 'conn_' || i;
118    WHILE (status = 1)
119    LOOP
120   sql := 'SELECT dblink_is_busy(' || QUOTE_LITERAL(conn) || ');';
121   execute sql into status;
122    END LOOP;
123    FOR sout IN
124    SELECT * FROM dblink_get_result( conn ) AS my_starqlout(WID bigint,
           subject varchar, predicate varchar, Object varchar)
125    LOOP
126   RAISE NOTICE 'Received % from con %', sout, i;
127   RETURN NEXT sout;
128    END LOOP;
129    sql := 'SELECT dblink_disconnect(' || QUOTE_LITERAL(conn) || ');';
130    execute sql;
131    RAISE NOTICE 'Connection % has finished', i;
132  END LOOP;
133  RETURN;
134 END;
135 $$
136 LANGUAGE 'plpgsql';
137
138 CREATE OR REPLACE FUNCTION disconnect(cons Int)
139   RETURNS VOID
140 AS $$
141 DECLARE
142 conn text;
143 sql text;
144 BEGIN
145  FOR i IN 1..cons LOOP
146    conn := 'conn_' || i;
```

```
147    RAISE NOTICE 'disconnect %', conn;
148    sql := 'SELECT dblink_disconnect(' || QUOTE_LITERAL(conn) || ');';
149    RAISE NOTICE 'disconnected %', conn;
150    execute sql;
151  END LOOP;
152 END;
153 $$
154 LANGUAGE 'plpgsql';
155
156 CREATE OR REPLACE FUNCTION disconnect(cons Int)
157    RETURNS VOID
158 AS $$
159 DECLARE
160 conn text;
161 sql text;
162 BEGIN
163   FOR i IN 1..cons LOOP
164    conn := 'conn_' || i;
165    RAISE NOTICE 'connect %', conn;
166    sql := 'SELECT dblink_disconnect(' || QUOTE_LITERAL(conn) || ');';
167    RAISE NOTICE 'connected %', conn;
168    execute sql;
169   END LOOP;
170 END;
171 $$
172 LANGUAGE 'plpgsql';
173
174 select distinct * from stream(457,2) order by wid; ---Params: number of wIDs
        , number of cons
```

Listing B.2: Distributed window implementation - Server

```
1  DROP TYPE my_hasval CASCADE;
2  CREATE TYPE my_hasval AS (
3      WID bigint,
4      ABOX bigint,
5      "timestamp" timestamp,
6      sid integer,
7      VALUE numeric(12,3)
8     );
9  DROP TYPE my_starqlout CASCADE;
10 CREATE TYPE my_starqlout AS (
11    WID bigint,
12    subject varchar,
13    predicate varchar,
14    Object varchar  );
15
16 DROP TYPE my_sout CASCADE;
17 CREATE TYPE my_sout AS (
18    WID bigint,
19    _sens text
20   );
21
22
23
```

```
24 --------------------------
25 --SET ALL VARIABLES AND RUN
26 --------------------------
27
28 CREATE OR REPLACE VIEW win_vars AS -- Breite und slide in minutes
29 SELECT 60 as width, 60 as slide, (SELECT min(timestamp) FROM
       measurement_public3y1) as start, (SELECT max(timestamp) FROM
       measurement_public3y1)  as ende;
30
31 ---------------------------
32 --END OF VARIABLE SECTION
33 ---------------------------
34 CREATE OR REPLACE VIEW stream_vars AS
35 SELECT (SELECT width FROM win_vars), (SELECT slide FROM win_vars), (SELECT
       start FROM win_vars), (SELECT ende FROM win_vars),
36 (SELECT trunc(date_part('epoch',(SELECT max(timestamp) FROM
       measurement_public)
37    - (SELECT min(timestamp) FROM measurement_public))/((SELECT width FROM
          win_vars)*60)) :: int) as num_wid;
38
39 ------------------------------------------
40 -- TRANSFORMED HAVING BEGIN
41 ------------------------------------------
42
43 CREATE or REPLACE FUNCTION s_out_having(BigInt)
44                            RETURNS SETOF my_sout AS
45 $$
46 BEGIN
47 RETURN QUERY
48 SELECT DISTINCT wid, _sens
49 FROM
50  ( SELECT * FROM
51     ( SELECT wid, _z2, _sens, _z, _obs1, _obs2 FROM
52          (  SELECT wid, _z, _z2, _sens, _obs1, _obs2 FROM (
53                SELECT * FROM(
54                   SELECT * FROM(
55                      (
56                         SELECT * FROM(
57                            SELECT wid, abox AS j, y AS _y, z2 AS _z2, sens
                                   AS _sens, obs2 AS _obs2 FROM (
58                            SELECT DISTINCT  qview1.wid,  qview1.abox,
59                               5 AS "yQuestType", NULL AS "yLang", qview1."
                                   value" AS "y",
60                               1 AS "z2QuestType", NULL AS "z2Lang", ('http
                                   ://www.sensor.net/result/' || qview1."sid"
                                   || '/' || qview1."timestamp" || '/' ||
                                   qview1."value") AS "z2",
61                               1 AS "sensQuestType", NULL AS "sensLang", ('
                                   http://www.sensor.net/' || qview1."sid") AS
                                    "sens",
62                               1 AS "obs2QuestType", NULL AS "obs2Lang", ('
                                   http://www.sensor.net/observation/' ||
                                   qview1."sid" || '/' || qview1."timestamp"
                                   || '/' || qview1."value") AS "obs2"
63                             FROM
64                            split_measurement_public($1) qview1
65                            WHERE
66                            qview1."timestamp" IS NOT NULL AND
```

```
67                         qview1."sid" IS NOT NULL AND
68                         qview1."value" IS NOT NULL
69                         ) SUB_QVIEW
70
71                         ) SUB_TRIPLE0
72                   )SUBJOIN1
73                   NATURAL JOIN
74                   (
75                      SELECT * FROM(
76                         SELECT wid, abox AS i, obs1 AS _obs1, z AS _z, x
                               AS _x, sens AS _sens FROM (
77                         SELECT DISTINCT  qview1.wid,  qview1.abox,
78                            1 AS "zQuestType", NULL AS "zLang", ('http://
                                 www.sensor.net/result/' || qview1."sid" ||
                                 '/' || qview1."timestamp" || '/' || qview1
                                 ."value") AS "z",
79                            5 AS "xQuestType", NULL AS "xLang", qview1."
                                 value" AS "x",
80                            1 AS "sensQuestType", NULL AS "sensLang", ('
                                 http://www.sensor.net/' || qview1."sid") AS
                                  "sens",
81                            1 AS "obs1QuestType", NULL AS "obs1Lang", ('
                                 http://www.sensor.net/observation/' ||
                                 qview1."sid" || '/' || qview1."timestamp"
                                 || '/' || qview1."value") AS "obs1"
82                          FROM
83                         split_measurement_public($1) qview1
84                         WHERE
85                         qview1."timestamp" IS NOT NULL AND
86                         qview1."sid" IS NOT NULL AND
87                         qview1."value" IS NOT NULL
88                         ) SUB_QVIEW
89
90                         ) SUB_TRIPLE1
91                   )SUBJOIN2
92                )SUB
93             )SUB
94          ) SUB_QVIEW
95       EXCEPT
96          SELECT wid, _z, _z2, _sens, _obs1, _obs2 FROM (
97             SELECT * FROM(
98                SELECT * FROM(
99                   (
100                     SELECT * FROM(
101                        SELECT wid, abox AS j, y AS _y, z2 AS _z2, sens
                               AS _sens, obs2 AS _obs2 FROM (
102                        SELECT DISTINCT  qview1.wid,  qview1.abox,
103                           5 AS "yQuestType", NULL AS "yLang", qview1."
                                 value" AS "y",
104                           1 AS "z2QuestType", NULL AS "z2Lang", ('http
                                 ://www.sensor.net/result/' || qview1."sid"
                                 || '/' || qview1."timestamp" || '/' ||
                                 qview1."value") AS "z2",
105                           1 AS "sensQuestType", NULL AS "sensLang", ('
                                 http://www.sensor.net/' || qview1."sid") AS
                                  "sens",
106                           1 AS "obs2QuestType", NULL AS "obs2Lang", ('
                                 http://www.sensor.net/observation/' ||
```

```
                           qview1."sid" || '/' || qview1."timestamp"
                           || '/' || qview1."value") AS "obs2"
107                        FROM
108                        split_measurement_public($1) qview1
109                        WHERE
110                        qview1."timestamp" IS NOT NULL AND
111                        qview1."sid" IS NOT NULL AND
112                        qview1."value" IS NOT NULL
113                        ) SUB_QVIEW
114
115                        ) SUB_TRIPLE2
116                  )SUBJOIN1
117                  NATURAL JOIN
118                  (
119                     SELECT * FROM(
120                        SELECT wid, abox AS i, obs1 AS _obs1, z AS _z, x
                               AS _x, sens AS _sens FROM (
121                        SELECT DISTINCT  qview1.wid,  qview1.abox,
122                           1 AS "zQuestType", NULL AS "zLang", ('http://
                               www.sensor.net/result/' || qview1."sid" ||
                               '/' || qview1."timestamp" || '/' || qview1
                               ."value") AS "z",
123                           5 AS "xQuestType", NULL AS "xLang", qview1."
                               value" AS "x",
124                           1 AS "sensQuestType", NULL AS "sensLang", ('
                               http://www.sensor.net/' || qview1."sid") AS
                                "sens",
125                           1 AS "obs1QuestType", NULL AS "obs1Lang", ('
                               http://www.sensor.net/observation/' ||
                               qview1."sid" || '/' || qview1."timestamp"
                               || '/' || qview1."value") AS "obs1"
126                        FROM
127                        split_measurement_public($1) qview1
128                        WHERE
129                        qview1."timestamp" IS NOT NULL AND
130                        qview1."sid" IS NOT NULL AND
131                        qview1."value" IS NOT NULL
132                        ) SUB_QVIEW
133
134                        ) SUB_TRIPLE3
135                  )SUBJOIN2
136               )SUB WHERE i < j
137            )SUB WHERE _x > _y
138         ) SUB_QVIEW
139      ) SUB_EXCEPT
140   ) SUB_HAVING
141 ) SUB_FROM;
142 END
143 $$
144 LANGUAGE plpgsql;
145
146 CREATE or REPLACE FUNCTION s_out(BigInt)
147                        RETURNS SETOF my_starqlout AS
148 $$
149 BEGIN
150 RETURN QUERY
151 SELECT DISTINCT wid, _sens::varchar AS Subject, 'a'::varchar AS Predicate,
        ':RecentMonInc'::varchar AS Object  FROM S_out_having($1);
```

```
152 END
153 $$
154 LANGUAGE plpgsql;
155
156 ------------------------------------------
157 -- TRANSFORMED HAVING END
158 ------------------------------------------
159
160 CREATE OR REPLACE FUNCTION time_start(BigInt)
161    RETURNS timestamp
162 AS $$
163 DECLARE
164 Result timestamp;
165 BEGIN
166 Result := (SELECT start + $1 * slide * interval '1 minute' from stream_vars)
        ;
167 RETURN Result;
168 END $$
169 LANGUAGE plpgsql;
170
171 CREATE OR REPLACE FUNCTION time_end(BigInt)
172    RETURNS timestamp
173 AS $$
174 DECLARE
175 Result timestamp;
176 BEGIN
177 Result := (SELECT (SELECT * FROM time_start($1)) + (width - 1) * interval '1
         minute' from stream_vars);
178 RETURN Result;
179 END $$
180 LANGUAGE plpgsql;
181
182 CREATE OR REPLACE FUNCTION split_measurement_public(BigInt)
183    RETURNS SETOF my_hasval
184 AS $$
185 BEGIN
186 RETURN QUERY
187 SELECT $1 as WID, rank() OVER (ORDER BY timestamp ASC) as ABOX, *
188 FROM measurement_public3y1 where timestamp between (SELECT * FROM time_start
        ($1)) and (SELECT * FROM time_end($1));
189 END
190 $$
191 LANGUAGE plpgsql;
192
193 --- Start of STARQL evaluation
194 CREATE OR REPLACE FUNCTION start_eval(integer)
195    RETURNS SETOF my_starqlout
196 AS $$
197 DECLARE
198 sout my_starqlout;
199 BEGIN
200     RAISE NOTICE 'start eval';
201     FOR sout IN
202    SELECT * from s_out($1)
203     LOOP
204    RAISE NOTICE 'Received % for wid %', sout, $1;
205         RETURN NEXT sout;
206     END LOOP;
```

```
207 END
208 $$
209 LANGUAGE plpgsql;
210
211 select * from stream_vars;
```

C. SRBench - Queries expressed in STARQL

Listing C.5: SRBench - Q6

```
PREFIX om-owl: <http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#>
PREFIX weather: <http://knoesis.wright.edu/ssw/ont/weather.owl#>

SELECT ?sensor
FROM STREAM <http://www.cwi.nl/SRBench/observations>
[ NOW - "PT1H"^^XSD:DURATION, NOW ]-> "PT1M"^^XSD:DURATION
SEQUENCE BY StdSeq AS SEQ1
HAVING EXISTS i,j,k in SEQ1,?observation (
  GRAPH i
  { ?observation om-owl:procedure ?sensor ;
                 a weather:VisibilityObservation ;
                 om-owl:result [om-owl:floatValue ?value ] .
  }  AND ?value < 10
  OR GRAPH j
  { ?observation om-owl:procedure ?sensor ;
                 a weather:RainfallObservation ;
                 om-owl:result [om-owl:floatValue ?value ] .

  } AND ?value > 30
  OR GRAPH k
  { ?observation om-owl:procedure ?sensor ;
                 a weather:SnowfallObservation .
  }
})
```

Listing C.6: SRBench - Q7

```
PREFIX om-owl: <http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#>

SELECT ?sensor
FROM STREAM <http://www.cwi.nl/SRBench/observations>
[ NOW - "PT1H"^^XSD:DURATION, NOW ]-> "PT1M"^^XSD:DURATION
SEQUENCE BY StdSeq AS SEQ1
HAVING NOT EXISTS i in SEQ1 ( GRAPH i {
  sensor om-owlgeneratedObservation observation .
})
```

Listing C.1: SRBench - Q1

```
PREFIX om-owl: <http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#>
PREFIX weather: <http://knoesis.wright.edu/ssw/ont/weather.owl#>

SELECT DISTINCT ?sensor ?value ?uom
FROM STREAM <http://www.cwi.nl/SRBench/observations>
[ NOW - "PT1H"^^XSD:DURATION, NOW ]-> "PT1M"^^XSD:DURATION
SEQUENCE BY StdSeq AS SEQ1
HAVING EXISTS i in SEQ1 ( GRAPH i {
  ?observation om-owl:procedure ?sensor ;
               a weather:RainfallObservation ;
               om-owl:result ?result .
  ?result om-owl:floatValue ?value ;
          om-owl:uom ?uom .
})
```

Listing C.2: SRBench - Q2

```
PREFIX om-owl: <http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#>
PREFIX weather: <http://knoesis.wright.edu/ssw/ont/weather.owl#>

SELECT DISTINCT ?sensor ?value ?uom
FROM STREAM <http://www.cwi.nl/SRBench/observations>
[ NOW - "PT1H"^^XSD:DURATION, NOW ]-> "PT1M"^^XSD:DURATION
SEQUENCE BY StdSeq AS SEQ1
HAVING EXISTS i in SEQ1 ( GRAPH i {
  ?observation om-owl:procedure ?sensor ;
               a weather:PrecipitationObservation ;
               om-owl:result ?result .
  ?result ?p1 ?value .
  OPTIONAL {
    ?result ?p2 ?uom .
  }
})
```

Listing C.3: SRBench - Q4

```
PREFIX om-owl: <http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#>
PREFIX weather: <http://knoesis.wright.edu/ssw/ont/weather.owl#>

SELECT ?sensor AVG(?windSpeed) AS ?averageWindSpeed
               AVG(?temperature) AS ?averageTemperature
FROM STREAM <http://www.cwi.nl/SRBench/observations>
[ NOW - "PT1H"^^XSD:DURATION, NOW ]-> "PT10M"^^XSD:DURATION
SEQUENCE BY StdSeq AS SEQ1
HAVING EXISTS i in SEQ1 ( GRAPH i {
  ?temperatureObservation om-owl:procedure ?sensor ;
                          a weather:TemperatureObservation ;
                          om-owl:result ?temperatureResult .
  ?temperatureResult om-owl:floatValue ?temperature ;
                     om-owl:uom ?uom .
  ?windSpeedObservation om-owl:procedure ?sensor ;
                        a weather:WindSpeedObservation ;
                        om-owl:result ?windResult .
  ?windResult om-owl:floatValue ?windSpeed .
} AND ?temperature > 32)
GROUP BY ?sensor
```

Listing C.4: SRBench - Q5

```
PREFIX om-owl: <http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#>
PREFIX weather: <http://knoesis.wright.edu/ssw/ont/weather.owl#>


CONSTRUCT { _:obs a weather:Blizzard; om-owl:procedure ?sensor }
FROM STREAM <http://www.cwi.nl/SRBench/observations>
[ NOW - "PT3H"^^XSD:DURATION, NOW ]-> "PT10M"^^XSD:DURATION
SEQUENCE BY StdSeq AS SEQ1
HAVING EXISTS i in SEQ1 ( GRAPH i {
      ?sensor om-owl:generatedObservation [a weather:SnowfallObservation] ;
              om-owl:generatedObservation ?o1 ;
              om-owl:generatedObservation ?o2 .
      ?o1 a weather:TemperatureObservation ;
          om-owl:observedProperty weather:_AirTemperature ;
          om-owl:result [om-owl:floatValue ?temperature] .
      ?o2 a weather:WindObservation ;
          om-owl:observedProperty weather:_WindSpeed ;
          om-owl:result [om-owl:floatValue ?windSpeed] .
    })
GROUP BY ?sensor
HAVING AGGREGATE
    AVG(?temperature) < 32  AND  # fahrenheit
     MIN(?windSpeed) > 40.0  #milesPerHour
```

Listing C.7: SRBench - Q8

```
1 PREFIX om-owl: <http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#>
2 PREFIX weather: <http://knoesis.wright.edu/ssw/ont/weather.owl#>
3 PREFIX wgs84_pos: <http://www.w3.org/2003/01/geo/wgs84_pos>
4
5 SELECT MIN(?temperature) AS ?minTemperature  MAX(?temperature) AS ?
      maxTemperature
6 FROM STREAM <http://www.cwi.nl/SRBench/observations>,
7 STATIC <http://www.cwi.nl/SRBench/sensors>
8 [ NOW - "PT1D"^^XSD:DURATION, NOW ]-> "PT1M"^^XSD:DURATION
9 SEQUENCE BY StdSeq AS SEQ1
10 HAVING EXISTS i in SEQ1 ( GRAPH i {
11   ?sensor om-owl:processLocation ?sensorLocation ;
12           om-owl:generatedObservation ?observation .
13   ?sensorLocation wgs84_pos:alt "%Altitude%"^^xsd:float ;
14                   wgs84_pos:lat "%Latitude%"^^xsd:float ;
15                   wgs84_pos:long "%Longitude%"^^xsd:float .
16   ?observation om-owl:observedProperty weather:_AirTemperature ;
17                om-owl:result [ om-owl:floatValue ?temperature ] .
18 })
19 GROUP BY ?sensor
```

Listing C.8: SRBench - Q9

```
1 PREFIX om-owl: <http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#>
2 PREFIX weather: <http://knoesis.wright.edu/ssw/ont/weather.owl#>
3 PREFIX wgs84_pos: <http://www.w3.org/2003/01/geo/wgs84_pos>
4
5 CREATE STREAM sourceStream AS
6    CONSTRUCT {_:bn1 weather:_WindSpeed AVG(?windSpeed);
7              weather:_WindDirection AVG(?windDirection) . }
8    FROM STREAM <http://www.cwi.nl/SRBench/observations>
9    [ NOW - "PT1D"^^XSD:DURATION, NOW ]-> "PT1D"^^XSD:DURATION,
10   STATIC <http://www.cwi.nl/SRBench/sensors>
11   WHERE{
12       ?sensorLocation wgs84_pos:alt "%Altitude%"^^xsd:float ;
13         wgs84_pos:lat "%Latitude%"^^xsd:float ;
14         wgs84_pos:long "%Longitude%"^^xsd:float .
15   }
16   SEQUENCE BY StdSeq AS SEQ1
17   HAVING EXISTS i in SEQ1 ( GRAPH i {
18    ?sensor om-owl:processLocation ?sensorLocation ;
19           om-owl:generatedObservation ?o1 ;
20           om-owl:generatedObservation ?o2 .
21     ?o1 om-owl:observedProperty weather:_WindSpeed ;
22        om-owl:result [ om-owl:floatValue ?windSpeed ] .
23     ?o2 om-owl:observedProperty weather:_WindDirection ;
24        om-owl:result [ om-owl:floatValue ?windDirection ] .
25   })
26   GROUP BY ?sensor
27
28 CREATE STREAM if0 AS
29   CONSTRUCT {_:bn1 weather:_WindSpeed 0;
```

```
30              weather:_WindDirection ?windDirection . }
31    [ NOW - "PT1D"^^XSD:DURATION, NOW ]-> "PT1D"^^XSD:DURATION
32    SEQUENCE BY StdSeq AS SEQ1
33    HAVING EXISTS i in SEQ1 ( GRAPH i {
34       _:bn1 weather:_WindSpeed ?windspeed;
35       weather:_WindDirection ?windDirection.
36    } AND ?windSpeed < 1)
37
38 [...]
39
40 CREATE STREAM resultStream AS
41    SELECT  ?windSpeed AS ?windForce ?windDirection
42    FROM STREAM if0
43    [ NOW - "PT1D"^^XSD:DURATION, NOW ]-> "PT1D"^^XSD:DURATION,
44    STREAM if1 [...]
45    SEQUENCE BY StdSeq AS SEQ1
46    HAVING EXISTS i in SEQ1 ( GRAPH i {
47     _:bn1 weather:_WindSpeed ?windspeed;
48       weather:_WindDirection ?windDirection.
49    })
50    GROUP BY ?sensor
```

Listing C.9: SRBench - Q10

```
1 PREFIX om-owl: <http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#>
2 PREFIX weather: <http://knoesis.wright.edu/ssw/ont/weather.owl#>
3 PREFIX wgs84_pos: <http://www.w3.org/2003/01/geo/wgs84_pos>
4
5 SELECT DISTINCT ?lat ?long ?alt
6 FROM STREAM <http://www.cwi.nl/SRBench/observations>
7 [ NOW - "PT1D"^^XSD:DURATION, NOW ]-> "PT1M"^^XSD:DURATION,
8 STATIC <http://www.cwi.nl/SRBench/sensors>
9 WHERE{
10    ?sensorLocation wgs84_pos:alt ?alt ;
11                    wgs84_pos:lat ?lat ;
12                    wgs84_pos:long ?long .
13 }
14 SEQUENCE BY StdSeq AS SEQ1
15 HAVING EXISTS i in SEQ1 ( GRAPH i {
16   ?sensor om-owl:generatedObservation [a weather:SnowfallObservation] .
17   ?sensor om-owl:processLocation ?sensorLocation .
18 })
```

Listing C.10: SRBench - Q11

```
1 PREFIX om-owl: <http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#>
2 PREFIX weather: <http://knoesis.wright.edu/ssw/ont/weather.owl#>
3
4
5 SELECT DISTINCT ?sensor
6 FROM STREAM <http://www.cwi.nl/SRBench/observations>
```

```
7 [ NOW - "PT1H"^^XSD:DURATION, NOW ]-> "PT1M"^^XSD:DURATION,
8 STATIC <http://www.cwi.nl/SRBench/sensors>
9 WHERE {
10   ?sensor om-owl:hasLocatedNearRel [om-owl:hasLocation ?nearbyLocation] .
11   ?sensor2 om-owl:hasLocatedNearRel [om-owl:hasLocation ?nearbyLocation] .
12   }
13 SEQUENCE BY StdSeq AS SEQ1
14 HAVING EXISTS i in SEQ1 ( GRAPH i {
15    ?sensor om-owl:generatedObservation ?observation
16    ?observation a ?observationType ;
17                om-owl:result [ om-owl:floatValue ?value ] .
18     ?sensor2 om-owl:generatedObservation ?observation2.
19     ?observation2 a ?observationType ;
20                     om-owl:result [ om-owl:floatValue ?value2 ] .
21   })
22 HAVING AGGREGATE (?value - AVG(?value2) / ?avgValue) > 0.10
23    OR (?value - AVG(?value2) / ?avgValue) < -0.10
```

Bibliography

[1] 52North. Initiative for Geospatial Open Source Software GmbH. `http://52north.org/`. Accessed: 2016-07-15.

[2] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, et al. The design of the Borealis stream processing engine. In *CIDR*, volume 5, pages 277–289, 2005.

[3] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal. The International Journal on Very Large Data Bases*, 12(2):120–139, 2003.

[4] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[5] A. Acciarri, D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, M. Palmieri, and R. Rosati. Quonto: querying ontologies. In *AAAI*, volume 5, pages 1670–1671, 2005.

[6] D. D. Aglio, E. D. Valle, J.-P. Calbimonte, and O. Corcho. RSP-QL semantics: a unifying query model to explain heterogeneity of RDF stream processing systems. *International Journal on Semantic Web and Information Systems IJSWIS, Volume 10(4). IGI Global*, 2015.

[7] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, et al. The Stratosphere platform for big data analytics. *The VLDB Journal*, 23(6):939–964, 2014.

[8] J. F. Allen. Towards a general theory of action and time. *Artificial Intelligence*, 23(2):123–154, 1984.

[9] W. Andy Seaborne. SPARQL 1.1 property paths. `https://www.w3.org/TR/sparql11-property-paths/`. Accessed: 2016-07-15.

[10] R. Angles and C. Gutierrez. *The Semantic Web - ISWC 2008: 7th International Semantic Web Conference, ISWC 2008, Karlsruhe, Germany, October 26-30, 2008. Proceedings*, chapter The Expressive Power of SPARQL, pages 114–129. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[11] D. Anicic, P. Fodor, S. Rudolph, and N. Stojanovic. EP-SPARQL: a unified language for event processing and stream reasoning. In *WWW*, pages 635–644, 2011.

[12] D. Anicic, S. Rudolph, P. Fodor, and N. Stojanovic. Stream reasoning and complex event processing in ETALIS. *Semantic Web*, 3(4):397–407, 2012.

[13] G. Antoniou and F. Van Harmelen. Web ontology language: OWL. In *Handbook on ontologies*, pages 67–92. Springer, 2004.

[14] H. Appelrath, D. Geesen, M. Grawunder, T. Michelsen, D. Nicklas, et al. Odysseus: a highly customizable framework for creating efficient event stream management systems. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, pages 367–368. ACM, 2012.

[15] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom. Stream: The Stanford data stream management system. *Book chapter*, 2004.

[16] A. Arasu, S. Babu, and J. Widom. CQL: A language for continuous queries over streams and relations. In *Database Programming Languages*, pages 1–19. Springer, 2004.

[17] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15:121–142, 2006.

[18] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. Linear road: a stream data management benchmark. In *Proceedings of the Thirtieth international conference on Very large data bases*, pages 480–491. VLDB Endowment, 2004.

[19] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. Spark sql: Relational data processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1383–1394. ACM, 2015.

[20] A. Artale, D. Calvanese, R. Kontchakov, and M. Zakharyaschev. The DL-Lite family and relations. *J. Artif. Intell. Res. (JAIR)*, 36:1–69, 2009.

[21] A. Artale and E. Franconi. A survey of temporal extensions of description logics. *Annals of Mathematics and Artificial Intelligence*, 30(1-4):171–210, Mar. 2001.

[22] A. Artale, R. Kontchakov, A. Kovtunova, V. Ryzhikov, F. Wolter, and M. Zakharyaschev. Temporal OBDA with LTL and DL-Lite. In M. Bienvenu, M. Ortiz, R. Rosati, and M. Simkus, editors, *Proc. of the 27th Int. Workshop on Description Logics (DL14)*, volume 1193, pages 21–32. CEUR Workshop Proceedings, 2014.

[23] A. Artale, R. Kontchakov, F. Wolter, and M. Zakharyaschev. Temporal description logic for ontology-based data access. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, IJCAI'13, pages 711–717, 2013.

[24] A. Artikis, A. Skarlatidis, F. Portet, and G. Paliouras. Logic-based event recognition. *Knowledge Eng. Review*, 27(4):469–506, 2012.

[25] F. Baader, S. Borgwardt, and M. Lippmann. Temporal conjunctive queries in expressive description logics with transitive roles. In B. Pfahringer and J. Renz, editors, *Proceedings of the 28th Australasian Joint Conference on Artificial Intelligence (AI'15)*, volume 9457 of *Lecture Notes in Artificial Intelligence*, pages 21–33, Canberra, Australia, 2015. Springer-Verlag.

[26] F. Baader and W. Nutt. Basic description logics. In F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider, editors, *The Description Logic Handbook*, pages 43–95. Cambridge University Press, 2003.

[27] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–16. ACM, 2002.

[28] S. Babu and J. Widom. Continuous queries over data streams. *ACM Sigmod Record*, 30(3):109–120, 2001.

[29] D. Barbieri, D. Braga, S. Ceri, E. Della Valle, and M. Grossniklaus. Stream reasoning: Where we got so far. In *Proceedings of the 4th workshop on new forms of reasoning for the Semantic Web: Scalable & dynamic*, pages 1–7, 2010.

[30] D. F. Barbieri, D. Braga, S. Ceri, E. Della Valle, and M. Grossniklaus. C-SPARQL: SPARQL for continuous querying. In *Proceedings of the 18th international conference on World wide web*, pages 1061–1062. ACM, 2009.

[31] D. F. Barbieri, D. Braga, S. Ceri, E. Della Valle, and M. Grossniklaus. Incremental reasoning on streams and rich background knowledge. In *Extended Semantic Web Conference*, pages 1–15. Springer, 2010.

[32] D. F. Barbieri, D. Braga, S. Ceri, and M. Grossniklaus. An execution environment for C-SPARQL queries. In *Proceedings of the 13th International Conference on Extending Database Technology*, pages 441–452. ACM, 2010.

[33] O. Bartlett. Linked data: Connecting together the BBC's online content. `http://www.bbc.co.uk/blogs/internet/entries/af6b613e-6935-3165-93ca-9319e1887858`. Accessed: 2016-07-15.

[34] J. Barwise. *Handbook of mathematical logic*. North-Holland Pub. Co, Amsterdam New York, 1977.

[35] H. Beck, M. Dao-Tran, T. Eiter, and M. Fink. Towards a logic-based framework for analyzing stream reasoning. In *Proceedings of the 3rd International Conference on Ordering and Reasoning-Volume 1303*, pages 11–22. CEUR-WS. org, 2014.

[36] H. Beck, M. Dao-Tran, T. Eiter, and M. Fink. Towards ideal semantics for analyzing stream reasoning. In *International Workshop on Reactive Concepts*, volume 2014, 2014.

[37] H. Beck, M. Dao-Tran, T. Eiter, and M. Fink. LARS: A logic-based framework for analyzing reasoning over streams. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[38] D. Beckett and A. Barstow. N-triples. *W3C RDF Core WG Internal Working Draft*, 2001. `http://www.w3.org/TR/n-triples/` Accessed: 2016-07-15.

[39] D. Beckett, T. Berners-Lee, and E. Prudhommeaux. Turtle-terse RDF triple language. *W3C Team Submission*, 14:7, 2008. `http://www.w3.org/TR/turtle/` Accessed: 2016-07-15.

[40] D. Beckett and B. McBride. RDF/XML syntax specification (revised). *W3C recommendation*, 10, 2004. `http://www.w3.org/TR/rdf-syntax-grammar/` Accessed: 2015-07-15.

[41] K. Bereta, D. Bilidas, Y. Chronis, C. Mallios, V. Nikolopoulos, A. Papadopoulos, C. Svingos, D. Theodosakis, T. Mailis, Y. Kotidis, M. Koubarakis, and Y. Ioannidis. Deliverable D7.3 – optimization techniques for distributed query planning and execution. Technical report, National and Kapodistrian University of Athens, 2015.

[42] K. Bereta, D. Bilidas, H. Kllapi, C. Mallios, A. Papadopoulos, C. Svingos, D. Theodosakis, M. Koubarakis, and Y. Ioannidis. Deliverable D7.2 – techniques for distributed query planning and execution: Continuous/streaming and temporal queries. Deliverable FP7-318338, EU, Oct. 2014.

[43] K. Bereta, P. Smeros, and M. Koubarakis. Representing and querying the valid time of triples for linked geospatial data. In *In the 10th Extended Semantic Web Conference (ESWC 2013). Montpellier, France. May 26–30*, 2013.

[44] T. Berners-Lee and D. Connolly. Notation3 (n3): A readable RDF syntax. *W3C Submission, Jan*, 2008. `https://www.w3.org/TeamSubmission/n3/` Accessed: 2016-07-15.

[45] C. Bizer and A. Seaborne. D2RQ-treating non-RDF databases as virtual RDF graphs. In *Proceedings of the 3rd international semantic web conference (ISWC2004)*, volume 2004, 2004.

[46] W. Bohlken, B. Neumann, L. Hotz, and P. Koopmann. Ontology-based real-time activity monitoring using beam search. In *ICVS*, pages 112–121, 2011.

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

[48] A. Bolles, M. Grawunder, and J. Jacobi. Streaming SPARQL extending SPARQL to process data streams. In *Proceedings of the 5th European semantic web conference on The Semantic Web: Research and Applications*, pages 448–462. Springer-Verlag, 2008.

[49] S. Borgwardt, M. Lippmann, and V. Thost. Temporal query answering in the description logic DL-Lite. In P. Fontaine, C. Ringeissen, and R. A. Schmidt, editors, *Frontiers of Combining Systems*, volume 8152 of *LNCS*, pages 165–180. Springer, 2013.

[50] S. Borgwardt, M. Lippmann, and V. Thost. Temporal query answering w.r.t. *DL-Lite*-ontologies. LTCS-Report 13-05, Chair of Automata Theory, TU Dresden, Dresden, Germany, 2013. See http://lat.inf.tu-dresden.de/research/reports.html.

[51] I. Botan, R. Derakhshan, N. Dindar, L. Haas, R. J. Miller, and N. Tatbul. Secret: a model for analysis of the execution semantics of stream processing systems. *Proceedings of the VLDB Endowment*, 3(1-2):232–243, 2010.

[52] D. Botstein, J. Cherry, M. Ashburner, C. Ball, J. Blake, H. Butler, A. Davis, K. Dolinski, S. Dwight, J. Eppig, et al. Gene ontology: tool for the unification of biology. *Nat Genet*, 25(1):25–29, 2000.

[53] M. Botts, G. Percivall, C. Reed, and J. Davidson. OGC® sensor web enablement: Overview and high level architecture. In *GeoSensor networks*, pages 175–190. Springer, 2008.

[54] M. Botts and A. Robin. Opengis sensor model language (SensorML) implementation specification. *OpenGIS Implementation Specification OGC*, 7(000), 2007.

[55] C. Burleson. Users of json ld. `https://github.com/json-ld/json-ld.org/wiki/Users-of-JSON-LD`. Accessed: 2016-07-15.

[56] J.-P. Calbimonte. *Ontology-based Access to Sensor Data Streams*. dissertation, Universidad Politecnica de Madrid, 2013.

[57] J.-P. Calbimonte, O. Corcho, and A. J. G. Gray. Enabling ontology-based access to streaming data sources. In *Proceedings of the 9th international semantic web conference on The semantic web - Volume Part I*, ISWC'10, pages 96–111, Berlin, Heidelberg, 2010. Springer-Verlag.

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

[59] J.-P. Calbimonte, J. Mora, and O. Corcho. Query rewriting in RDF stream processing. In *International Semantic Web Conference*, pages 486–502. Springer, 2016.

[60] D. Calvanese, B. Cogrel, S. Komla-Ebri, R. Kontchakov, D. Lanti, M. Rezk, M. Rodriguez-Muro, and G. Xiao. Ontop: Answering SPARQL queries over relational databases. *Submitted to the Semantic Web Journal*, 2015. http://ontop.inf.unibz.it/ Accessed: 2016-07-15.

[61] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, A. Poggi, M. Rodríguez-Muro, and R. Rosati. Ontologies and databases: The DL-Lite approach. In S. Tessaris and E. Franconi, editors, *Semantic Technologies for Informations Systems (RW 2009)*, volume 5689 of *LNCS*, pages 255–356. Springer, 2009.

[62] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, and R. Rosati. DL-Lite: Tractable description logics for ontologies. In *AAAI*, volume 5, pages 602–607, 2005.

[63] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, and R. Rosati. Tractable reasoning and efficient query answering in description logics: The DL-Lite family. *Journal of Automated reasoning*, 39(3):385–429, 2007.

[64] D. Calvanese, M. Giese, P. Haase, I. Horrocks, T. Hubauer, Y. E. Ioannidis, E. Jiménez-Ruiz, E. Kharlamov, H. Kllapi, J. W. Klüwer, M. Koubarakis, S. Lamparter, R. Möller, C. Neuenstadt, T. Nordtveit, Ö. L. Özçep, M. Rodriguez-Muro, M. Roshchin, D. F. Savo, M. Schmidt, A. Soylu, A. Waaler, and D. Zheleznyakov. Optique: OBDA solution for big data. In *Proc. ESWC (Satellite Events)*, pages 293–295, 2013.

[65] D. Calvanese, E. Kharlamov, W. Nutt, and C. Thorne. Aggregate queries over ontologies. In *Proceedings of the 2nd international workshop on Ontologies and information systems for the semantic web*, pages 97–104. ACM, 2008.

[66] D. Calvanese and D. Lembo. Ontology-based data access. In *Tutorial. 6th Int. Semantic Web Conf.(ISWC 2007)*, 2007.

[67] D. Calvanese, P. Liuzzo, A. Mosca, J. Remesal, M. Rezk, and G. Rull. Ontology-based data integration in EPNet: Production and distribution of food during the roman empire. *Eng. Appl. of AI*, 51:212–229, 2016.

[68] U. Cetintemel, D. Abadi, Y. Ahmad, H. Balakrishnan, M. Balazinska, M. Cherniack, J. Hwang, W. Lindner, S. Madden, A. Maskey, et al. The aurora and Borealis stream processing engines. *Data Stream Management: Processing High-Speed Data Streams,, Springer-Verlag*, pages 1–23, 2006.

[69] W. Ceusters, B. Smith, A. Kumar, and C. Dhaen. Ontology-based error detection in SNOMED-CT. *Proceedings of MEDINFO*, 2004:482–6, 2004.

[70] S. Chandrasekaran. *Query Processing over Live and Archived Data Streams*. PhD thesis, University of California at Berkeley, Berkeley, CA, USA, 2005. AAI3210530.

[71] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[72] H. Chen, R. H. Chiang, and V. C. Storey. Business intelligence and analytics: From big data to big impact. *MIS quarterly*, 36(4):1165–1188, 2012.

[73] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: a scalable continuous query system for internet databases. *SIGMOD Rec. 29, 2, 379–390.*, 2000.

[74] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng, et al. Benchmarking streaming computation engines: Storm, Flink and Spark Streaming. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1789–1792. IEEE, 2016.

[75] C. Civili, M. Console, G. De Giacomo, D. Lembo, M. Lenzerini, L. Lepore, R. Mancini, A. Poggi, R. Rosati, M. Ruzzi, V. Santarelli, and D. F. Savo. MASTRO STUDIO: managing ontology-based data access applications. *PVLDB*, 6(12):1314–1317, 2013.

[76] M. Compton, P. Barnaghi, L. Bermudez, R. GarcÃa-Castro, O. Corcho, S. Cox, J. Graybeal, M. Hauswirth, C. Henson, A. Herzog, V. Huang, K. Janowicz, W. D. Kelsey, D. L. Phuoc, L. Lefort, M. Leggieri, H. Neuhaus, A. Nikolov, K. Page, A. Passant, A. Sheth, and K. Taylor. The SSN ontology of the W3C semantic sensor network incubator group. *Web Semantics: Science, Services and Agents on the World Wide Web*, 17(0):25–32, 2012.

[77] M. Compton, C. A. Henson, L. Lefort, H. Neuhaus, and A. P. Sheth. A survey of the semantic specification of sensors. *2nd International Workshop on Semantic Sensor Networks, at 8th International Semantic Web Conference*, 2009.

[78] M. Compton, H. Neuhaus, K. Taylor, and K.-N. Tran. Reasoning about sensors and compositions. In *SSN*, pages 33–48. Citeseer, 2009.

[79] C. Cranor, Y. Gao, T. Johnson, V. Shkapenyuk, and O. Spatscheck. Gigascope: High performance network monitoring with an SQL interface. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 623–623. ACM, 2002.

[80] CSRBench. CSRBench-oracle. `https://github.com/dellaglio/csrbench-oracle/`. Accessed: 2016-07-15.

[81] G. Cugola and A. Margara. Processing flows of information: From data stream to complex event processing.sing (IFP) Sys. *ACM Comput. SuACM, 44(3):15:1-15:62*, 2011.

[82] G. Cugola and A. Margara. Complex event processing with T-REX. *Journal of Systems and Software*, 85(8):1709–1728, 2012.

[83] G. Cugola and A. Margara. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3):15, 2012.

[84] D. Culler, D. Estrin, and M. Srivastava. Guest editors' introduction: Overview of sensor networks. *Computer*, 37(8):41–49, 2004.

[85] R. Cyganiak. The linking open data cloud diagram. `http://lod-cloud.net/`. Accessed: 2015.

[86] R. Cyganiak, A. Harth, and A. Hogan. N-quads: Extending n-triples with context. `http://www.w3.org/TR/n-quads/`, 2008.

[87] M. Dao-Tran, H. Beck, and T. Eiter. Contrasting RDF stream processing semantics. Technical report, Technical report, Institut für Informationssysteme, TU Wien, 2015.

[88] D. Dell Aglio, J.-P. Calbimonte, M. Balduini, O. Corcho, and E. Della Valle. On correctness in RDF stream processor benchmarking. In *The Semantic Web–ISWC 2013*, pages 326–342. Springer, 2013.

[89] D. DellAglio, J.-P. Calbimonte, E. Della Valle, and O. Corcho. Towards a unified language for RDF stream query processing. In *European Semantic Web Conference*, pages 353–363. Springer, 2015.

[90] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for XML. *Computer networks*, 31(11):1155–1169, 1999.

[91] O. Erling and I. Mikhailov. RDF support in the virtuoso DBMS. In *Networked Knowledge-Networked Media*, pages 7–24. Springer, 2009.

[92] Espertech. Esper. `http://www.espertech.com`, 2010.

[93] A. S. Foundation. Apache Flink. `flink.apache.org`. Accessed: 2016-07-15.

[94] A. S. Foundation. Spark Streaming. `http://spark.apache.org/streaming/`, 2015. last checked: 13.10.2015.

[95] I. Galpin, C. Y. Brenninkmeijer, F. Jabeen, A. A. Fernandes, and N. W. Paton. Comprehensive optimization of declarative sensor network queries. In *International Conference on Scientific and Statistical Database Management*, pages 339–360. Springer, 2009.

[96] A. Galton. Reified temporal theories and how to unreify them. In *Proceedings of the 12th international joint conference on Artificial intelligence - Volume 2*, IJCAI'91, pages 1177–1182, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.

[97] S. Gao, T. Scharrenbach, J. Kietz, and A. Bernstein. Running out of bindings? integrating facts and events in linked data stream processing. In *Joint Proceedings of the 1st Joint International Workshop on Semantic Sensor Networks and Terra Cognita (SSN-TC 2015) and the 4th International Workshop*

on Ordering and Reasoning (OrdRing 2015) co-located with the 14th International Semantic Web Conference (ISWC 2015), Bethlehem, Pennsylvania, United States, October 11th - and - 12th, 2015., pages 63–74, 2015.

[98] M. Garofalakis, J. Gehrke, and R. Rastogi. *Data stream management: processing high-speed data streams (data-centric systems and applications).* Springer-Verlag New York, Inc., 2007.

[99] GeoNames. The GeoNames geographical database. `http://http://www.geonames.org/`. Accessed: 2016-07-15.

[100] L. George. *HBase: the definitive guide.* " O'Reilly Media, Inc.", 2011.

[101] G. Gidofalvi, T. B. Pedersen, T. Risch, and E. Zeitler. Highly scalable trip grouping for large-scale collective transportation systems. In *Proceedings of the 11th international conference on Extending database technology: Advances in database technology*, pages 678–689. ACM, 2008.

[102] M. Giese, D. Calvanese, P. Haase, I. Horrocks, Y. Ioannidis, H. Kllapi, M. Koubarakis, M. Lenzerini, R. Möller, M. Rodriguez-Muro, et al. Scalable end-user access to big data. *Big Data Computing*, pages 205–245, 2013.

[103] M. Giese, O. Ozcep, R. Rosati, A. Soylu, G. Vega-Gorgojo, A. Waaler, P. Haase, E. Jimenez-Ruiz, D. Lanti, M. Rezk, et al. Optique: Zooming in on big data. *IEEE Computer*, pages 60–67, 2015.

[104] GitHub. YABench source code. `https://github.com/YABench/`. Accessed: 2016-07-15.

[105] L. Golab and M. T. Özsu. Issues in data stream management. *ACM Sigmod Record*, 32(2):5–14, 2003.

[106] J. Golbeck and M. Rothstein. Linking social networks on the web with FOAF. In *Proc. 17th Int. World Wide Web Conf.(April 2008)*, 2008.

[107] S. González-Valenzuela, M. Chen, and V. C. Leung. Mobility support for health monitoring at home using wearable sensors. *IEEE Transactions on Information Technology in Biomedicine*, 15(4):539–549, 2011.

[108] G. Gottlob, G. Orsi, and A. Pieris. Ontological query answering via rewriting. In *Advances in Databases and Information Systems*, pages 1–18. Springer, 2011.

[109] O. E. Group. MorphRDB, Oct. 2015.

[110] T. P. G. D. Group. PostreSQL wikipage. `https://wiki.postgresql.org/wiki/FAQ`. Accessed: 2016-07-15.

[111] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, and P. Valduriez. Streamcloud: A large scale data streaming system. In *Distributed Computing Systems (ICDCS), 2010 IEEE 30th International Conference on*, pages 126–137. IEEE, 2010.

[112] C. Gutierrez, C. Hurtado, and R. Vaisman. Temporal RDF. In *In European Conference on the Semantic Web (ECSW' 05)*, pages 93–107, 2005.

[113] C. Gutierrez, C. A. Hurtado, and A. Vaisman. Introducing time into RDF. *Knowledge and Data Engineering, IEEE Transactions on*, 19(2):207–218, 2007.

[114] S. Hallé and S. Varvaressos. A formalization of complex event stream processing. In *Enterprise Distributed Object Computing Conference (EDOC), 2014 IEEE 18th International*, pages 2–11. IEEE, 2014.

[115] S. Harris, A. Seaborne, and E. Prud'hommeaux. SPARQL 1.1 query language. *W3C Recommendation*, 21, 2013.

[116] P. K. Harshal. Linked sensor data. `https://datahub.io/dataset/knoesis-linked-sensor-data`. Accessed: 2016-07-15.

[117] F. Heintz, J. Kvarnström, and P. Doherty. Stream-based reasoning support for autonomous systems. In *ECAI*, pages 183–188, 2010.

[118] F. Heintz, P. Rudol, and P. Doherty. From images to traffic behavior - a UAV tracking and monitoring application. In *FUSION*, pages 1–8, 2007.

[119] M. Hert, G. Ghezzi, M. Würsch, and H. C. Gall. *The Semantic Web – ISWC 2011: 10th International Semantic Web Conference, Bonn, Germany, October 23-27, 2011, Proceedings, Part II*, chapter How to "Make a Bridge to the New Town" Using OntoAccess, pages 112–127. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[120] M. Hert, G. Reif, and H. Gall. A comparison of RDB-to-RDF mapping languages. In *Proceedings of the 7th International Conference on Semantic Systems (I-Semantics)*, Graz, Austria, SEP 2011.

[121] I. Hodkinson and M. Reynolds. Temporal logic. In P. Blackburn, J. van Benthem, and F. Wolter, editors, *Handbook of Modal Logic*, volume 6, chapter 11, pages 655–720. Elsevier Science, 2006.

[122] I. Horrocks and U. Sattler. A description logic with transitive and inverse roles and role hierarchies. *Journal of logic and computation*, 9(3):385–410, 1999.

[123] I. Horrocks and U. Sattler. A description logic with transitive and inverse roles and role hierarchies. *Journal of Logic and Computation*, 9(3):385–410, 1999.

[124] I. Horrocks and U. Sattler. A tableau decision procedure for $\mathcal{SHOIQ}$. *Journal of Automated Reasoning*, 39(3):249–276, 2007.

[125] N. Jain, L. Amini, H. Andrade, R. King, Y. Park, P. Selo, and C. Venkatramani. Design, implementation, and evaluation of the linear road bnchmark on the stream processing core. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 431–442, New York, NY, USA, 2006. ACM.

[126] K. Janowicz and M. Compton. The stimulus-sensor-observation ontology design pattern and its integration into the semantic sensor network ontology. In *SSN*, 2010.

[127] E. Jiménez-Ruiz, E. Kharlamov, D. Zheleznyakov, I. Horrocks, C. Pinkel, M. G. Skjæveland, E. Thorstensen, and J. Mora. Bootox: practical mapping of RDBs to OWL 2. In *The Semantic Web-ISWC 2015*, pages 113–132. Springer, 2015.

[128] R. Kajic. Evaluation of the stream query language CQL. Master's thesis, Uppsala Universitet, Institutionen for informationsteknologi, 2010.

[129] Y. Kazakov. $\mathcal{RIQ}$ and $\mathcal{SROIQ}$ are harder than $\mathcal{SHOIQ}$. In *In Proc. 11th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR 08)*, pages 274–284, 2008.

[130] E. Kharlamov, S. Brandt, M. Giese, E. Jiménez-Ruiz, Y. Kotidis, S. Lamparter, T. Mailis, C. Neuenstadt, Ö. Özçep, C. Pinkel, et al. Enabling semantic access to static and streaming distributed data with Optique: demo. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, pages 350–353. ACM, 2016.

[131] E. Kharlamov, S. Brandt, E. Jimenez-Ruiz, Y. Kotidis, S. Lamparter, T. Mailis, C. Neuenstadt, Ö. Öezçep, C. Pinkel, C. Svingos, et al. Ontology-based integration of streaming and static relational data with Optique. In *Proceedings of the 2016 International Conference on Management of Data*, pages 2109–2112. ACM, 2016.

[132] E. Kharlamov, D. Hovland, E. Jiménez-Ruiz, D. L. C. Pinkel, M. Rezk, M. G. Skjæveland, E. Thorstensen, G. Xiao, D. Zheleznyakov, E. Bjørge, and I. Horrocks. Enabling Ontology Based Access at an Oil and Gas Company Statoil. In *ISWC*, 2015.

[133] E. Kharlamov, S. B. M. G. E. Jiménez, R. S. Lamparter, C. Neuenstadt, O. O. C. P. A. Soylu, D. Zheleznyakov, and I. Horrocks. Semantic access to Siemens streaming data: the Optique way. *ISWC (Posters and Demos)*, 2015.

[134] E. Kharlamov, E. Jiménez-Ruiz, D. Zheleznyakov, D. Bilidas, M. Giese, P. Haase, I. Horrocks, H. Kllapi, M. Koubarakis, Ö. Özçep, et al. Optique: Towards OBDA systems for industry. In *The Semantic Web: ESWC 2013 Satellite Events*, pages 125–140. Springer, 2013.

[135] E. Kharlamov, Y. Kotidis, T. Mailis, C. Neuenstadt, C. Nikolaou, Ö. Özcep, C. Svingos, D. Zheleznyakov, S. Lamparter, I. Horrocks, et al. Towards analytics aware ontology based access to static and streaming data (extended version). In *Proceedings of the 15th International Semantic Web Conference (ISWC 2016)*, 2016.

[136] E. Kharlamov, N. Solomakhina, Ö. L. Özçep, D. Zheleznyakov, T. Hubauer, S. Lamparter, M. Roshchin, A. Soylu, and S. Watson. How Semantic Technologies Can Enhance Data Access at Siemens Energy. In *ISWC*, 2014.

[137] E. Kharlamov, N. Solomakhina, Ö. L. Özçep, D. Zheleznyakov, T. Hubauer, S. Lamparter, M. Roshchin, A. Soylu, and S. Watson. How semantic technologies can enhance data access at siemens energy. In *The Semantic Web–ISWC 2014*, pages 601–619. Springer, 2014.

[138] J. Kietz, T. Scharrenbach, L. Fischer, A. Bernstein, and K. Nguyen. TEF-SPARQL: The DDIS query-language for time annotated event and fact triple-streams. Technical report, Technical report, University of Zurich, Department of Informatics, 2013.

[139] J.-H. Kim, H. Kwon, D.-H. Kim, H.-Y. Kwak, and S.-J. Lee. Building a service-oriented ontology for wireless sensor networks. In *Computer and Information Science, 2008. ICIS 08. Seventh IEEE/ACIS International Conference on*, pages 649–654. IEEE, 2008.

[140] H. Kllapi, P. Sakkos, A. Delis, D. Gunopulos, and Y. Ioannidis. Elastic processing of analytical query workloads on IaaS Clouds. *arXiv preprint arXiv:1501.01070*, 2015.

[141] G. Klyne and J. J. Carroll. Resource description framework (RDF): Concepts and abstract syntax. *http://www.w3.org/TR/rdf-concepts/*, 2006.

[142] M. Kolchin and P. Wetz. Demo: YABench-yet another RDF stream processing benchmark. In *RSP Workshop*, 2015.

[143] S. Komazec, D. Cerri, and D. Fensel. Sparkwave: continuous schema-enhanced pattern matching over RDF data streams. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, pages 58–68. ACM, 2012.

[144] R. Kontchakov, C. Lutz, D. Toman, F. Wolter, and M. Zakharyaschev. The combined approach to query answering in DL-Lite. In F. Lin and U. Sattler, editors, *Proceedings of the 12th International Conference on Principles of Knowledge Representation and Reasoning (KR2010)*. AAAI Press, 2010.

[145] E. V. Kostylev and J. Reutter. Answering counting aggregate queries over ontologies of the DL-Lite family. In *Proceedings of the 27th AAAI Conference on Artificial Intelligence (AAAI–13), Bellevue, Washington,*, 2013.

[146] R. A. Kowalski and F. Sadri. Towards a logic-based unifying framework for computing. *CoRR*, abs/1301.6905, 2013.

[147] R. A. Kowalski, F. Toni, and G. Wetzel. Towards a declarative and efficient glass-box CLP language. In *WLP*, pages 138–141, 1994.

[148] J. Krämer and B. Seeger. Semantics and implementation of continuous sliding window queries over data streams. *ACM Transactions on Database Systems (TODS)*, 34(1):4, 2009.

[149] M. Krötzsch. *OWL 2 Profiles: An introduction to lightweight ontology languages.* Springer, 2012.

[150] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[151] N. D. Lane, E. Miluzzo, H. Lu, D. Peebles, T. Choudhury, and A. T. Campbell. A survey of mobile phone sensing. *IEEE Communications magazine*, 48(9):140–150, 2010.

[152] D. Laney. 3d data management: Controlling data volume, velocity and variety. *META Group Research Note*, 6:70, 2001.

[153] M. Lanthaler and C. Gütl. On using json-ld to create evolvable restful services. In *Proceedings of the Third International Workshop on RESTful Design*, pages 25–32. ACM, 2012.

[154] D. Le-Phuoc, M. Dao-Tran, M.-D. Pham, P. Boncz, T. Eiter, and M. Fink. Linked stream data processing engines: Facts and figures. In P. Cudre-Mauroux, J. Heflin, E. Sirin, T. Tudorache, J. Euzenat, M. Hauswirth, J. Parreira, J. Hendler, G. Schreiber, A. Bernstein, and E. Blomqvist, editors, *The*

Semantic Web–ISWC 2012, Lecture Notes in Computer Science, pages 300–312. Springer Berlin Heidelberg, 2012.

[155] F. Lécué. Diagnosing changes in an ontology stream: A DL reasoning approach. In *AAAI*. Citeseer, 2012.

[156] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. Semantics and evaluation techniques for window aggregates in data streams. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 311–322. ACM, 2005.

[157] L. Liu and C. Pu. A dynamic query scheduling framework for distributed and evolving information systems. In *Distributed Computing Systems, 1997., Proceedings of the 17th International Conference on*, pages 474–481. IEEE, 1997.

[158] L. Liu, C. Pu, and W. Tang. Continual queries for internet scale event-driven information delivery. *IEEE Trans. on Knowl. and Data Eng. 11, 4, 610–628.*, 1999.

[159] C. Lutz, D. Toman, and F. Wolter. Conjunctive query answering in $\mathcal{EL}$ using a database system. In *In Proceeding of the 5th International Workshop on OWL: Experiences and Directions (OWLED 2008)*, 2008.

[160] C. Lutz, D. Toman, and F. Wolter. Conjunctive query answering in the description logic $\mathcal{EL}$ using a relational database system. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI-09).* AAAI Press, 2009.

[161] N. Marz and J. Warren. *Big Data: Principles and best practices of scalable realtime data systems.* Manning Publications Co., 2015.

[162] V. Mayer-Schönberger and K. Cukier. *Big data: A revolution that will transform how we live, work, and think.* Houghton Mifflin Harcourt, 2013.

[163] X. Meng, J. Bradley, B. Yuvaz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, et al. MLlib: Machine learning in Apache Spark. *JMLR*, 17(34):1–7, 2016.

[164] Microsoft. Sending flight information to Microsoft Cortana with contextual awareness. `https://msdn.microsoft.com/en-us/library/dn632191.aspx`, 2014. Accessed: 2016-07-15.

[165] V. Milea, F. Frasincar, and U. Kaymak. tOWL: A Temporal Web Ontology Language. *IEEE Transactions on Systems, Man and Cybernetics*, 42(1):268–281, 2012.

[166] E. Miller. An introduction to the resource description framework. *Bulletin of the American Society for Information Science and Technology*, 25(1):15–19, 1998.

[167] R. Möller, C. Neuenstadt, O. Ozçep, and S. Wandelt. Advances in accessing big data with expressive ontologies. In I. J. Timm and M. Thimm, editors, *KI 2013: Advances in Artificial Intelligence*, volume 8077 of *Lecture Notes in Computer Science*, pages 118–129. Springer Berlin Heidelberg, 2013.

[168] R. Möller, C. Neuenstadt, and Özgür. L. Özçep. Stream-temporal querying with ontologies. In D. Nicklas and Özgür. L. Özçep, editors, *HiDeSt '15—Proceedings of the First Workshop on High-Level Declarative Stream Processing (co-located with KI 2015)*, volume 1447 of *CEUR Workshop Proceedings*, pages 42–55. CEUR-WS.org, 2015.

[169] J. Mora and Ó. Corcho. Engineering optimisations in query rewriting for OBDA. In *Proceedings of the 9th International Conference on Semantic Systems*, pages 41–48. ACM, 2013.

[170] J. Mora and O. Corcho. Towards a systematic benchmarking of ontology-based query rewriting systems. In *International Semantic Web Conference*, pages 376–391. Springer, 2013.

[171] J. Mora, R. Rosati, and O. Corcho. Kyrie2: Query rewriting under extensional constraints in\mathcal{ELHIO}. In *International Semantic Web Conference*, pages 568–583. Springer, 2014.

[172] B. Motik. Representing and querying validity time in RDF and OWL: a logic-based approach. In *Proceedings of the 9th international semantic web conference on The semantic web - Volume Part I*, ISWC'10, pages 550–565, Berlin, Heidelberg, 2010. Springer-Verlag.

[173] K. Munir, M. Odeh, and R. McClatchey. Ontology-driven relational query formulation using the semantic and assertional capabilities of owl-dl. *Knowledge-Based Systems*, 35:144 – 159, 2012.

[174] D. Namiot. On big data stream processing. *International Journal of Open Information Technologies*, 3(8):48–51, 2015.

[175] C. Neuenstadt, R. Möller, and Özgür. L. Özçep. OBDA for temporal querying and streams with STARQL. In D. Nicklas and Özgür. L. Özçep, editors, *HiDeSt '15—Proceedings of the First Workshop on High-Level Declarative Stream Processing (co-located with KI 2015)*, volume 1447 of *CEUR Workshop Proceedings*, pages 70–75. CEUR-WS.org, 2015.

[176] H. Neuhaus and M. Compton. The semantic sensor network ontology. In *AGILE workshop on challenges in geospatial data harmonisation, Hannover, Germany*, pages 1–33, 2009.

[177] B. Neumann and H.-J. Novak. Event models for recognition and natural language description of events in real-world image sequences. In *IJCAI*, pages 724–726, 1983.

[178] B. Neumann and H.-J. Novak. NOAS: Ein System zur natürlichsprachlichen Beschreibung zeitveränderlicher Szenen. *Inform., Forsch. Entwickl.*, 1(2):83–92, 1986.

[179] Ö. L. Özçep and R. Möller. Ontology based data access on temporal and streaming data. In *Reasoning Web. Reasoning and the Web in the Big Data Era*, volume 8714 of *LNCS*, 2014.

[180] Ö. L. Özçep and R. Möller. Ontology based data access on temporal and streaming data. In M. Koubarakis, G. Stamou, G. Stoilos, I. Horrocks, P. Kolaitis, G. Lausen, and G. Weikum, editors, *Reasoning Web. Reasoning and the Web in the Big Data Era*, volume 8714. of *Lecture Notes in Computer Science*, 2014.

[181] Özgür. L. Özçep, R. Möller, and C. Neuenstadt. A stream-temporal query language for ontology based data access. In *KI 2014*, volume 8736 of *LNCS*, pages 183–194. Springer International Publishing Switzerland, 2014.

[182] Özgür. L. Özçep, R. Möller, and C. Neuenstadt. Stream-query compilation with ontologies. In B. Pfahringer and J. Renz, editors, *Poceedings of the 28th Australasian Joint Conference on Artificial Intelligence 2015 (AI 2015)*, volume 9457 of *LNAI*. Springer International Publishing, 2015.

[183] Özgür L. Özçep, R. Möller, and C. Neuenstadt. A stream-temporal query language for ontology based data access. In M. Bienvenu, M. Ortiz, R. Rosati, and M. Simkus, editors, *Proceedings of the 7th International Workshop on Description Logics (DL-2014)*, 2014.

[184] K. Patroumpas and T. Sellis. Window specification over data streams. In *Current Trends in Database Technology–EDBT 2006*, pages 445–464. Springer, 2006.

[185] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. In *International semantic web conference*, volume 4273, pages 30–43. Springer, 2006.

[186] M. Perry. *A Framework to Support Spatial, Temporal and Thematic Analytics over Semantic Web Data.* PhD thesis, Wright State UNiversity, 2008.

[187] D. Pfisterer, K. Römer, D. Bimschas, O. Kleine, R. Mietz, C. Truong, H. Hasemann, A. Kröller, M. Pagel, M. Hauswirth, et al. Spitfire: toward a semantic web of things. *IEEE Communications Magazine*, 49(11):40–48, 2011.

[188] M.-D. Pham, P. Boncz, and O. Erling. S3g2: A scalable structure-correlated social graph generator. In *Selected Topics in Performance Evaluation and Benchmarking*, pages 156–172. Springer, 2012.

[189] D. L. Phuoc, M. Dao-Tran, J. X. Parreira, and M. Hauswirth. A native and adaptive approach for unified processing of linked streams and linked data. In *International Semantic Web Conference (1)*, pages 370–388, 2011.

[190] pipelinedb.com. PipelineDB–the streaming SQL database, 2015.

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

[192] A. Poggi, M. Rodriguez, and M. Ruzzi. Ontology-based database access with dig-mastro and the obda plugin for protégé. In *Proc. of OWLED*, volume 8, 2008.

[193] T. PostgreSQL Global Development Group. About PostgreSQL. https://www.postgresql.org/about/. Accessed: 2016-07-15.

[194] F. Priyatna, O. Corcho, and J. Sequeda. Formalisation and experiences of R2RML-based SPARQL to SQL query translation using morph. In *World Wide Web Conference*, 2014.

[195] F. Probst. Ontological analysis of observations and measurements. In *Geographic information science*, pages 304–320. Springer, 2006.

[196] F. Reiss and J. M. Hellerstein. Data triage: an adaptive architecture for load shedding in TelegraphCQ. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pages 155–156, April 2005.

[197] Y. Ren and J. Z. Pan. Optimising ontology stream reasoning with truth maintenance system. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 831–836. ACM, 2011.

[198] T. Rist, G. Herzog, and E. André. Ereignismodellierung zur inkrementellen High-Level Bildfolgenanalyse. In *ÖGAI*, pages 1–11, 1987.

[199] A. Rodrıguez, R. McGrath, Y. Liu, J. Myers, and I. Urbana-Champaign. Semantic management of streaming data. *Proc. Semantic Sensor Networks*, 80:80–95, 2009.

[200] J. B. Rodríguez, O. Corcho, and A. Gómez-Pérez. R2O, an extensible and semantically based database-to-ontology mapping language. 3372:1069–1070, August 2004. Ontology Engineering Group OEG.

[201] J. B. Rodriguez and A. Gómez-Pérez. Upgrading relational legacy data to the semantic web. In *Proceedings of the 15th international conference on World Wide Web*, pages 1069–1070. ACM, 2006.

[202] M. Rodriguez-Muro and D. Calvanese. High performance query answering over DL-Lite ontologies. In *KR*, 2012.

[203] M. Rodriguez-Muro and D. Calvanese. Quest, an OWL 2 QL reasoner for ontology-based data access. In *OWLED*, 2012.

[204] M. Rodriguez-Muro, R. Kontchakov, and M. Zakharyaschev. Query rewriting and optimisation with database dependencies in ontop. In *Description Logics*, pages 917–929, 2013.

[205] R. Rosati. *The Semantic Web: Research and Applications: 9th Extended Semantic Web Conference, ESWC 2012, Heraklion, Crete, Greece, May 27-31, 2012. Proceedings*, chapter Prexto: Query Rewriting under Extensional Constraints in DL-Lite, pages 360–374. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[206] R. Rosati and A. Almatelli. Improving query answering over DL-Lite ontologies. *KR*, 10:51–53, 2010.

[207] D. J. Russomanno, C. R. Kothari, and O. A. Thomas. Building a sensor ontology: A practical approach leveraging ISO and OGC models. In *IC-AI*, pages 637–643, 2005.

[208] S. S. Sahoo, W. Halb, S. Hellmann, K. Idehen, T. Thibodeau Jr, S. Auer, J. Sequeda, and A. Ezzat. A survey of current approaches for mapping of relational databases to RDF. *W3C RDB2RDF Incubator Group Report*, pages 113–130, 2009.

[209] J. F. Sequeda and D. P. Miranker. Ultrawrap: SPARQL execution on relational data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 22:19–39, 2013.

[210] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pages 1–10. IEEE, 2010.

[211] A. Silberschatz, H. F. Korth, S. Sudarshan, et al. *Database system concepts*, volume 4. McGraw-Hill Singapore, 1997.

[212] A. Singhal. Introducing the knowledge graph: things, not strings. *Official google blog*, 2012. https://developers.google.com/knowledge-graph/ Accessed: 2016-07-15.

[213] N. Spangenberg, M. Roth, and B. Franczyk. Evaluating new approaches of big data analytics frameworks. In *International Conference on Business Information Systems*, pages 28–37. Springer, 2015.

[214] M. Sporny, G. Kellogg, M. Lanthaler, W. R. W. Group, et al. JSON-LD 1.0: a JSON-based serialization for linked data. *W3C Recommendation*, 16, 2014. https://www.w3.org/TR/json-ld/ Accessed: 2016-07-15.

[215] SRBench. SRBench wiki page. http://w3.org/wiki/SRBench/. Accessed: 2016-07-15.

[216] U. Srivastava and J. Widom. Memory-limited execution of windowed stream joins. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pages 324–335. VLDB Endowment, 2004.

[217] I. L. Stats. Twitter usage statistics. http://www.internetlivestats.com/twitter-statistics/. Accessed: 2016-07-15.

[218] M. Sullivan and A. Heybey. A system for managing large databases of network traffic. In *Proceedings of USENIX*, 1998.

[219] M. Sullivan and A. Heybey. Tribeca: a system for managing large databases of network traffic. In *In ATEC 98: Proceedings of the annual conference on USENIX Annual Technical Conference. USENIX Association, Berkeley, CA, USA, 2–2.*, 1998.

[220] M. Svensson. Benchmarking the performance of a data stream management system. Master's thesis, MSc thesis report, Uppsala University, 2007.

[221] J. Tappolet and A. Bernstein. Applied temporal RDF: Efficient temporal querying of RDF data with SPARQL. In *Proceedings of the 6th European Semantic Web Conference on The Semantic Web: Research and Applications*, ESWC 2009 Heraklion, pages 308–322, Berlin, Heidelberg, 2009. Springer-Verlag.

[222] D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continous queries over append-only databases. In *In Proceedings of SIGMOD 92, pages 321-330. ACM*, 1992.

[223] R. S. Tibbetts III. Linear road: Benchmarking stream-based data management systems. Master's thesis, Massachusetts Institute of Technology, 2003.

[224] S. Tobies. *Complexity results and practical algorithms for logics in knowledge representation.* PhD thesis, Aachen, 2001. Aachen, Techn. Hochsch., Diss., 2001.

[225] Z. Toptas. Implementation of the linear road benchmark on the basis of the real-time stream-processing system Storm. Master's thesis, Hamburg University of Technology, 2014.

[226] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, et al. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 147–156. ACM, 2014.

[227] M. M. Tsangaris, G. Kakaletris, H. Kllapi, G. Papanikos, F. Pentaris, P. Polydoras, E. Sitaridi, V. Stoumpos, and Y. E. Ioannidis. Dataflow processing and optimization on grid and cloud infrastructures. *IEEE Data Eng. Bull.*, 32(1):67–74, 2009.

[228] W3C. OWL. `https://www.w3.org/OWL/`. Accessed: 2016-07-15.

[229] W3C. OWL 2 web ontology language profiles. `https://www.w3.org/TR/owl-profiles/`. Accessed: 2016-07-15.

[230] W3C. R2RML. `https://www.w3.org/TR/r2rml/`. Accessed: 2016-07-15.

[231] W3C. RDB2RDF working group. `http://www.w3.org/2001/sw/rdb2rdf/`. Accessed: 2016-07-15.

[232] W3C. RDF schema 1.1. `https://www.w3.org/TR/rdf-schema/`. Accessed: 2016-07-15.

[233] W3C. RDF stream processing community group. `https://www.w3.org/community/rsp/`. Accessed: 2016-07-15.

[234] W3C. SPARQL. `https://www.w3.org/TR/rdf-sparql-query/`. Accessed: 2016-07-15.

[235] W3C. SPARQL 1.1. `https://www.w3.org/TR/sparql11-query/`. Accessed: 2016-07-15.

[236] W3C. Time Ontology in OWL. `https://www.w3.org/TR/owl-time/`. Accessed: 2016-04-15.

[237] W3C. W3C semantic sensor network incubator group. `https://www.w3.org/2005/Incubator/ssn/`. Accessed: 2016-07-15.

[238] S. Wandelt and R. Möller. Towards abox modularization of semi-expressive description logics. *Applied Ontology*, 7(2):133–167, 2012.

[239] C. Watterson and D. Heffernan. Runtime verification and monitoring of embedded systems. *IET software*, 1(5):172–179, 2007.

[240] W. White, M. Riedewald, J. Gehrke, and A. Demers. What is next in event processing? In *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 263–272. ACM, 2007.

[241] J. Widom et al. The stanford data stream management system. *believed to be prior to*, page 24, 2007.

[242] K. J. Witt, J. Stanley, D. Smithbauer, D. Mandl, V. Ly, A. Underbrink, and M. Metheny. Enabling sensor webs by utilizing SWAMO for autonomous operations. In *8th NASA Earth Science Technology Conference*, 2008.

[243] F. Yang, E. Tschetter, X. Léauté, N. Ray, G. Merlino, and D. Ganguli. Druid: A real-time analytical data store. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 157–168. ACM, 2014.

[244] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[245] E. Zeitler and T. Risch. Using stream queries to measure communication performance of a parallel computing environment. In *Distributed Computing Systems Workshops, 2007. ICDCSW'07. 27th International Conference on*, pages 65–65. IEEE, 2007.

[246] E. Zeitler and T. Risch. Scalable splitting of massive data streams. In *Database Systems for Advanced Applications*, pages 184–198. Springer, 2010.

[247] E. Zeitler and T. Risch. Massive scale-out of expensive continuous queries. *VLDB Endowment*, 4(11), 2011.

[248] Y. Zhang, P. Minh Duc, O. Corcho, and J. P. Calbimonte. SRBench: A Streaming RDF/SPARQL Benchmark. In *Proceedings of International Semantic Web Conference 2012*, Nov. 2012.

[249] Y. Zhu and D. Shasha. Statstream: Statistical monitoring of thousands of data streams in real time. In *Proceedings of the 28th international conference on Very Large Data Bases*, pages 358–369. VLDB Endowment, 2002.

Listings

List of Figures

2.1 Schema of an architecture for Continuous Queries 7
2.2 Schema of an architecture for continuous queries 8
2.3 General schema of a CEP network 9
2.4 General schema of CQL window and stream operator 11
2.5 Aurora graphical user interface [68] 16
2.6 Architecture of Spark components 22
2.7 Structure of the SSN ontology [76] 37
2.8 Schematic OBDA process for static data 47
2.9 Example tables and R2RML mapping in a sensor based scenario . . 56
2.10 The LARS streaming model to capture RSP queries (from [37]) . . . 70
2.11 The LARS operators as shown in [37] 71
2.12 The oracle of the CSRBench (from [88]) 76

3.1 Example data for a measurement input stream 97
3.2 Simplified syntax for STARQL (OL, ECL) 110
3.3 Grammar for STARQL HAVING clauses 114
3.4 Combination of Guards 116
3.5 Rule set for checking a formula in SRNF for range restriction from [4] 117

4.1 Schematic Transformation of STARQL queries 130
4.2 STARQL transformation architecture 137

5.1 Schematic implementation of the STARQL prototype 148
5.2 Processing pipeline of the STARQL transformation 150

6.1 Comparison of tuple translation delay 184

Listings

2.1 Basic StreaQuel query (ST = start time) 13

2.2 Basic Tapestry Algorithm for periodic query execution 18
2.3 Example for a window operator with UDFs in ExaQL 19
2.4 Continuous view in PipelineDB with simulated window 20
2.5 A query formulated in SPARQL with filter constraints 41
2.6 A query formulated in Streaming SPARQL (values of 30 minutes) . . 62
2.7 A query formulated in C-SPARQL (average of 30 minutes) 63
2.8 A query formulated in CQELS (average of 30 minutes) 63
2.9 A query formulated in SPARQLstream (average of 30 minutes) . . . 64
2.10 A query formulated in EP-SPARQL (values incrby 100 in 30 mins) . 65
2.11 A query formulated in TEF-SPARQL (values incrby 100 in 30 mins) 66
2.12 A query formulated in RSP-QL (average values of 30 minutes) 67

3.1 Basic STARQL example 1 92
3.2 Basic STARQL example 2 96
3.3 Window operator from basic STARQL example 1 97
3.4 Sequence operator from Example 1 98
3.5 STARQL example for combining multiple streams 100
3.6 Example for coarsening 101
3.7 STARQL example with static information 102
3.8 STARQL example for an aggregation operator 103
3.9 Basic STARQL HAVING example 104
3.10 Example for pulse definition 105
3.11 STARQL example for advanced multi streams 106
3.12 Example for comparison of live and recorded streams 107
3.13 Example for SELECT operator 108
3.14 Example for CONSTRUCT operator 108

4.1 Example for generating a windowed sequence 133
4.2 Example query in STARQL 137
4.3 An example for STARQL orthogonality 145

5.1 Example answer set for Query1 in JSON-LD format 153
5.2 Representation of two Siemens sensors used in a turbine installation 155
5.3 Representation of a sensor observation 155
5.4 Mapping of sensor meta data based on a turbine installation 156
5.5 Mapping of sensor observations based on the SSN ontology 156
5.6 STARQL Query **Q1** (Threshold and static data) 157
5.7 STARQL Query **Q2** (Sequencing) 158
5.8 STARQL Query **Q3** (Aggregation) 158
5.9 STARQL Query **Q4** (Orthogonality) 159
5.10 Example SQL Code for Query Example 1 in PostgreSQL 162

5.11 Transformed HAVING clause in PostgreSQL for example **Q1** 163
5.12 Transformed HAVING clause in pl/pgsql for query example1 165
5.13 Simplyfied transformation in Exareme for query **Q1** 166
5.14 Transformation result in Spark Streaming for Query Example 1 . . . 168
5.15 Simplified Java code for Spark windows 169
5.16 Transformation result for SPARK SQL 170
5.17 Transformation result for example1 in PipelineDB 171

6.1 Representation of a sensor observation 180
6.2 Example transformation for observations and results (abbreviated) . 181
6.3 Representation of a sensor observation (reified) 181
6.4 Transformation of a sensor observation (reified) 182
6.5 Representation of a sensor observation (reified) 182
6.6 Transformation of a sensor observation (non-reified) 183

A.1 Transformation results - example Q1 199
A.2 Transformation results - example Q2 200
A.3 Transformation results - example Q3 201
A.4 Transformation results - example Q4 202

B.1 Distributed window implementation - Client 205
B.2 Distributed window implementation - Server 208

C.5 SRBench - Q6 215
C.6 SRBench - Q7 215
C.1 SRBench - Q1 216
C.2 SRBench - Q2 216
C.3 SRBench - Q4 217
C.4 SRBench - Q5 217
C.7 SRBench - Q8 218
C.8 SRBench - Q9 218
C.9 SRBench - Q10 219
C.10 SRBench - Q11 219

List of Tables

2.1 Comparison of DSMS systems 25
2.2 Comparison of DSMS query languages 27
2.3 Results of application function $gr(g, \alpha)$ for applying a PI α to an UCQ atom g [61] 50

4.1 Example for incoming data in the measurement scenario 132
4.2 Example for a sliding window view in the measurement scenario . . . 132

5.1 Exareme REST API Functionality Overview 151
5.2 Example Data as used for query experiments 154
5.3 Comparison of implemented backend examples 161

6.1 Comparison of RDF-Stream query languages (Part1) 174
6.2 Comparison of RDF-Stream query languages (Part2 175
6.3 Comparison of RDF stream query languages 176
6.4 SRBench result table 177
6.5 Query rewriting delays for different query examples 185
6.6 Query times for different examples and parameters 187
6.7 Query times for distributed window execution 188