Project N^{o}:	**FP7-318338**
Project Acronym:	**Optique**
Project Title:	**Scalable End-user Access to Big Data**
Instrument:	**Integrated Project**
Scheme:	**Information & Communication Technologies**

Deliverable D5.2
OBDA with Temporal and Stream-Oriented Queries: Optimization Techniques

Due date of deliverable:	(T0+24)
Actual submission date:	October 31, 2014

Start date of the project:	**1st November 2012**	Duration:	**48 months**
Lead contractor for this deliverable:		**UoL**	
Dissemination level:		**PU** – Public	

Final version

Executive Summary:

OBDA with Temporal and Stream-Oriented Queries: Optimization Techniques

In this deliverable we summarize the results of tasks T5.2, T5.3, and T5.5. Whereas in Task T5.1 the stream-temporal query language framework (STARQL) was formally specified with a grammar and a semantics, the main objective for work package WP5 in year 2 was to fit this query framework into the OBDA paradigm by showing how to rewrite and unfold STARQL queries into queries over the backend sources (O5.2).

We show that STARQL queries can be transformed into CQL, a de-facto-standard relational stream query language, to give a theoretical account of STARQL. For practical experiments we describe the architecture of two implemented query transformation modules, both mapping STARQL to query languages supported by the ADP system, which is one of the backend engines used within the Optique platform. The first one is used for continuous queries as required by the proactive Siemens use case, the second module is used for temporal reasoning as needed by the reactive diagnosis scenario of the Siemens use case.

Authors
Ralf Möller (UoL)
Christian Neuenstadt (UoL)
Özgür L. Özçep (UoL)

Contributors
Konstantina Bereta (UoA)
Dimitris Bilidas (UoA)
Herald Kllapi (UoA)
Christoforos Sbigos (UoA)
Thomas Hubauer (SIEMENS)

Contents

Chapter 1

Introduction

The work described in this deliverable continues the work described in deliverable D5.1 and contributes results to recent efforts for extending the paradigm of ontology-based data access (OBDA) in order to handle streaming data [7, 5, 16] as well as temporal data [4, 3]. In particular, in this paper we report on the results of tasks T5.2, T5.3, and T5.5.

Considering streams the main challenge is that data cannot be processed as a whole. Hence blocking operators such as the classical grouping operator or aggregation operators cannot be directly be applied to it. The simple but fundamental idea as introduced by CQL [2] is to apply a (small) sliding window which is updated as new elements from the stream arrive at the query answering system. This idea of using stream window operators is also realized in the query language framework STARQL (Streaming and Temporal ontology Access with a Reasoning-based Query Language). Using the window approach has some nice consequences for the rewritability and unfoldability issues risen by previous efforts to adapt OBDA to streams [7, 5, 16]. The simple idea of previous approaches is that answering continuous ontology-level queries on streams can be reduced to answering ontology-level queries on dynamically updated finite window contents, which are treated as single ABoxes. Unfortunately, this approach can lead to unintended inconsistencies, as exemplified in our Siemens use case. For example, multiple measurement values for a sensor collected at different time points provide for inconsistencies since the value association should be functional. In STARQL, the idea of processing streams with a window operator is pushed further by defining a finite *sequence of consistent ABoxes* for each window.

Considering temporal reasoning for reactive diagnostics in the Siemens use case we found that a window based approach leads to elegant solutions as well. A window is used to focus on some subset of temporal data in a temporal query. Now, over time, different foci are relevant for reactive diagnosis. Thus, foci changes can be modelled by a window virtually "sliding" over temporal data, albeit the case that sliding is not done in realtime. Indeed, internally, all window instances could be evaluated in parallel while, from a user perspective, the window "slides" over temporal data (also called historical data). Obviously, in this case start time points and end time points can be specified for analyses of temporal data. Thus, our query language framework STARQL, for which we gave a full syntax and semantics in Deliverable D5.1 (see also corresponding publications [12, 11]) was defined such that it can be used equally for temporal (or historical) and stream reasoning. The semantics of STARQL does not distinguish between the realtime and the historical querying scenario—hence querying with STARQL on historical data is going to give the same answers as if STARQL were run on the same data in real-time.

The ABox sequencing strategy for windows required to avoid inconsistencies, as argued above, makes rewriting and, more importantly, unfolding of STARQL queries a challenging task. In particular, one may ask whether it is possible to rewrite and unfold one STARQL query into a single backend query formulated in the query language provided by the backend systems. In fact, in this deliverable, we demonstrate that such a one-to-one transformation is possible for the ADP system developed by the UoA partner, and we present the STARQL engine prototype, integrated into the Optique platform, for realizing this transformation.

Deliverables D8.1 and D1.1, for the Siemens use case and the Siemens requirements, respectively, specify engineers' information needs. In particular the deliverables contain a catalog of queries which Siemens engineers need to get answered. We conducted tests showing that around 70 percent of the Siemens queries can be handled right now by the STARQL prototype (see also [9]). Also, for these queries, the prototype produces the answers in a reasonably short amount of time. For the remaining queries, the STARQL prototype will be extended to better support numerical aggregate functions.

The deliverable is structured as follows. After this introduction we give a recap on STARQL's syntax and semantics (Chapter 2). Chapter 3 discusses safety conditions as introduced in [15, 14]. In Chapter 4 we sketch the transformation of STARQL into CQL-like languages. Chapter 5 describes the architecture of the standalone STARQL prototype and discusses the experimental results for temporal reasoning with it. In the last chapter before the conclusion (Chapter 6), we discuss the architecture and the experimental results with STARQL prototype engine integrated into the Optique platform.

Chapter 2

The STARQL Framework

We recap the syntax and the semantics for a fragment of STARQL, ignoring among others macro definitions, aggregators etc. (see [13] for the full version). But note that, here, we use a slightly different syntax within the HAVING clauses, where we overload the subgraph key word `GRAPH` known from SPARQL. This is a purely syntactical decision. A full translation of STARQL into SPARQL (with some additional extensions to handle streams), was not in the focus of the work related to the relevant tasks of this deliverable, because the status of the Optique platform does not demand such a translation: in particular the visual interface cannot express full SPARQL queries.

Here and in the following chapters we assume familiarity with the description logic DL-Lite [6].

For illustration purposes, our running example is a measurement scenario in which there is a (possibly virtual) stream S_{Msmt} of ABox assertions. Its initial part, called $S_{Msmt}^{\leq 5s}$ here, contains timestamped ABox assertions giving the value of a temperature sensor s_0 at 6 time points starting with $0s$.

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

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

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

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

Though the monotonicity condition seems simple, it should be noted that recent approaches for temporal DL-Lite logics as that of [4] cannot express it.[1]

[1]The reason in case of [4] is that it does not provide an all-quantifier or a negation operator on top of the bindings produced in the ABoxes.

Convention: In order to make the examples more readable, we are going to leave out sometimes the prefix declarations. Moreover, rather than using the awkward xsd notation for durations or for the date times we use more abstract notations. Additionally, we write variables with a leading question mark only in the listings for STARQL queries. In the discussion of the semantics and the safety conditions, where we use the DL oriented syntax, we leave out the question marks.

2.1 Syntax

The example demonstrates many of the syntactical possibilities within STARQL whose grammar is sketched in Fig. 2.1. The rules for the HAVING clause are not given there but are discussed in more detail in the following sections.

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

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

Within the WHERE clause one can bind variables w.r.t. the non-streaming sources (ABox, TBox) mentioned in the `FROM` clause by using (unions) of BGPs. We assume an underlying DL-Lite logic for the static ABox, the TBox and the BGP (considered as unions of conjunctive queries UCQs) which allows for concrete domain values, e.g., DL-Lite$_{\mathcal{A}}$ [6]. In this example, instantiations of the sensors $?s$ are fixed w.r.t. a static ABox and a TBox given by URIs.

The heart of the STARQL queries is the window operator in combination with the sequencing mechanism. In the example, the operator `[NOW-2s, NOW]->"1S"^^xsd:duration` describes a sliding window operator, which collects the timestamped ABox assertions in the last two seconds and then slides 1s forward in time. Every temporal ABox produced by the window operator is converted to a sequence of (pure) ABoxes. At every time point, one has a sequence of ABoxes on which temporal (state-based) reasoning can be applied. This is realized in STARQL by a sorted first-order logic template in which state stamped UCQs conditions are embedded. We use here again the `GRAPH` notation from SPARQL. In the example above, the HAVING clause expresses a monotonicity condition stating that for all values $?x$ that are values of sensor $?s$ w.r.t the i^{th} ABox (subgraph) and for all values $?y$ that are values of the same sensor $?s$ w.r.t. the j^th ABox (subgraph), it must be the case that $?x$ is less than or equal to $?y$.

2.2 Semantics

STARQL queries have streams of ABox assertions (RDF triples) as input and output. So, the semantics for STARQL has to explicate how the output stream of ABox assertions is computed from the input streams. Using compositionality, the semantics definition for STARQL can be accomplished by defining the semantic denotations for the substructures of the query and then by composing them to the denotation of the whole query. We sketch the semantics of the window operator, of the sequencing, and of the HAVING clause.

A stream of ABox assertions is an infinite set of timestamped ABox assertions of the form $ax\langle t\rangle$. The timestamps stem from a flow of time $(T, \leq)$ where T may even be a dense set and where $\leq$ is a linear order. Let S be a stream name with its denotation $[\![S]\!]$ being such a stream of timestamped ABox assertions. We declare the denotation of the windowed stream $ws = S\ winExp = S$ `[`$timeExp_1, timeExp_2$`]->`sl as a stream of temporal ABoxes, where a temporal ABox is a set of timestamped assertions.

Let $\lambda t.g_1(t) = [\![timeExp_1]\!]$ and $\lambda t.g_2(t) = [\![timeExp_2]\!]$ be the functions corresponding to the time expressions. We did not explicate rules for the time expressions $timeExp_1, timeExp_2$, but mention here that these may be any reasonable arithmetical terms using possibly the variable `NOW` and leading to a well defined

$$
\begin{array}{rcl}
\textit{starqlQuery} & \longrightarrow & [\textit{prefixDeclaration}]\ \textit{createExp} \\
\textit{createExp} & \longrightarrow & \texttt{CREATE STREAM}\ \textit{name}\ \texttt{AS}\ \textit{constrExp} \\
\textit{pulseExp} & \longrightarrow & \texttt{PULSE WITH START = }\textit{startTime}[,\texttt{END = }\textit{endTime}]\texttt{, FREQUENCY = }\textit{freq} \\
\textit{constrExp} & \longrightarrow & \texttt{CONSTRUCT}\ \textit{constrHead}(\vec{x},\vec{y}) \\
 & & \ \texttt{FROM}\ \textit{listWinStreamExp}\ [\ \texttt{,}\ \textit{listOfRessources}] \\
 & & [\texttt{USING}\ \textit{pulseExp}] \\
 & & \ \texttt{WHERE}\ \textit{whereClause}(\vec{x}) \\
 & & \ \texttt{SEQUENCE BY}\ \textit{seqMethod}\ [\texttt{HAVING}\ \textit{safeHavingClause}(\vec{x},\vec{y})] \\
\textit{constrHead}(\vec{x},\vec{y}) & \longrightarrow & \texttt{GRAPH}\ \textit{timeExp}\ \textit{BGP}(\vec{x},\vec{y})\ [\ \texttt{,}\ \textit{constrHead}] \\
\textit{listWinStreamExp} & \longrightarrow & (\textit{name} \mid \textit{constrExp})\textit{windowExp}\ [\texttt{START = }\textit{startTime}\ \texttt{END = }\textit{endTime}] \\
 & & [\ \texttt{,}\ \textit{listWinStreamExp}] \\
\textit{windowExp} & \longrightarrow & \texttt{[}\ \textit{timeExp}_1, \textit{timeExp}_2\ \texttt{]->}\textit{sl} \\
\textit{listOfRessources} & \longrightarrow & \textit{typedRessourceList}[\ \texttt{,}\ \textit{listOfRessources}] \\
\textit{typedRessourceList} & \longrightarrow & \texttt{STATIC ABOX}\ \textit{listofURIstoStaticABoxes} \mid \\
 & & \texttt{TEMPORAL ABOX}\ \textit{listofURIstoTemporalABoxes} \mid \\
 & & \texttt{TBOX}\ \textit{listofURIstoTBoxes} \\
\textit{whereClause}(\vec{x}) & \longrightarrow & \Psi(\vec{x}) \quad (\Psi(\vec{x})\text{ a union of BGPs with distinguished variables }\vec{x}) \\
\textit{seqMethod} & \longrightarrow & \textit{StdSeq} \mid \textit{SeqMethod}(\sim)
\end{array}
$$

Figure 2.1: Syntax for STARQL (without HAVING clauses)

interval. The pulse declaration defines a subset $T' \subseteq T$ of the time domain. Assume, it has the following form

```
USING PULSE WITH START = st, FREQUENCY = fr
```

T' is the set of timestamps of the stream of temporal ABoxes $[\![ws]\!]$. Let T' be represented by the increasing sequence of timestamps $(t_i)_{i\in\mathbb{N}}$, where t_0 is the starting point fixed in the pulse declaration.

Now, one defines for every t_i the temporal ABox $\tilde{\mathcal{A}}_{t_i}$ such that $(\tilde{\mathcal{A}}_{t_i}, t_i) \in [\![wS]\!]$. If $t_i < sl - 1$, then $\tilde{\mathcal{A}}_{t_i} = \emptyset$. Else set first $t_{start} = \lfloor t_i/sl \rfloor \times sl$ and $t_{end} = max\{t_{start} - (g_2(t_i) - g_1(t_i)), 0\}$, and define on that basis

$$\tilde{\mathcal{A}}_{t_i} = \{(ass, t) \mid (ass, t) \in [\![S]\!] \text{ and } t_{end} \le t \le t_{start}\}$$

In our example, $timeExp_1 = NOW - 2s$ (so $[\![timeExp_1]\!] = \lambda t.t - 2$), $timeExp_2 = NOW$ and $sl = 1s$; the example's results for second 4s and 5s are the following.

Time	Temporal ABox
$4s$	$\{val(s_0, 94°)\langle 2s\rangle, val(s_0, 92°)\langle 3s\rangle, val(s_0, 93°)\langle 4s\rangle\}$
$5s$	$\{val(s_0, 92°)\langle 3s\rangle, val(s_0, 93°)\langle 4s\rangle, val(s_0, 95°)\langle 5s\rangle\}$

If the STARQL query refers to more than one stream, then these are joined by time-wise union of the temporal ABoxes of the windowed streams, which is possible as the pulse declaration synchronizes all streams of temporal ABoxes.

This window semantics is that described in Deliverable D5.1 and mimics the window operator definition for CQL [2]. From the implementation point of view, an operational semantics is more helpful—at least it gives a different perspective on the intended semantics of the window. Hence, we illustrate the window semantics within a more operational mode. Let us assume that the following query template is given:

```
CREATE STREAM S_{out}

...
FROM S [NOW-wr, NOW] -> sl
USING PULSE WITH START = st, FREQUENCY = fr
...
```

So, $timeExp_1$ = `NOW-wr`, where `wr` is a constant denoting the window range, and $timeExp_2$ = `NOW`. We distinguish a pulse time t_{pulse}, and a stream time t_{str}. (For more than one stream one would have more local stream times.) The pulse time t_{pulse} evolves regularly according to the frequency specification, $t_{pulse} = st \longrightarrow st + fr \longrightarrow st + 2fr \longrightarrow \dots$. In contrast, the stream time t_{str} is jumping/sliding and is determined by the trace of endpoints of the sliding window. More concretely, if Let t'_{end} denotes the next stream time, then the evolvement of t_{str} is specified as follows:

$$t_{str} \xrightarrow{\text{IF } t_{str} + m \times sl \leq t_{pulse} \text{ (for } m \in \mathbb{N} \text{ maximal)}} t_{str} + m \times sl$$

The window contents at t_{pulse} is given by: $\{ax\langle t\rangle \in S_{Msmt} \mid t_{str} - wr \leq t \leq t_{str}\}$. Note that always $t_{str} \leq t_{pulse}$. This a crucial point which guarantees that applying the window on real-time streams does not give different stream elements than applying the window on a simulated stream from a DB with historical data.

To make this more concrete assume the following instantiation of the query:

```
CREATE STREAM S_{out} AS
...
FROM STREAM S  [NOW-3s, NOW] -> "3S"^^xsd:duration
USING PULSE WITH START =  "2005-01-01T00:00:00CET"^^xsd:dateTime,
                 FREQUENCY = "2S"^^xsd:duration
...
```

Then we get the following evolvements of the pulse time and the streaming time.

$$\begin{array}{rl} t_{pulse}: & 0s \to 2s \to 4s \to 6s \to 8s \to 10s \to 12s \to \\ t_{str}: & 0s \to 0s \to 3s \to 6s \to 6s \to \ 9s \ \to 12s \to \end{array}$$

An example for multiple streams demonstrates the synchronization effect of the pulse.

```
CREATE STREAM Sout AS
CONSTRUCT { ?sens rdf:type :RecentMonInc }<NOW>
FROM  STREAM S_{Msmt_1} [NOW-3s, NOW]->"3S"^^xsd:duration,
      STREAM S_{Msmt_2} [NOW-3s, NOW]->"2S"^^xsd:duration
USING  PULSE WITH START  = "2005-01-01T00:00:00CET"^^xsd:dateTime,
                  FREQUENCY = "2S"^^xsd:duration
SEQUENCE BY StdSeq AS seq
HAVING  (...)
```

$$\begin{array}{rl} t_{pulse}: & 0s \to 2s \to 4s \to 6s \to 8s \to 10s \to 12s \to \\ t_{S_{Msmt_1}}: & 0s \to 0s \to 3s \to 6s \to 6s \to \ 9s \ \to 12s \to \\ t_{S_{Msmt_2}}: & 0s \to 2s \to 4s \to 6s \to 8s \to 10s \to 12s \to \end{array}$$

The stream of temporal ABoxes is the input for the sequencing operator which produces for every time point of the pulse a sequence of (pure) ABoxes. The sequencing methods used in STARQL refer to an equivalence relation $\sim$ to specify which assertions go into the same ABox. The equivalence classes $[x]_\sim$ for $x \in T$ form a partition of T. We restrict the class of admissible equivalence relations to those $\sim$ that respect the time ordering, i.e., the equivalence classes under $\sim$ should be intervals on the time domain.

Now, we define the sequence of ABoxes generated by $seqMethod(\sim)$ on the stream of temporal ABoxes as follows: Let $(\tilde{\mathcal{A}}_t, t)$ be the temporal ABox at t. Let $T' = \{t_1, \dots, t_l\}$ be the time points occurring in $\tilde{\mathcal{A}}_t$

and let k the number of equivalence classes generated by the time points in T'. Then define the sequence at t as $(\mathcal{A}_0, \dots, \mathcal{A}_k)$ where for every $i \in \{0, \dots, k\}$ the pure ABox $\mathcal{A}_i$ is

$$\mathcal{A}_i = \{ax\langle t'\rangle \mid ax\langle t'\rangle \in \tilde{\mathcal{A}}_t \text{ and } t' \text{ in } i^{th} \text{ equivalence class}\}$$

In the example above, the equivalence is the identity (keyword `StdSeq` for standard sequencing), so that the resulting sequence of ABoxes at time point 5s is trivial as there are no more than two ABox assertions with the same timestamp: $\{val(s_0, 92°)\}\langle 0\rangle, \{val(s_0, 93°)\}\langle 1\rangle, \{val(s_0, 95°)\}\langle 2\rangle$.

A non-standard sequencing strategy may be any congruence $\sim$ on time domain. For example, define $x \sim y$ iff $\lfloor x/2 \rfloor = \lfloor y/2 \rfloor$ for all $x, y \in T = \mathbb{N}$. Then the temporal ABoxes for seconds 4 and 5 from the example above are as follows.

Time	Temporal ABox (Window contents before sequencing)
$4s$	$\{val(s_0, 94°)\langle 2s\rangle, val(s_0, 92°)\langle 3s\rangle, val(s_0, 93°)\langle 4s\rangle\}$
$5s$	$\{val(s_0, 92°)\langle 3s\rangle, val(s_0, 93°)\langle 4s\rangle, val(s_0, 95°)\langle 5s\rangle\}$

Time	Window contents after $\sim$ sequencing
$4s$	$\{\{val(s_0, 94°), val(s_0, 92°)\}\langle 0\rangle, \{val(s_0, 93°)\}\langle 1\rangle\}$
$5s$	$\{\{val(s_0, 92°)\}\langle 0\rangle, \{val(s_0, 93°), val(s_0, 95°)\}\langle 1\rangle\}$

STARQL's semantics for the HAVING clauses relies on the certain answer semantics (see [6]) for the embedded UCQs. The idea is to view the tuples in the certain answer sets as members of a sorted FOL structure $\mathcal{I}_t$. Assume that the sequence of ABoxes at some given time point t is $seq = (\mathcal{A}_0, \dots, \mathcal{A}_k)$. Then the domain of $\mathcal{I}_t$ consists of the index set $\{0, \dots, k\}$ as well as the set of all individual constants and all value constants of the signature. Now, if the HAVING clause contains, for example, the state tagged condition query $val(s, x)\langle i\rangle$ (with embedded UCQ $val(s, x)$), then we introduce for it a ternary relation symbol R and replace $val(s, x)\langle i\rangle$ by $R(s, x, i)$ in the HAVING clause. This symbol is denoted in $\mathcal{I}_t$ by the certain answers of the embedded query extended with the index i: $R^{\mathcal{I}_t} = \{(a, b, i) \mid (a, b) \in cert(val(s, x), \mathcal{A}_i \cup \mathcal{A}_{static} \cup \mathcal{T})\}$. Constants are denoted by themselves in $\mathcal{I}_t$. This already 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 the substitutions) on the structure $\mathcal{I}_t$.

Chapter 3

A Safe Fragment for STARQL HAVING clauses

As demonstrated with the monotonicity example, STARQL allows a sorted FOL to reason on the ABox sequences. The semantics for the HAVING clauses rests on the structure $\mathcal{I}_t$ whose domain $\Delta^{\mathcal{I}_t}$ does not consist only of the individual and value constants in the interpretations of the relations, the so called *active domain* according to database terminology [1], but the whole set *Dom* of individual constants, value constants and the indices produced in the sequencing. With a safety mechanism on the HAVING clauses it can be guaranteed that the evaluation of the HAVING clause on the ABox sequence depends only on the active domain, i.e., HAVING clauses are domain independent (d.i.). Formally, a query q is d.i. iff for all interpretations $\mathcal{I}_1, \mathcal{I}_2$ having 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 $\mathcal{I}_1$ is the same as the answers for q in $\mathcal{I}_2$. Without a safety mechanism, a `HAVING` clause of the form $?y > 3$, with free concrete domain variable $?y$, would be allowed: the set of bindings for $?y$ would be infinite, namely, the set of all real number bigger than 3. In particular, $?y > 3$ is not d.i.

Figure 3.1 contains the grammar for the HAVING clauses with its safety mechanism realized by variable guards/adornments. This grammar was introduced in the paper [15]. The safe HAVING clauses (denoted by the start symbol *safeHavingClause*) contain only those variables for individuals that have guard status +. We illustrate the meaning of the rules with Rule (3.1) for the `OR` case and then go in more detail w.r.t. adornments.

$$hCl(\vec{z}^{\vec{g}^1 \vee \vec{g}^2}) \longrightarrow hCl(\vec{z}^{\vec{g}^1}) \texttt{ OR } hCl(\vec{z}^{\vec{g}^2}) \tag{3.1}$$

This rule says the following: if during the derivation of a formula starting 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 clauses under some conditions on the variables $\vec{z}$ occurring in them. More concretely: If during production a clause $hCl(\vec{z}^{\vec{g}})$ with variables $\vec{z}$ and some adornment $\vec{g}$ for them is reached, then Rule (3.1) 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., if $\vec{g}$ is the result of applying a function $\vee$ on the adornment lists $\vec{g}^1, \vec{g}^2$.

The adornments $\vec{g} = g_1, \ldots, g_n$ are lists of guard status g_i (g-status for short), 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 $\vec{g} = g_1, \ldots, g_n$. We assume the ordering $\emptyset \preceq -- \preceq - \preceq +$ on the guards. This ordering is relevant for the calculation of g_{max} in the rule of Fig. 3.1 where the clause is constructed from an arbitrary clause hCl and an identity atom. 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.

The meanings of the functions $\neg, \vee, \wedge, \rightarrow$ over vectors of g-status are fixed by the tables in Figure 3.2. Combinations with the g-status $\emptyset$ is handled in an extra table 3.2b. So for example, assume that one has produced a HAVING clause $F(x_1^{--}, x_2^{+}, x_3^{-})$, where x_1 has g-status $--$, x_2 has g-status $+$, and x_3 has g-status $-$. Then rule (3.1) and the tables allow, e.g., the production of $F_1(x_1^{--}, x_2^{+}, x_3^{-})$ `OR` $F_2(x_1^{+}, x_2^{+}, x_3^{\emptyset})$. Let us verify this for the variable x_1: Its g-status $--$ in F_1 and its g-status $+$ in F_2 combines to the g-status $-- = -- \vee +$ in F—according to the entry for the pair $(--, +)$ in the table of $\vee$.

Now, we will show how to transform HAVING clauses to SQL. In particular this shows that the HAV-

$$
\begin{array}{rcl}
\mathit{safeHavingClause}(\vec{z}) & \longrightarrow & \mathit{hCl}(\vec{z}^{+}) \quad (\text{for } \vec{z} \in Var_{val} \cup Var_{ind}) \\
 & & \\
\mathit{term}(i^{+}) & \longrightarrow & i \\
\mathit{term}() & \longrightarrow & \texttt{max} \mid \texttt{0} \mid \texttt{1} \\
\mathit{stateAtom}(\vec{y}^{+}, i^{+}) & \longrightarrow & \texttt{GRAPH i } \Psi(\vec{x}, \vec{y}) \\
 & & (\text{for a UCQ } \Psi(\vec{x}, \vec{y}) \text{ and} \\
 & & \vec{x} \subseteq X,\ \vec{y} \subseteq Var_{ind} \cup Var_{val} \setminus X) \\
\mathit{stateAtom}(x^{--}, y^{--}) & \longrightarrow & x = y \quad (\text{for } y, x \notin X \cup Var_{val}) \\
\mathit{stateAtom}(x^{+}) & \longrightarrow & x = a \mid a = x \\
 & & (\text{for } a \in (X \cap Var_{ind}) \cup Const_{const},\ x \in Var_{ind} \setminus X) \\
\mathit{vAtom}(z_1^{+}) & \longrightarrow & z_1 = v \mid v = z_1 \\
 & & (\text{for } z_1 \in Var_{val} \setminus X \text{ and } v \in Const_{val}) \\
\mathit{vAtom}(z_1^{+}) & \longrightarrow & z_1 = z_2 \mid z_2 = z_1 \\
 & & (\text{for } z_1 \in Var_{val} \setminus X, \text{ and } z_2 \in X \cap Var_{val}) \\
\mathit{vAtom}(z_1^{--}, z_2^{--}) & \longrightarrow & z_1 \ \mathit{op} \ z_2 \\
 & & (\text{for } \mathit{op} \in \{\texttt{<,<=, >, >=, =}\};\ z_1, z_2 \in Var_{val} \setminus X) \\
\mathit{vAtom}(z_1^{--}) & \longrightarrow & z_1 \ \mathit{op} \ z_2 \quad (\text{for } \mathit{op} \in \{\texttt{<,<=, >, >=}\},\ z_1 \in Var_{val} \setminus X, \\
 & & \qquad\qquad z_2 \in Val_{const} \cup (X \cap Val_{var})) \\
\mathit{stateArithAtom}(i_1^{g_1}, i_2^{g_2}) & \longrightarrow & \mathit{term}_1(i_1^{g_1}) \ \mathit{op} \ \mathit{term}_2(i_2^{g_2}) \\
 & & (\text{for } \mathit{op} \in \{\texttt{<,<=, =, >, >=}\}) \\
\mathit{stateArithAtom}(i_1^{g_1}, i_2^{g_2}, i_3^{g_3}) & \longrightarrow & \texttt{plus(}\mathit{term}_1(i_1^{g_1}), \mathit{term}_2(i_2^{g_2}), \mathit{term}_3(i_3^{g_3})\texttt{)} \\
 & & \\
\mathit{hCl}(\vec{z}^{\vec{g}}) & \longrightarrow & \mathit{stateAtom}(\vec{z}^{\vec{g}}) \mid \mathit{vAtom}(\vec{z}^{\vec{g}}) \mid \mathit{stateArithAtom}(\vec{z}^{\vec{g}}) \\
\mathit{hCl}(\vec{z}^{\vec{g}^1 \vee \vec{g}^2}) & \longrightarrow & \mathit{hCl}(\vec{z}^{\vec{g}^1}) \ \texttt{OR} \ \mathit{hCl}(\vec{z}^{\vec{g}^2}) \\
\mathit{hCl}(\vec{z}^{\vec{g}^1 \wedge \vec{g}^2}) & \longrightarrow & \mathit{hCl}(\vec{z}^{\vec{g}^1}) \ \texttt{AND} \ \mathit{hCl}(\vec{z}^{\vec{g}^2}) \quad (\text{both conjuncts are} \\
 & & \text{not of form } x = y \text{ for } x, y \in Var_{ind} \cup Var_{val}) \\
\mathit{hCl}(z_1^{g_{max}}, z_2^{g_{max}}, \vec{z_3}^{g_3}) & \longrightarrow & \mathit{hCl}(z_1^{g_1}, z_2^{g_2}, \vec{z_3}^{g_3}) \ \texttt{AND} \ z_1^{h_1} = z_2^{h_2} \\
 & & (\text{for } g_{max} = max\{g_1, g_2, h_1, h_2\}) \\
\mathit{hCl}(\vec{z}^{\neg \vec{g}}) & \longrightarrow & \texttt{NOT} \ \mathit{hCl}(\vec{z}^{\vec{g}}) \\
\mathit{hCl}(\vec{z}^{\vec{g}^1 \rightarrow \vec{g}^2}) & \longrightarrow & \texttt{IF} \ \mathit{hCl}(\vec{z}^{\vec{g}^1}) \ \texttt{THEN} \ \mathit{hCl}(\vec{z}^{\vec{g}^2}) \\
\mathit{hCl}(\vec{z}^{\vec{g}^1 \rightarrow \vec{g}^2}) & \longrightarrow & \texttt{FORALL} \ \tilde{y} \ \texttt{IF} \ \mathit{hCl}(\vec{z}^{\vec{g}^1}, \vec{y}^{+}) \ \texttt{THEN} \ \mathit{hCl}(\vec{z}^{\vec{g}^2}, \vec{y}^{g}) \\
\mathit{hCl}(\vec{z}^{\vec{g}^1 \wedge \vec{g}^2}) & \longrightarrow & \texttt{EXISTS} \ \tilde{y} \ \mathit{hCl}(\vec{z}^{\vec{g}^1}, \vec{y}^{+}) \ \texttt{AND} \ \mathit{hCl}(\vec{z}^{\vec{g}^2}, \vec{y}^{g})
\end{array}
$$

Figure 3.1: Grammar for `HAVING` clauses
(the set of variables X is the set of variables that are bounded by the WHERE clause in the STARQL query)

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 3.2: Combination of Guards

ING clause language is d.i. as SQL is d.i.For a formula F let $SRNF(F)$ be the formula in *safe range normal form (SRNF)* [1, 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 $F = SRNF(F)$.

Domain independence for formulas in SRNF is handled in the literature [1] also by a guard concept. This is realized by a function rr as follows.

1. $rr(r(t_1, \ldots, t_n))$ = 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. $rr(\texttt{EXISTS } \vec{x} F) = \begin{cases} rr(F) \setminus \vec{x} & \text{if } \vec{x} \subseteq rr(F) \\ \text{return } \perp & \text{else} \end{cases}$

The definition of rr in [1] are simpler than our adornment technique used in the grammar because of two main reasons: the authors in [1] assume that the formula is already in SRNF form, whereas we do not. Moreover, we define the HAVING clause grammar in the context of the grammar for STARQL queries. So, we have to take care of variables X that are already bounded by the WHERE clause. This leads to many sub-cases in our grammar.

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).

Relating our set of g-status with the set of g-status used in [1] leads to the desired theorem.

Theorem 3.0.1 *All safe HAVING clauses (considered as queries on the DB $\mathcal{I}_t$ of certain answers within the actual ABox sequence at t) are d.i.*

Let $safehCl(\vec{u}^{+})$ be a safe HAVING clause. Let $safehClNF(\vec{u})$ be the formula resulting from applying the SRNF normalization rules. The status of all the guards are not changed by the rules. Now, 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 $safehcLNF(\vec{u})$. 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 G in Fig. 3.1 and checking that only those with g-status + are in $rr(G)$. Hereby, variables $x \in X$, where X is defined in the grammar, are treated as constants in the definition of $rr(\cdot)$. The case of conjunction is clear too as any + g-status combines with any other g-status to +. Now take negation $G = \texttt{NOT}\ F(\vec{x}^{+}, \vec{y}^{-}, \vec{z}^{--})$. The definition of rr for the negation case says $rr(G) = \emptyset$. Actually we know that F is an atomic formula. Looking at all variables for these formulas in the grammar we see that no one of these as g-status $-$, hence actually $\vec{y} = \emptyset$ and we have $G(\vec{x}^{-}, \vec{z}^{--})$, so there is no variable in G with g-status +, hence indeed we get that $rr(G) = \emptyset =$ the variables in G with g-status +. The case for disjunction is clear as a positive g-status results for a variable in a disjunction only if both variables exists in the disjuncts and are labelled +. Now the last case is that of the existential quantifier $G = \texttt{EXISTS}\ xF(\vec{x}^{+}, \vec{y}^{-}, \vec{z}^{--})$. According to induction assumption $rr(F) = \vec{x}$. G may result from a transformation of an exists subformula $\texttt{EXISTS}\ xhCl(x^{+}, \dots)\ \texttt{AND}\ F'$ in $hcl(\vec{u}^{+})$. So the variable x is by definition in the set $\vec{x}$ of variables in F with g-status +, hence $rr(G) = rr(F) \setminus \{x\}$. But G 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. Now G may also result from applying somewhere the rule `FORALL` $\equiv$ `NOT EXISTS NOT`. But again, there is a formula with variables that have g-status + and are bounded by the all quantifier so that one gets again a formula of the form $\texttt{EXISTS}\ xhCl(x^{+}, \dots)\ \texttt{AND}\ F'$.

Chapter 4

Stream Reasoning by Mapping STARQL to CQL

Having shown domain independence for the HAVING clause language is the main step towards using STARQL for OBDA in the classical sense according to which queries on the ontology level are rewritten and unfolded into queries over the data source. As a data source we consider any database system providing a declarative language with stream capabilities. For the STARQL engine which is integrated into OPTIQUE platform and which is discussed in this section, the relevant language is the one provided by the stream extended version of the ADP system. In the standalone STARQL prototype, which (as of now) is used for temporal reasoning only, it may be the SQL language supported by PostgreSQL or it may be Datalog. In fact the transformation to SQL is done via Datalog (more concretely Prolog). For the details we refer the reader to Chapter 5.

4.1 CQL

CQL [2] is one of the early relational stream query languages having served as a blue print for many stream query languages even on the ontological level (e.g., [7],[5],[16]). We are going to describe CQL in more detail in order to understand the general idea of the unfolding strategy and in order to exactly see which additional and necessary features the stream query language of ADP adds to CQL.

Next to streams, CQL presupposes a data structure called *relation*; this data structure lifts relations of classical relational algebra to the temporal setting; formally, let be given a classical relational schema, then a (temporal) relation R is a total function from the time domain T to the set of finite bags (multi-sets) of tuples over the given schema. The (classical) relation at time point $t \in T$ is called *instantaneous relation*.

CQL defines operators that map streams to relations (the most important being the window operator), operators that map relations to streams, and moreover relation-to-relation operators, which adapt the usual relational algebra operators to the temporal setting. Stream-to-stream operators can be simulated by the given operators.

The most important stream-to-relation construct is that of a sliding window. Let S denote a stream and wr be an element of the time domain T. The time-based sliding window operator with window range wr is denoted by `[Range wr]` and attached to stream arguments in post-fix notation. The relation R denoted by `S [Range wr]` is defined for every $t \in T$ as follows:

$$R(t) = \{s \mid (s, t') \in S \text{ and } (t' \leq t) \text{ and } t' \geq max\{t - wr, 0\}\}$$

So the bag of tuples at time point t consists of all tuples from S whose timestamps are in the interval $[t - wr, t]$—with an intuitive handling of the cases of all t with $t \leq wr$. The special case of a window with zero window range is also denoted by `[Now]`; the case of an unbounded window by `[Unbound]`.

The following example illustrates the effects of time sliding windows. Let be given a stream S of timestamped tuples having the form $(sensor, value)\langle time\rangle$. The smallest time granularity of time measurements is seconds, so we can presuppose that T is given by the natural numbers standing for time points measured in seconds. Now let the stream S start as follows:

$$\begin{aligned} S \; = \; & \{(s_0, 80°)\langle 0\rangle, (s_1, 93°)\langle 0\rangle, (s_0, 81)\langle 1\rangle, (s_0, 82°)\langle 2\rangle, (s_0, 83°)\langle 3\rangle, \\ & (s_0, 85°)\langle 5\rangle, (s_0, 86°)\langle 6\rangle....\} \end{aligned}$$

Then the relation $R =$ `S [Range 2]` is given by:

t :	0	1	2	3	4	5	6
$R(t)$:	$\{(s_0, 80), (s_1, 93)\}$	$\{(s_0, 80), (s_1, 93), (s_0, 81)\}$	$\{(s_0, 80), (s_1, 93), (s_0, 81), (s_0, 82)\}$	$\{(s_0, 81), (s_0, 82), (s_0, 83)\}$	$\{(s_0, 82), (s_0, 83)\}$	$\{(s_0, 82), (s_0, 83), (s_0, 85)\}$	$\{(s_0, 83), (s_0, 85), (s_0, 86)\}$

Please note, that the bag-approach for defining the relation is not unproblematic; the bags are timestamp agnostic, i.e., the tuples within a bag do not contain timestamps anymore. Formulating functionality constraints then becomes a demanding issue as these are not formulated w.r.t. a specific schema but directly over the domain. Take, for example, in a measurement scenario the functionality dependency (fd) constraint on measurement tuples with the content that a sensor can have maximally one value for every time point, respectively. Applying a window operator to the stream measurement may lead to a bag of tuples where the same sensor has more than one value. So, streaming these tuples out again (with e.g. the RStream operator below) may lead to a stream of tuples violating the fd constraint.

A generalized version of the sliding window allows the specification of the sliding parameter, i.e. the frequency at which the window is slided forward in time. The semantics of `R = S [Range wr Slide sl]` is as follows:

$$R(t) \;=\; \begin{cases} \emptyset & \text{if } t < sl \\ \{s \mid (s, t') \in S \text{ and } max\{\lfloor t/sl \rfloor \cdot sl - wr, 0\} \leq t' \leq \lfloor t/sl \rfloor \cdot sl\} & \text{else} \end{cases}$$

So the sliding window operator takes snapshots of the stream every sl second; for all time points in between multiples of sl the snapshot is the same. The relation of the window range parameter wr and the slide parameter sl determines the effects of the sliding window; if t$wr = sl$, then the window is tumbling, if $wr > sl$, then the window contents are overlapping, an if $sl > wr$, then the operator has a sampling effect. For other stream-to-relation operators such as the tuple based window operators or partition operators, the reader is referred to the original article [2].

Relation-to-relation operations are the usual ones known from SQL adapted to the new temporal relation setting by using them time-point wise on the instantaneous relations. In particular, the join of (temporal) relations is done pointwise. For example, the CQL expression in Listing 4.1 joins the relations of the window application results to a measurement and an event stream, filters a subset out according to a condition and projects out the sensor IDs.

```
SELECT m.sensorID
FROM Msmt[Range 1] as m, Events[Range 2] as e
WHERE m.val > 30 AND e.category = Alarm AND
      m.sensorID = e.sensorID
```

Listing 4.1: Example relation-to-relation operators in CQL

As relation-to-stream operators, the authors of [2] define *Istream* (giving a stream of newly inserted tuples w.r.t. the last time point), *Dstream* (giving a stream of newly deleted tuples w.r.t. the last point) and

Rstream (returning all elements in the relation). Assuming that for $t < 0$ one specifies $R(t) = \emptyset$ for $t < 0$ the formal definitions are:

$$\begin{aligned} \mathit{Istream}(R) &= \bigcup_{tinT} (R(t) \setminus R(t-1)) \times \{t\} \\ \mathit{Dstream}(R) &= \bigcup_{t \in T} (R(t-1) \setminus R(t)) \times \{t\} \\ \mathit{Rstream}(R) &= \bigcup_{t \in T} R(t) \times \{t\} \end{aligned}$$

There are two observations regarding the CQL window semantics which, amongst others, have consequences for realizing a transformation from STARQL to CQL (see below). First of all, the that the original time points of the stream elements get lost by applying the window operator. All tuples in the window get the times of the endpoint of the window. A second observation is that the CQL semantics, taken literally, do not work for non-discrete time flows. Take, e.g., the continuos time flow $(T, \leq) = (\mathbb{R}, \leq)$ and consider the input stream $S = \{i\langle i \rangle \mid i \in \mathbb{N}\}$ with unary tuples where the instantiations of the argument are their timestamps. Then $\mathit{RStream}(S[\mathit{RANGE}\ 1\ \mathit{SLIDE}\ 1])$ is not a "stream", as it has cardinality of $\mathbb{R}$.

So is the CQL semantics flawed? Actually, the "solution" is hidden int the stream engine layer, where one assumes a heartbeat with smallest possible time granularity. So the time flow is again discrete but may be annotated with real-time values. In STARQL (and in ADP) this granularity parameter is pushed to the query language level, so that the user can specify the granularity (see below).

4.2 Rewriting and Unfolding

The general procedure for rewriting and unfolding STARQL queries into queries over the backend source is illustrated below.

```
Input: STARQL Query SQ, Mappings M      Output: CQL query OQ
SQ1 = Rewrite(SQ, TBox(SQ))
SQ2 = SRNF(SQ1)
OQ = Unfold(SQ2, M)
```

Rewriting is done locally w.r.t. every embedded UCQ, using the TBox of the STARQL query `SQ`. This is possible as we assume (similar to [4]) that the TBox language does not contain temporal operators. In particular, this means that the TBox cannot express inter-temporal constraints. For example, assume that the TBox contains the role axiom $\mathit{tempVal} \sqsubseteq \mathit{val}$ stating that the role of having a temperature value is a sub-role of having a value. Assume further that `SQ` contains a HAVING clause that starts with `EXISTS i GRAPH i {?s :val ?x}`.... Then SQ would be rewritten into SQ1 with a HAVING clause starting with `EXISTS i GRAPH i ({?s :val ?x} UNION {?s :tempVal ?x})`. Here, the embedded CQ `{?s :val ?x}` is perfectly rewritten with the classical perfect rewriting algorithm.

The result `SQ1` is transformed to a query `SQ2` in (range-restricted) SRNF, which can be unfolded to an SQL like streaming language query `OQ`. In the following we illustrate the unfolding process into CQL queries.

Following the classical OBDA approach we assume that the streams to which STARQL refers are produced by mappings. In our example, let be given a CQL stream of measurements *Msmt* with schema `Msmt(MID, MtimeStamp, SID, Mval)`. A mapping takes a CQL query over this stream and produces a stream of assertions of the form $val(x, y)\langle t \rangle$.

$val(x, y)\langle z \rangle \longleftarrow$ `SELECT Rstream(f(SID) as x, Mval as y, MtimeStamp as z) FROM Msmt[NOW]`

The main challenge of an unfolding strategy for STARQL is the new data structure of an ABox sequence, which is not representable in CQL. Hence, we assume that the STARQL queries use the standard sequencing only, so that from every state i in the sequence associated with t_{NOW} one can reconstruct the timestamps of the tuples occurring in the ABox $\mathcal{A}_i$. A second assumption is that all tuples in the input stream contain an attribute timestamp the value of which is the same as the value of the timestamp. Such a method is also discussed in [2], in order to do computations directly on the timestamps. More concretely, we assume a default stream-to-stream operator implemented into the CQL answering system and applied directly after every window-to-stream operator. It transforms $d\langle t\rangle$ to $(d,t)\langle t\rangle$, thereby extending the schema of the tuples in the input stream by a time attribute for the output schema. If d already contains a time attribute, then it is overwritten by the new values. In the definition of the mapping for $val(x,y)\langle t\rangle$, we applied this assumption, where we refer to the time attribute `MtimeStamp` of the input stream `Msmt`. Note that this assumption mitigates the weakness of time-ignoring bag semantics in the window operators of CQL.

The following listing shows the unfolded CQL pendant of the STARQL query. The outer WHERE clause is the pendant (in SRNF form) of the monotonicity formula expressed in the HAVING clause.

```
CREATE VIEW windowRelation as
SELECT *  FROM Msmt[RANGE 2s Slide 1s];
SELECT Rstream(sensor, timestamp)
FROM windowRel, sensorRel
WHERE sensorRel.type = ''BurnerTipTempSens'' AND
NOT EXISTS (
 SELECT * FROM
 (SELECT timestamp as i, value as x FROM windowRelation),
 (SELECT timestamp as j, value as y FROM windowRelation)
 WHERE   i < j AND x > y );
```

Chapter 5

Temporal Reasoning by Mapping STARQL to SQL

For reactive diagnosis as investigated in Optique with the Siemens power plant use case, specific patterns, aka events, are to be found in previously recorded data. Data represents measurement values of sensors on the one hand, and turbines with assembly groups, mounted sensors and their properties on the other. Reactive diagnosis deals with analyzing data in order to understand reasons for engine shutdown. Not in all cases predefined queries lead the engineers to the right conclusion. In some cases new queries have to be formulated.

If we consider again the information need discussed in Chapter 2, it becomes clear that in the Siemens use case, patterns to be detected are defined w.r.t. certain time windows in which relevant event take place (e.g., monotonically increasing in a window of 10 minutes, say). The goal is to find window instances and representations for describing the events based on real-world data. Thus, in this context, temporal queries are formulated using time windows in the same spirit as time windows are used for continuous queries in stream processing systems as discussed in the previous chapter. Metrical aspects indeed play a prominent role in these use cases. Hence, due to the requirements of the use cases, Optique focuses on developing a window-based temporal query language for formulating "historical queries", with the idea that windows slide over the data in a virtual manner. Indeed, in principle, all windows can be processed in parallel if there are no dependencies of a window on results obtained for previous window, of course.

Considering the learning effort of an engineer, we argue that "historical" window-based temporal queries should be easy to formulate, given that the engineer is familiar with continuous queries for online stream-based data processing. Indeed, the same queries can be registered as a continuous query or a historical query. For the latter, it should obviously possible to specify a period of interest, i.e., a starting point and an end point for finding answers, e.g., as discussed above, to be used for reactive diagnosis.

5.1 Transforming Historical STARQL Queries into SQL

While in Chapter 4 continuous STARQL queries are translated to CQL, here we translate historical queries to SQL in order to provide a reference to backends well-known in the literature. In Chapter 6 we provide translations to the ADP engine developed in Optique as part of the Optique platform, and we deal with translations for continuous as well as for historical queries being mapped to ADP.

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

Figure 5.1: Example STARQL query for monotonic increase

The following example for a historical query finds sensors whose readings increase monotonically within a specified window time. The query is very similar to the continuous query explained in Chapter 2.

The example demonstrates that in STARQL window-based historical queries are built using the same principles as continuous queries. A few changes should be discussed, however. For historical queries, for each stream a start timestamp and an end timestamp can be specified in order to denote some range of interest in the past[1]. Compared to the continuous version of the query in Chapter 2, we habe changed the start time to an absolute value. We assume that a query is registered to the system after the stream s_msmt is declared (see below).

In Figure 5.2 it is shown how historical STARQL queries are processed internally. The prototype is

[1]Despite the fact that this could in principle also be specified in the mapping declarations, we prefer not to have mappings for particular queries and provide additional stream parameters.

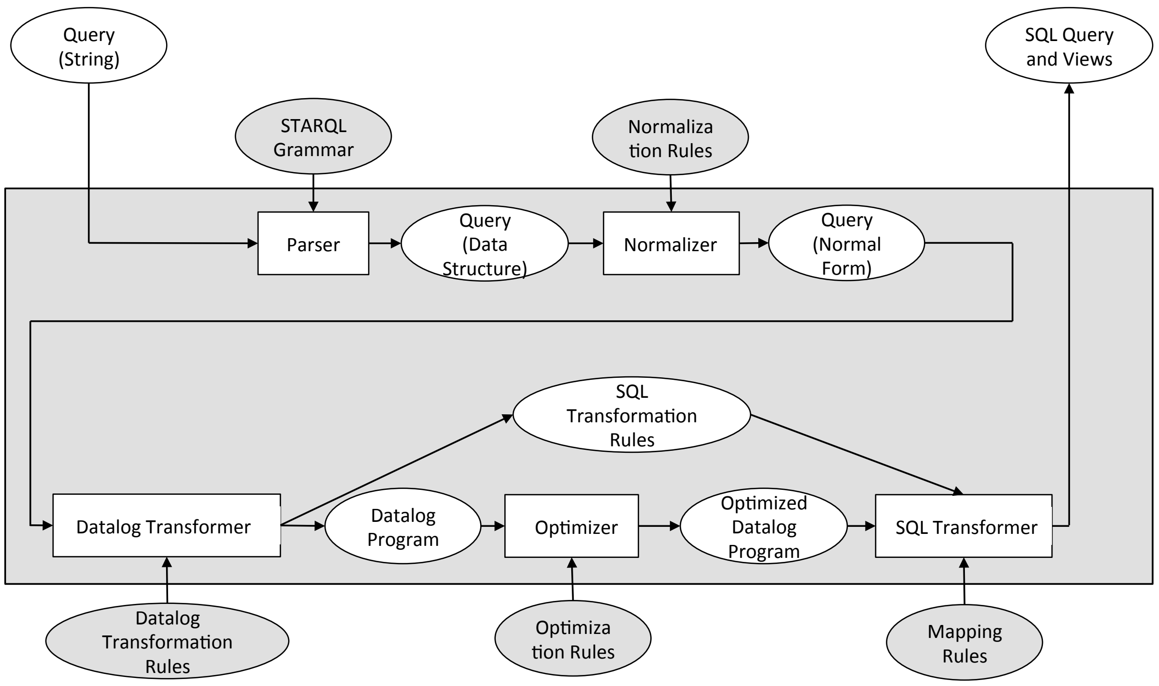

Figure 5.2: Processing pipeline for transforming a STARQL query into an SQL query.

implemented in Prolog, and the rules sets used as inputs in Figure 5.2 are DCG rules or Prolog rules.

A query is parsed in order to produce parse tree data structures, which then are normalized. Normalization converts FORALL expressions in the HAVING clause into NOT EXISTS expressions by pushing negation inside in the obvious way (starting from NOT NOT FORALL). Our example query is converted into the following expression.

```
PREFIX : <http://www.optique-project.eu/siemens>
CREATE STREAM s_out AS
CONSTRUCT GRAPH NOW { ?s rdf:type :MonInc }
FROM STREAM s_msmt [NOW-"2S"^^xsd:duration, NOW]->"1S"^^xsd:duration
        WITH START = "2005-01-01T01:00:00CET"^^xsd:dateTime,
             END = "2005-01-01T02:00:00CET"^^xsd:dateTime
     STATIC ABOX <http://www.optique-project.eu/siemens/ABoxstatic>,
     TBOX <http://www.optique-project.eu/siemens/TBox>
USING PULSE WITH START = "2005-01-01T01:00:00CET"^^xsd:dateTime,
                 FREQUENCY = "1S"^^xsd:duration
WHERE { ?s rdf:type :TempSens }
SEQUENCE BY STDSEQ AS seq
HAVING NOT EXISTS ?i < ?j IN seq :
             EXISTS ?x, ?y :
               ( GRAPH ?i { ?s val ?x } AND
                 GRAPH ?j { ?s  val ?y } AND
                 ?x > ?y ) ;
```

The normalized query is then translated into a datalog program (see Figure 5.2) with fresh predicate names being generated automatically. For every language element found in the parse tree (e.g., `NOT EXISTS...`), we generate a new datalog rule. In the body of the first (outermost) rule, datalog code for the WHERE clause is inserted (q0).

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

The WHERE expression is rewritten and unfolded due to the axioms in the TBox given in the query and the mapping rules, respectively. How the corresponding view is computed is explained in the previous chapter. Here we assume that sensors are created by a view defined below.

The following datalog code is generated for the HAVING clause of the query above (atom using built-in predicates are moved to the end of the body).

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

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

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

is added to the datalog program. From the mapping specifications for the stream `s_msmt`, the relations `val` and `sensor` are also defined using datalog clauses.

```
val( WID, Sensor, Value, Index ) :-
   window( Index, Timestamp, WID ),
   measurement( Sensor, Value, Timestamp ).

sensor( WID, Sensor ) :- val( WID, Sensor, _Value, _Index ).

seq( WID, Index ) :- window( Index, _Timestamp, WID ).
```

The table `measurement` contains raw data. The table *window* is generated according to the window specification in the query. This will be explained in the next section.

During clause generation, SQL transformation rules are generated (see Figure 5.2). Transformation rules represent the name of the SQL relation, the number of arguments as well as the type of each parameter. SQL transformation are statically specified also for the relations val, window, and seq.

The datalog program generated by the module Datalog Transformer (see Figure 5.2) is then optimized. Optimization eliminates wrapper clauses (see the clause q3 in our concrete example). In addition, body atoms are removed if bindings are already generated by other atoms. In our concrete example, it is the case that both *seq* atoms can be eliminated in q2. This has the consequence that also the clause for *seq* is eliminated such that the datalog program looks as follows:

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

The datalog program is non-recursive and safe, and therefore can easily be translated to SQL. The column names for relations are given by the SQL Transformation Rules generated by the Datalog Transformer (see above) as well as by explicit mapping rules given as input to the processing pipeline (see Figure 5.2). We proceed from bottom to top in the above datalog program in order to produce corresponding views.

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

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

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

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

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

The translation to SQL relies on definitions for tables `window` and `window_range`. In the next section we describe how both tables are computed.

5.2 Generating a Window Table With PostgreSQL

The tables `window` and `window_range` are based on the stream specification(s) in the query. Given start and end points for accessing temporal data as well as window size and window slide specifications, the goal is to compute a representation for all possible windows (with specific time points for the window range) together with all indexes for states that are built for the window by the sequencing method specified in the query. For the standard sequencing method the generation of the `window` and `window_range` relations uses specific features of PostgreSQL such as the `generate_series` function, and is shown below.

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

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

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

The expressions in `$s` are parameter for which concrete values are inserted. `$slide$`, `$window_size$`, `$start$`, and `$end$` are taken from the stream specification of the historical query. If `$start$`, and `$end$` all data and the minimum and maximum time point is computed automatically, as shown in the following query:

```
CREATE TABLE window_range AS
SELECT row_number() OVER (ORDER BY x.timestamp) - 1 AS wid,
       x.timestamp as timestamp
FROM ( SELECT generate_series(MIN(mp.timestamp),
                              MAX(mp.timestamp),
                              $slide$) AS timestamp
       FROM measurement mp
     ) x ;
```

5.3 Summary of Evaluation Using PostgreSQL

The table `window` is generated in less than 10 minutes for all sensors and measurement data in the database SiemensAnonymous2. The table `s_out` is computed for this database in a few minutes.

Separating the computation of the `window` and `window_range` table from the computation of s_out has a big advantage. Different WHERE and HAVING clauses can be specified, which, in turn, lead to different `s_out` queries, which are evaluated fast because the `window` and `window_range` table need not be recomputed. In addition, further optimizations are possible in case the `window` and `window_range` table can be computed by a selection due to a containment relation. For multiple queries made available to the system, `window` and `window_range` tables can be shared. This will be investigated as part of Task T5.4 in the next year.

Please note that PostgreSQL per se is an RDBMS and not a data stream management system (DSMS). The window table generation above is a one-step generation; but it could be made dynamical in order to get a real relational DSMS on top of PostgreSQL. The stream-enhanced ADP language already offers the operators needed for a DSMS.

Chapter 6

Stream Reasoning and Temporal Reasoning by Mapping STARQL to ADP

6.1 ADP

As explained above, we had to make some restrictions on the STARQL language and assume additional features for CQL to make the STARQL-CQL transformation possible. ADP has some nice features that ease the transformation.

ADP [17] is a system for large scale elastic data processing on the cloud. It follows the relational data model and it offers a high level language which is based on SQL92, extended with a wide variety of UDFs, offering a high-level and flexible syntax for expressing data pipelines. The user can easily extend the query language of ADP by writing their own UDFs in Python, and these UDFs are eventually executed efficiently inside the system. Into this direction, ADP, to which we will refer by ADP_{Stream} in the following, was extended with stream functionalities, offering a very flexible means to unfold STARQL queries as it provides window indexing operators with which sequences of ABoxes can be simulated. ADP adopts state of the art techniques for the efficient execution of complex dataflows in a distributive environment. So it provides the high distributability capabilities that are needed for performant OBDA applications as the one developed in Optique. ADPs specific stream capabilities are extended within this project in order to deal also with the demands of the stream query language STARQL within the Optique project. So, regarding the point of expressivity, the query language of ADP_{Stream} offers more operators than other relational DSMS—and of course more than RDBMs such as PostgreSQL.

ADP already extends the SQL query language with the possibility to generate virtual tables without explicitly formulating the create expressions. This capability is also used in ADP_{Stream} in order to read out from csv-files (e.g., with measurement data) the data and represent them in a virtual table (keyword `file` in the listing.) In ADP, one can specify any source as input for the `file` constructor, not only a file as the constructor name may suggest. In particular, it is possible to specify a TCP socket—a feature that is relevant for applying ADP for real streaming scenarios.

Listing 6.1 demonstrates the stream capabilities of ADP_{Stream} that are of relevance for the unfolding of STARQL queries to ADP queries in synchronous settings. (ADP also has a non-temporal row based window which is useful for processing streams with out of order arrival of stream elements). A stream `SMsmtADP` is constructed from the .csv file using a timesliding window for which one can explicitly specify the time column (here 0). The time window parameter sets the size of the window (also called the range); the frequency parameter sets the sliding parameter (here one second) and the granularity parameter in combination with the equivalence parameter defines the granularity according to which tuples are gathered in the same window content. The value *floor* just hints to a strategy for labeling the wid with some timestamp. The window operator works here by attaching to the tuples of the input table an additional column with a window ID (wid). If one were to apply a window operator twice then this would result in the addition of another wid.

The ADP query in Listing 6.2 is the unfolded counterpart of the monotonicity STARQL query as produced

```
CREATE STREAM SMsmtADP AS
 SELECT sensor AS x, value AS y, timestamp AS z}
 FROM (ordered timeslidingwindow timecolumn:0  timewindow:0 frequency:1  granularity:1
     equivalence:floor select * from (file 'burner3.csv' header:t));
```

Listing 6.1: ADP Stream Generated From a File

```
1  CREATE stream s_out_1 AS select * from
2  (ordered timeslidingwindow timecolumn:0 timewindow:2 frequency:1 granularity:1 equivalence
       :floor SELECT * FROM (file 'burner3.csv' header:t));
3
4  CREATE STREAM s_out_1_having AS
5  SELECT wid AS wid0, max(timestamp) AS now
6  FROM s_out_1
7
8  WHERE NOT EXISTS(
9
10 SELECT * FROM
11 (SELECT value as x, timestamp as i, wid as wid1 FROM s_out_1) as triple1
12
13 NATURAL JOIN
14
15 (SELECT value as y, timestamp as j, wid as wid2 FROM s_out_1) as triple2
16 WHERE wid0 = wid1 AND wid1 = wid2 AND i < j AND x > y)
17 AND (s_out_1."sensor" = 'TC255') AND s_out_1."value" IS NOT NULL
18 GROUP BY wid;
19
20 SELECT ':s0' as Subject, 'rdf:type' as Predicate, ':RecentMonInc' as Object, now as ts
21 FROM s_out_1_having;
```

Listing 6.2: Monotonicity STARQL query unfolded into ADP

by the STARQL engine prototype. In lines 1, 2 a stream view is built which is in essence the result of the STARQL window operator to the stream produced by the mapping. In line 4 a stream of tuples (with window IDs) is generated that mimics the stream of ABox sequences. As the window IDs are not labelled with timestamps—but these are needed in the STARQL query—the `max(timestamp)` expression in line refers to maximal timestamp in the window content. Line 8 to 18 just realize the SRNF form of the HAVING clause (see section on HAVING clause). The conjunction of the atoms are handled by joins. Lines 20,21 output the stream in the format as prescribed by the STARQL query.

6.2 A Stream-Temporal Implementation for Optique

In Chapter 5 we described a general tranformation strategy for transforming historical queries to SQL. In this chapter we will show how we transform continous STARQL queries to the ADP system used in Optique as part of the Optique platform. Later, we will evaluate the performance of our example query on the ADP-system.

As an example for a continous query we consider again the query from Figure 5.1. The query finds sensors whose values increase monotonically within a specified time window. In this case we want to translate a continous query, therefore we have to change the query and drop the information on start and end time for evaluating all available data. The modified query is shown in Figure 6.1.

The Optique STARQL to ADP-SQL translator is implemented in Java and makes use of specific features of the ADP-system; e.g., the translator uses several integrated Python operators such as the integrated sliding window operator. ADP-SQL describes the internal query language of the ADP which actually is a

```
PREFIX : <http://www.optique-project.eu/siemens>
CREATE STREAM s_out AS
CONSTRUCT GRAPH NOW { ?s rdf:type :MonInc }
FROM STREAM s_msmt [NOW-"2S"^^xsd:duration, NOW]->"1S"^^xsd:duration
     STATIC ABOX <http://www.optique-project.eu/siemens/ABoxstatic>,
     TBOX <http://www.optique-project.eu/siemens/TBox>
USING PULSE WITH FREQUENCY = "1S"^^xsd:duration
WHERE { ?s rdf:type :TempSens }
SEQUENCE BY STDSEQ AS seq
HAVING FORALL ?i < ?j IN seq :
          FORALL ?x, ?y :
             IF (GRAPH ?i { ?s val ?x }  AND GRAPH ?j { ?s  val ?y })
                THEN ?x <= ?y ;
```

Figure 6.1: Example continous STARQL query for monotonic increase

combination of SQLite with custom Python operators.

Figure 6.2 shows the internal translation process for queries into ADP-SQL. The query translation is analog to the SQL translation from chapter 5.1. The STARQL query is parsed into an object data structure with an ANTLR-Parser and specific grammar.

In the next step the HAVING clause is normalized, which is identical to the process in the standard SQL translation from chapter 5.1, where we replace the FOR ALL structure by a NOT EXISTS NOT construct.

For the further processing we handle different parts of the query differently and build a pipeline of streams, which together represent one single STARQL query.

The first stream that has to be translated into ADP-SQL consists of two parts. It reads the data and splits it into windows. The ADP version of 2013 that is used here loads data directly by using a file operator implemented in Python. The window generation—which was done in the SQL implementation using a specific SQL query—is realized here also by an additional Python operator called *timeslidingwindow*. It uses parameter for frequency, granularity, slide and range of each window. The generated input stream is shown in Listing 6.2 line 1 and 2.

We have to consider that the input data can be built of more than one stream. The idea that is used in the processing is that for each input stream windows are generated separately and the data is joined in the next step of the pipeline by an internal ADP process. For further information on data streams please consider Optique Deliverable 7.1 [10] and D7.2.

The next stream part are aggregator streams. Tuples can only grouped once per subquery in SQL, therefore we need to implement one additional stream for each aggregation or grouping. Thus, aggregators are also are handled separately in the implementation, however they are not part of the example.

The generation of the output is also done in a separate process. As we are generating streams of (timestamped) RDF triples, we are using a specific convention for the triple output format of 4 columns (Timestamp, Subject, Predicate, Object). It allows us to use the output of any STARQL query as input for another STARQL query. The output stream is shown in Listing 6.2 line 20 and 21.

The most important partial stream is translated from the having clause. After the normalization process a treewalker walks through the object tree of the having clause and translates every FORALL with time indices into joined tables with columns for each variable. All logical constraints from STARQL are now handled in an additional WHERE clause.

For the mapping of SQL tables and columns to RDF triples, respectively, the unfolding in the reverse direction, we use the Ontop system as a third party library. Each triple of the STARQL query is translated into table and column specifications separately. Necessary constraints of the unfolded WHERE clause are also imported from the Ontop unfolding. Additionally ontology and mapping files have to be provided for Ontop. This process differs from the corresponding one for the SQL implementation of Chapter 5.1 where an additional conversion of datalog rules is implemented.

The translated example of the having clause is shown in Listing 6.2 line 4 to 18.

6.3 Evaluation

In this section we will evaluate the runtime of two STARQL queries. A simple one with a temperature threshold in Figure 5.1 and the already mentioned recent monotonically increase query from Figure 6.1 in Section 5.1 with different sizes of data. Therefore, we use two different SQL translations and platforms each. First we measure the time for query answers on a classical PostgreSQL server with the translation strategy from Chapter 5.1. Second, we evaluate the same queries on different sizes of data on a ADP system installed on a virtual machine. The tests have been made on a system with a i7 2.8 ghz cpu and 16 GB of ram.

Please note that the tests regarding ADP have been conducted with its 2013 version in a virtual machine run on the PC system with the specifications above. At the time of writing this deliverable, the 2014 version of the ADP system could not be robustly setup in our testing environment. Evaluations of the 2014 version in comparison with PostgreSQL are now being conducted within the Optique platform on the same (server) conditions. Results of this evaluation are going to be described in Deliverable D1.3.

The first test query in Listing 6.3 shows a simple threshold check and returns a warning if the temperature for too high for a period of time. The corresponding SQL- and ADP-SQL queries are shown in Listing 6.4 and 6.5.

The dataset we use are simple sensor data provided by Siemens to the Optique project. It consists of timestamp, sensor, and value information. For more about the dataset compare Optique Deliverable 8.1 [8]. We use two different sizes of datasets here. A public set with 82080 rows over 3 days and a larger anonymized set with 420000 rows of data over 5 years.

The results for all 8 testcases can be seen in Figure 6.6.

Although the queries on the public data can be run in acceptable time on both systems, we see a gap between the ADP system and PostgreSQL. Query 1 is about 10 times and query 2 about 100 times faster

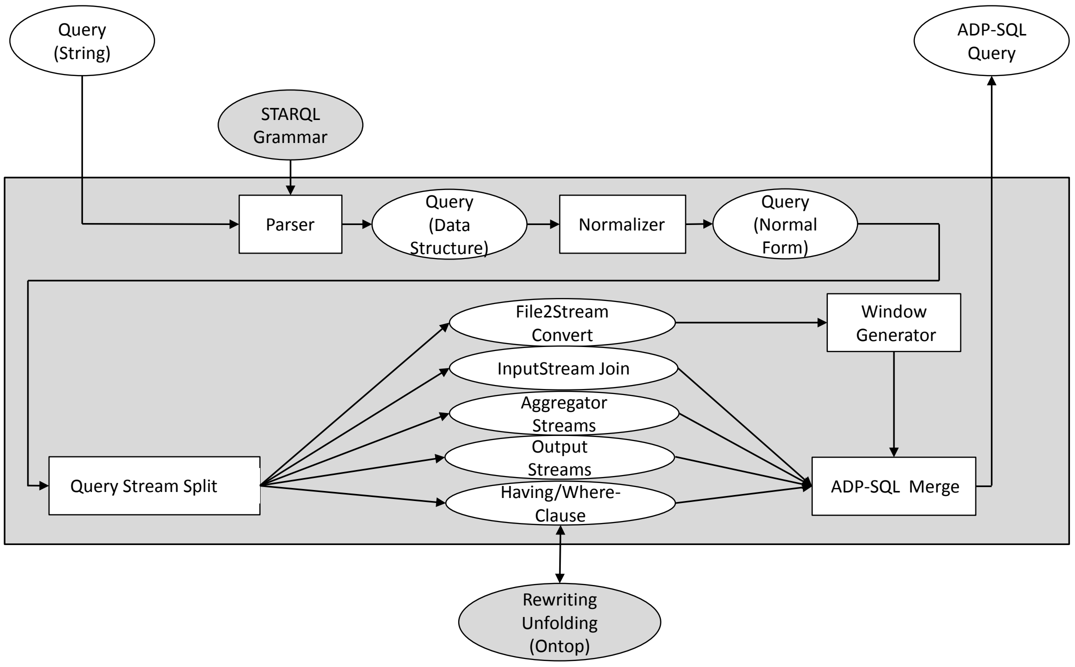

Figure 6.2: Processing pipeline for transforming a STARQL query into an ADP-SQL query.

```
CREATE STREAM S_out AS
SELECT { :TC255 rdf:type :tooHigh }< NOW>
FROM burner_3 0 seconds <- [ NOW - 180 seconds, NOW ]->1 seconds
SEQUENCE BY StdSeq AS SEQ1
HAVING FORALL i in stateSequence (
                FORALL ?x (
IF  { :TC255 :hasVal ?x }< i> THEN ?x > 90));
```

Figure 6.3: STARQL Testquery1

```
CREATE OR REPLACE VIEW q3 AS
SELECT rel1.WID , rel1.SID AS S
FROM HASVAL rel1
WHERE rel1.VALUE <= 90;

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

Figure 6.4: HAVING clause of Testquery1 (SQL)

```
CREATE STREAM S_out_having AS
SELECT burner.wid as wid0, max(burner.timestamp) AS now
FROM burner
WHERE Not Exists(

SELECT * FROM
(SELECT burner."value" as _x, abox as i, wid as wid1 FROM burner WHERE burner."value" IS
   NOT NULL) as triple1
WHERE wid0 = wid1 AND _x <= 90)

AND burner."value" IS NOT NULL
GROUP BY wid0;
```

Figure 6.5: HAVING clause of Testquery1 (ADP-SQL)

Dataset	Testquery	Testsystem	Time
Public	1	SQL	0.361 sec
Public	1	ADP-SQL	3 min 19 sec
Public	2	SQL	30 sec
Public	2	ADP-SQL	53 min
Anonym	1	SQL	3 sec
Anonym	1	ADP-SQL	1m53s (1 day), 7m22s (2 days), 17m9s (3 days)
Anonym	2	SQL	25 sec
Anonym	2	ADP-SQL	36 min 40 sec (1 month of data)

Figure 6.6: Results for tests 1 and 2 (ADP-SQL)

on the PostgreSQL system.

When it comes to the larger anonymized data set the gap increases. Both queries on the ADP system had to be aborted after 24 hours and the dataset to be shrunk to a max of 1 month of data. Even though, the query still takes a lot of time to be computed.

As a conclusion we have two results. The PostgreSQL system shows that it is possible to implement STARQL queries fast and efficiently, on the other hand the current system and implementation does not get close to these performance results. Which makes several optimization strategies necessary for the next years.

Please note that these experimental results regarding ADP are made w.r.t. to its 2013 version. Experiments conducted by the partners from Athens (see D7.2) with the 2014 version of ADP show by far better results.

Chapter 7

Conclusion

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

In this deliverable we argued that the (new) STARQL query language extended with a safety mechanism provides an adequate solution for streamified and temporalized OBDA scenarios as that of Siemens. Indeed, first of all, STARQL offers a semantics that allows a unified treatment of querying temporal and streaming data. Second, the Siemens use case requires a language providing logical constructors such as negation, disjunction, and quantifiers in combination with infinite domains for numerical values in order to reason over the (ABox sequences in the) window. These constructors would immediately lead to infinite sets of answers, if their use were not restricted by safety conditions. So, we provided a new safety mechanism for STARQL guaranteeing domain independence and hence also finite sets of answers.

Transformations of STARQL queries in the temporal scenarios (reactive diagnosis) and the continuous scenarios (proactive diagnosis) are possible, as argued for with proofs and implementations of the transformations from STARQL to ADP and from STARQL to PostgreSQL, respectively. Regarding the feasibility—at least w.r.t. the queries of the SIEMENS query catalogue, the standalone STARQL prototype shows acceptable query execution times, if run with the optimization mentioned here. Regarding the integrated prototype with the old ADP 2013 version, there are still some optimization needed w.r.t. to temporal processing. Preliminary experiments of the 2014 version of APD are more promising. But even here, many optimization possibilities, in particular w.r.t. the parallelization aspects within ADP still have to be investigated and implemented.

Bibliography

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15:121–142, 2006.

[3] Alessandro Artale, Roman Kontchakov, Frank Wolter, and Michael 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.

[4] Stefan Borgwardt, Marcel Lippmann, and Veronika Thost. Temporal query answering in the description logic DL-Lite. In Pascal Fontaine, Christophe Ringeissen, and Renate A. Schmidt, editors, *Frontiers of Combining Systems*, volume 8152 of *LNCS*, pages 165–180. Springer, 2013.

[5] Jean-Paul Calbimonte, Hoyoung Jeung, Oscar Corcho, and Karl Aberer. Enabling query technologies for the semantic sensor web. *Int. J. Semant. Web Inf. Syst.*, 8(1):43–63, January 2012.

[6] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, Antonella Poggi, Mariano Rodríguez-Muro, and Riccardo Rosati. Ontologies and databases: The DL-Lite approach. In Sergio Tessaris and Enrico Franconi, editors, *Semantic Technologies for Informations Systems (RW 2009)*, volume 5689 of *LNCS*, pages 255–356. Springer, 2009.

[7] Emanuele Della Valle, Stefano Ceri, Davide Barbieri, Daniele Braga, and Alessandro Campi. A first step towards stream reasoning. In John Domingue, Dieter Fensel, and Paolo Traverso, editors, *Future Internet – FIS 2008*, volume 5468 of *LNCS*, pages 72–81. Springer Berlin / Heidelberg, 2009.

[8] Thomas Hubauer, Steffen Lamparter, Christian Neuenstadt, Mikhail Roshchin, and Gerd Völksen. Deliverable D8.1: Interim siemens use case report 1. Deliverable, Optique, 2013.

[9] Evgeny Kharlamov, Nina Solomakhina, Özgür Lütfü Özcep, Dmitriy Zheleznyakov, Thomas Hubauer, Steffen Lamparter, Mikhail Roshchin, and Ahmet Soylu. How semantic technologies can enhance data access at siemens energy. In *Proceedings of the 13th International Semantic Web Conference (ISWC 2014)*, 2014.

[10] Herald Kllapi, Dimitris Bilidas, Yannis Ioannidis, and Manolis Koubarakis. Deliverable D7.1: Techniques for distributed query planning and execution: One-time queries. Deliverable, Optique, 2013.

[11] 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.

[12] Özgür L. Özçep, Ralf Möller, and Christian Neuenstadt. OBDA stream access combined with safe first-order temporal reasoning. Techn. report, Hamburg Univ. of Technology, 2014.

[13] Özgür L. Özçep, Ralf Möller, Christian Neuenstadt, Dmitriy Zheleznyakov, and Evgeny Kharlamov. Deliverable D5.1 – A semantics for temporal and stream-based query answering in an OBDA context. Deliverable FP7-318338, EU, October 2013.

[14] Özgür L. Özçep, Ralf Möller, and Christian 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.

[15] Özgür L. Özçep, Ralf Möller, and Christian Neuenstadt. A stream-temporal query language for ontology based data access. In Meghyn Bienvenu, Magdalena Ortiz, Riccardo Rosati, and Mantas Simkus, editors, *Proceedings of the 7th International Workshop on Description Logics (DL-2014)*, 2014.

[16] Danh Le Phuoc, Minh Dao-Tran, Josiane Xavier Parreira, and Manfred 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.

[17] Manolis M. Tsangaris, George Kakaletris, H. Kllapi, Giorgos Papanikos, Fragkiskos Pentaris, Paul Polydoras, E. Sitaridi, Vassilis Stoumpos, and Yannis E. Ioannidis. Dataflow processing and optimization on grid and cloud infrastructures. *IEEE Data Eng. Bull.*, 32(1):67–74, 2009.