Journal of Ambient Intelligence and Smart Environments 0 (0) 1
IOS Press

Querying Industrial Stream-Temporal Data: an Ontology-based Visual Approach [1]

Ahmet Soylu [a,*], Martin Giese [b], Rudolf Schlatte [b], Ernesto Jimenez-Ruiz [b], Evgeny Kharlamov [c], Özgür Özçep [d], Christian Neuenstadt [d] and Sebastian Brandt [e]

[a] *Department of Computer Science, Norwegian University of Science and Technology, Gjøvik, Norway*
E-mail: ahmet.soylu@ntnu.no
[b] *Department of Informatics, University of Oslo, Oslo, Norway*
E-mail: {martingi, rudi, ernestoj}@ifi.uio.no
[c] *Department of Computer Science, University of Oxford, Oxford, UK*
E-mail: evgeny.kharlamov@cs.ox.ac.uk
[d] *Institute of Information Systems, University of Lübeck, Lübeck, Germany*
E-mail: {oezcep, neuenstadt}@ifis.uni-luebeck.de
[e] *Corporate Technology, Research and Technology Center, Siemens AG, Munich, Germany*
E-mail: sebastian-philipp.brandt@siemens.com

Abstract. An increasing number of sensors are being deployed in business-critical environments, systems, and equipment; and stream a vast amount of data. The operational efficiency and effectiveness of business processes rely on domain experts' agility in interpreting data into actionable business information. A domain expert has extensive domain knowledge but not necessarily skills and knowledge on databases and formal query languages. Therefore, centralised approaches are often preferred. These require IT experts to translate the information needs of domain experts into extract-transform-load (ETL) processes in order to extract and integrate data and then let domain experts apply predefined analytics. Since such a workflow is too time intensive, heavy-weight and inflexible given the high volume and velocity of data, domain experts need to extract and analyse the data of interest directly. Ontologies, i.e., semantically rich conceptual domain models, present an intelligible solution by describing the domain of interest on a higher level of abstraction closer to the reality. Moreover, recent ontology-based data access (OBDA) technologies enable end users to formulate their information needs into queries using a set of terms defined in an ontology. Ontological queries could then be translated into SQL or some other database query languages, and executed over the data in its original place and format automatically. To this end, this article reports an ontology-based visual query system (VQS), namely OptiqueVQS, how it is extended for a stream-temporal query language called STARQL, a user experiment with the domain experts at Siemens AG, and STARQL's query answering performance over a proof of concept implementation for PostgreSQL.

Keywords: visual query formulation, ontology-based data access, temporal data, stream sensor data, data retrieval, usability

1. Introduction

The advances in pervasive computing and the emergence of low cost wireless and non-intrusive sensors open up new possibilities for industries such as oil and gas, power, mining, and agriculture [38,35,20,34]. For example, operators can recognise hazardous conditions by actively monitoring *stream sensor data* coming from plant equipment such as pumps, motors, and turbines, or analyse *historical sensor data* in the event of a problem for a proper diagnosis. The operational efficiency and effectiveness of business processes rely on domain experts' agility in interpreting data into ac-

[1] This work was funded by the EU FP7 Grant "Optique" (agreement 318338), and by the EPSRC projects MaSI[3], DBOnto, and ED[3].
[*] Corresponding author. E-mail: ahmet.soylu@ntnu.no

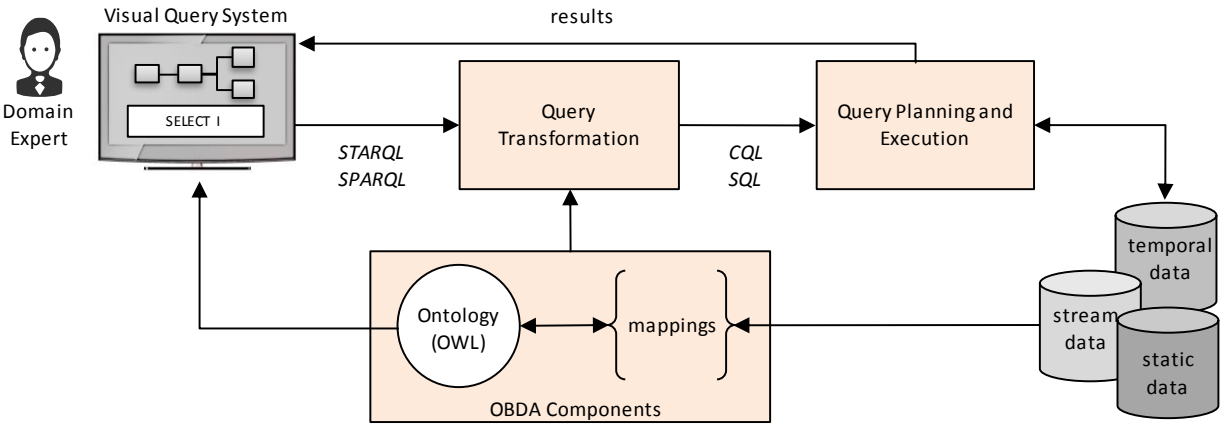

Fig. 1. OptiqueVQS in an OBDA scenario over relational databases.

tionable business information, so as to give *reactive* and *proactive* responses with respect to important data patterns appearing in data streams [51]. However, *domain experts*, who have extensive domain knowledge, may or may not have technical skills and knowledge on databases and formal textual query languages for *stream-temporal* data sources, such as CQL [2], C-SPARQL [3] and STARQL [30], to specify and extract data of interest [39]. Therefore, centralised approaches are often preferred. These require *IT experts* to translate the *information needs* of domain experts into *extract-transform-load* (ETL) [12] processes in order to extract and integrate data possibly from disperse data sources. Domain experts then apply predefined analytics over the delivered data. However, such a workflow is too time intensive, heavy-weight and inflexible given the high volume and velocity of data.

Turnaround time between an important event and a possible reaction could be reduced drastically, if domain experts could directly specify and isolate important data fragments rather than having IT experts in the middle. A simple example could be shutting down an overheated turbine; however, an event could also be of a more complex nature involving more than one sensory source and *static data*. *Visual query formulation* [8] is a viable approach as it aims to lower the knowledge and skill barriers to a minimum. In this context, *ontology-based visual query formulation* is gaining attention as ontologies come with certain benefits compared to visual query formulation over database schemas (cf. [47]). Firstly, ontologies, i.e., semantically rich conceptual domain models, present an intelligible solution by describing the domain of interest on a *higher level of abstraction* closer to the reality. Secondly, the *federation* and *reasoning* power of ontologies are very valuable (cf. [14]) for addressing scenarios where data is distributed, incomplete or conflicting. Finally, *ontology-based data access* (OBDA) approach extends the reach of ontology-based querying from triple stores to relational databases [32,48,22]. In OBDA, end users formulate their information needs into queries using a set of terms defined in an ontology. Ontological queries could then be translated into SQL or some other database query languages through a set of mappings linking the ontology and the underlying data sources, and executed over the data in its original place and format automatically.

Although a considerable amount of work exists on ontology-based visual query formulation for SPARQL, it is limited for *ontology-based visual stream-temporal querying* (cf. [47]). Therefore, OptiqueVQS [44,43, 45], an *ontology-based visual query system* (VQS), has been extended for stream-temporal querying upon the requirements provided by Siemens AG[1] within an OBDA project called Optique[2] [14,15]. Previously, OptiqueVQS was successfully evaluated with different type of user groups in non-stream and non-temporal scenarios with casual users [44], and with domain experts at Statoil ASA[3] and Siemens AG [43]. Although OptiqueVQS used in an OBDA scenario over relational databases in the context of Siemens and Optique project, it could be used directly over a triple store or any other OBDA framework. Figure 1 shows how OptiqueVQS is used in an OBDA scenario along

[1] http://www.siemens.com
[2] http://optique-project.eu
[3] http://www.statoil.com

with relevant OBDA components. Now, this article reports how OptiqueVQS has been extended to generate stream-temporal queries in STARQL [30], a user experiment with the domain experts at Siemens over stream sensor data, and STARQL's query answering performance over a proof of concept implementation for PostgreSQL[4]. The results suggest that OptiqueVQS is a viable tool for domain experts for querying stream-temporal data sources and STARQL's computational performance is promising.

In what follows, Section 2 introduces the Siemens case and OBDA. Section 3 presents STARQL and OptiqueVQS interface with stream-temporal querying. Section 4 presents a computational experiment with STARQL and a user experiment with Siemens' domain experts. Finally, Section 5 presents the related work and Section 6 concludes the article.

2. Background

Siemens presents a real and large-scale industrial use case, which drives the research presented in this article on ontology-based visual query formulation for stream-temporal data sources. The use case is also important to demonstrate that OBDA over legacy relational databases plays a pivotal role for the integral and wide-ranging proliferation ontologies and ontology-based approaches in the industry.

2.1. The Siemens use case

Siemens produces a wide range of complex appliances, such as gas and steam turbines, generators, and compressors, which are used in business-critical processes, including power generation, in energy sector. Therefore, in order to prevent high downtime costs, Siemens runs several service centres, each responsible for remote monitoring and diagnostics of such many thousands of appliances. Data is stored in several thousand databases with varied schemas and the size of the data is in the order of hundreds of terabytes, e.g., there is about 15 GB of data associated to a single turbine, and it currently grows with the average rate of 30 GB per day [20]. Service centres have two main categories of tasks:

(a) *reactive tasks*: engineers become active once a problem is reported by customers, and then query and analyse time-stamped sensor data distributed across multiple sources;
(b) and *predictive tasks*: data received from the appliances is actively monitored, and pre-defined patterns are detected in the incoming sensor and event data for early diagnosis.

Figure 2 presents the service process triggered after a malfunction of a unit (i.e., reactive scenario) [20]. In this case, a service ticket requesting assistance is created either manually or automatically by a diagnostic system. The ticket often has very limited information concerning the location and cause of the problem. Service engineers query databases containing sensor and event data (i.e., data acquisition) through manipulating 4.000 *predefined queries and query patterns*. In case these are not sufficient, an IT expert has to be involved to create a new query or query pattern. Standard diagrams are used to visualise sensor data and event messages are listed in excel spreadsheet with timestamps and other attributes. Data then is pre-processed manually using generic procedures in order to, for example, see whether the sensor data quality is appropriate or not. The engineer uses sophisticated diagnostic models and tools, such as principal component analysis or other statistical methods, to analyse and detect the given problem based on the pre-processed data. Finally, the process is terminated when an explanation for the problem is found. The process for predictive analysis is similar, but have to be applied online to streaming data with minimal user intervention.

For situations not initially anticipated, new queries are required, and an IT expert familiar with both the power plant system and the data sources in question (e.g., up to 2.000 sensors in a part of appliance and static data sources) has to be involved to formulate these queries. Thus, unforeseen situations may lead to significant delays of up to *several hours or even days* due to *miscommunication*, *high workload* of IT personnel, *complexity of query formulation*, and *long query execution times*. In average, up to 35 queries require modification every month, up to 10% of queries are changed throughout a year, and several new queries are developed monthly [20].

With few built-in features for manipulating time intervals, traditional database systems often offer insufficient support for querying time series data, and it is highly non-trivial to combine querying techniques with the statistics-based methods for trend analysis that are typically in use in such cases. By enabling engineers to formulate complex stream-temporal queries

[4] https://www.postgresql.org

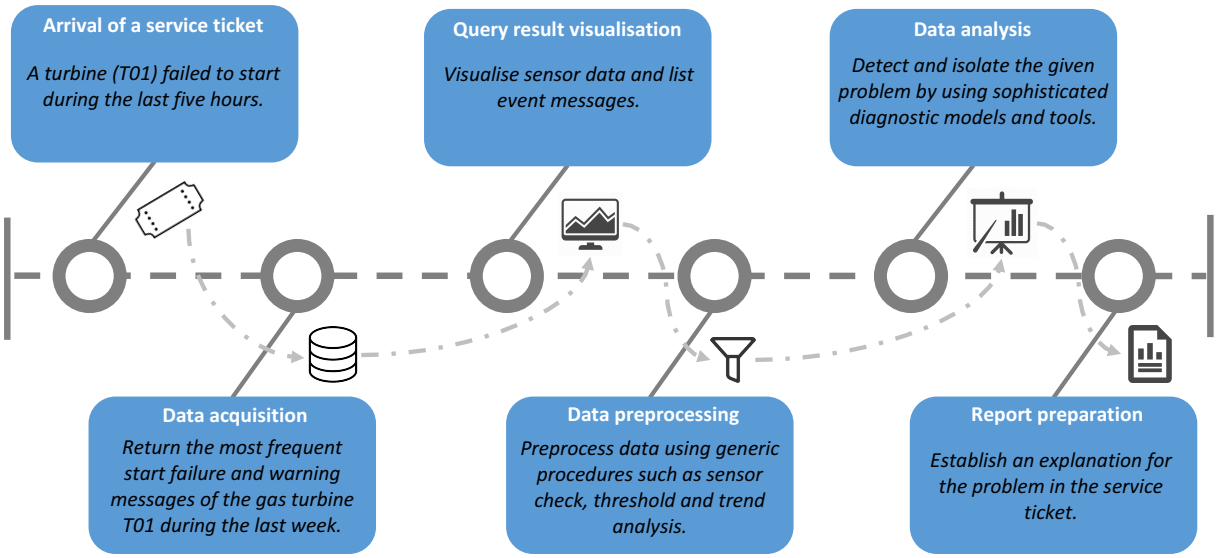

Fig. 2. Siemens service process for reactive tasks.

on their own with respect to an expressive domain vocabulary, IT experts will not be required anymore for adding new queries, and manual pre-processing steps can be avoided. This would lead to *(i) timely-decision making*, *(ii) augmented value creation* by redeploying freed-up time, and *(iii)* previously *unforeseen uses of data* through ad-hoc querying.

2.2. Ontology-based data access

A significant amount of world's enterprise data resides in relational databases rather than triple stores. Therefore, ontology-based visual query formulation would not be a pragmatic solution for industry without technologies for ontology-based data access over relational databases. OBDA technologies, such as Ontop [6], Mastro [10], and Ultrawrap [36], make it possible to *virtualise* RDF graphs from relational databases and enable *in-place access* to relational data over ontologies without migrating or duplicating any data. Moreover, thanks to OBDA, while using well-established query optimisation and evaluation support available for traditional database systems, one could also *(i)* integrate data from multiple databases with different schemas by relating each to a common ontology, and *(ii)* utilise implicit information in query the answering process by relating the whole set of implied information with logical reasoning.

OBDA over relational databases is based on two key mechanisms:

(a) *mappings* to describe the relationships between the terms in the ontology and their representations in the data sources (i.e., data and database schema).

(b) and *query transformation* to expand and translate the posed queries (e.g., in SPARQL) into the language of the underlying relational database system (e.g., to SQL).

Figure 3 presents a simplified OBDA scenario. The example involves an ontology, in which a turbine could be either gas or diesel turbine. A diesel turbine could have different types depending on the fuel type, such as biodiesel turbine. The corresponding relational database includes two tables, one for gas turbines and other for diesel turbines. The latter has fuel attribute to differentiate between different diesel turbine types. There are three mappings indicating how to construct RDF triples given the data that comes from an SQL query over the data source. A user submits a SPARQL query Q asking for all Turbines. At this point, several query transformations take place (cf. [5]). First, the query is *rewritten* into Q^I taking ontological constrains into account in order to retrieve both explicit and implicit answers. Second, by using mappings, Q^I is *unfolded* into Q^{II}, which is the query in the language of the underlying database system. Finally, the Q^{II} is optimised and transformed into Q^{III}. The exam-

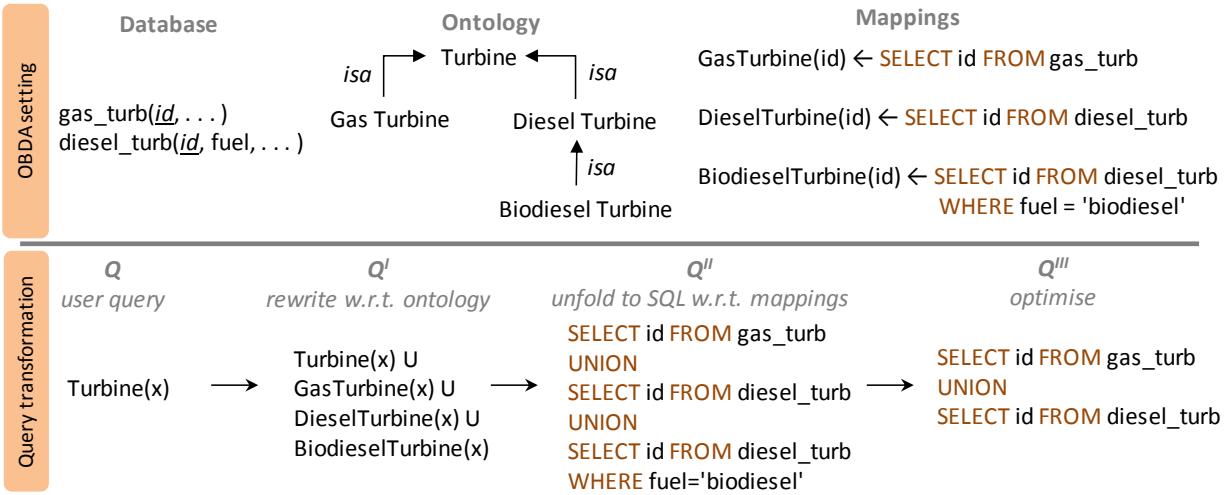

Fig. 3. An example OBDA scenario.

ple is described considering the Ontop[5] [6], which is the OBDA tool used in the Optique project.

In order to ensure that these transformation steps preserve the right semantics of query answering, one should ensure the right combination of ontology and mapping languages. In particular, it has been shown that the combination of the DL-Lite as the ontology language (e.g., OWL 2 QL[6]) and *global-as-view* (GAV) mappings guarantees that both rewriting and unfolding are semantically correct and efficient (cf. [5, 32,48]). Moreover, in OBDA, one assumes that the ontology alone is consistent and it is also consistent in a combination with the data computed by executing mappings, i.e., by populating the classes in the ontology with the data from databases.

A key benefit of the mapping and query transformation approach is avoiding the representation controversy [28] between ontologies and relational databases (i.e., *impedance mismatch*) due to their semantic and syntactic differences [40]. By separating transactional and domain perspectives, end users can formulate their information needs into queries in an intelligible manner using the terms coming from an ontology, and get their queries answered efficiently and effectively with OBDA and traditional mature database technologies. Another advantage is that there is no need to chase GBs of data for reasoning to generate all the implicit facts derived by the ontology, since this problem is handled by embedding the consequences of the ontology into the mappings (cf. [15]) – see Figure 3 for an example.

[5] http://ontop.inf.unibz.it
[6] https://www.w3.org/TR/owl2-profiles

3. Stream-temporal querying and OptiqueVQS

In an industrial context, OptiqueVQS is meant for domain experts who may or may not have technical skills and knowledge, for example on databases, query languages, programming languages, and semantic web technologies, and usually have *frequent*, *varied*, and *sophisticated* information needs.

Catarci et al. [8] in their review point out that query interfaces combining multiple presentation and integration paradigms have a better potential to address broader user and task types. Therefore, a *multi-paradigm* approach is in the core of OptiqueVQS' design and implementation. OptiqueVQS is not concerned with reflecting the underlying formality (i.e., query language and ontology) per se, as this would demand considerable technical background from end users. Full *expressivity* is also not a goal, as simpler interfaces covering the majority or the essential fragments of end-user information needs are more likely to succeed [7]. Expressivity is intentionally compromised for the sake of *usability* by targeting frequently needed query fragments with less perceived complexity for the user [**?**]. Therefore, OptiqueVQS primarily targets *tree-shaped conjunctive queries* and the essential fragments of STARQL.

STARQL has been selected as an underlying query language for querying temporal-stream data primarily

due to its support for OBDA; otherwise, OptiqueVQS could potentially generate queries in any other language. STARQL offers a query framework dealing with streams of timestamped RDF triples with respect to a set of mappings and an ontology. The development of STARQL was inspired by the Siemens use case requirements. STARQL allows expressing typical mathematical, statistical, and event pattern features needed in real-time monitoring scenarios (i.e., *expressivity*); comes with a formal syntax and semantics (i.e., *neat semantics*); takes streams of timestamped assertions as input and produces streams of timestamped assertions (i.e., *orthogonality*); allows selecting an ontology and streams over which the query will be evaluated (i.e., *scope locality*); allows storing and re-using often-used query patterns (i.e., *library functions*); and uses roughly same STARQL queries to query historic data or to query real-time streams (i.e., *common interface*) [30].

3.1. STARQL

STARQL [29**?** ,30] provides an expressive declarative interface to both historical and streaming data. In STARQL, querying historical and streaming data proceeds in an *analogous* way and in both cases the query may refer to *static data*, i.e., data that do not have a timestamp and hence are considered to hold at every time point. The answers coming from the static subquery are used for the stream processing in the remainder of the query. This separation between the static and dynamic aspects provides a useful abstraction which eases the query building process.

The relevant slices of the temporal data are specified with a *window*, a *sliding parameter* that determines the rate at which snapshots of the data are taken, and a *window width*. The window, both in the case of historical data and in the case of streaming data, is a moving window containing a reference to the developing time NOW. The difference is that in the case of historical data, the data is read from an ordinary database (such as a PostgreSQL database), whereas in the real-time case the data is coming from a real-time stream source. Moreover, in the historical case, it may make sense to specify windows with a right end-point bigger than the running time NOW (as the data is available) whereas in the stream case this is not possible.

The contents of the temporal data are grouped according to a *sequencing strategy* into a sequence of small graphs that represent different states. For each state *i*, (referenced by the keyword `GRAPH i`), one may ask whether some assertion holds in it. On top of the sequence, relevant patterns and aggregations are formulated in the HAVING clause, using a highly expressive template language. In Figure 4, an example STARQL query is given, which asks for a train with turbine named "Bearing Assembly", and queries for the journal bearing temperature reading in the generator. It uses a simple echo to display the results.

OptiqueVQS (and its underlying engine) follows the OBDA paradigm. Hence, it has to be configured with data sources, an ontology, and last but not least with mappings. The followings illustrate these components and how they are actually used w.r.t. the STARQL query in Figure 4. The focus is on the historical-data scenario, that is, it is assumed that the stream measurement referenced in the STARQL query is read from an ordinary relational table such as a PostgreSQL table.

The source data is assumed to be stored in relational tables. For example, historical measurement data are stored in a table *measurement* containing an ID, an attribute for the time of the measurement, the source sensor of the measurement, and the measured value. The structure of trains, turbines, and sensors are stored in various tables containing information regarding which component is attached to which component.

For example, there is a ternary relation for sensors including an identifier, the assembly at which they are attached and a human-readable name, and a similar relation exist for measurements:

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

These tables and schemata may be quite complex and should be hidden from the user of the VQS. The user should access data via an ontology, a conceptual model containing relevant concepts (such as *Sensor*, *Turbine*, *Train* etc.) and properties/roles (such as *hasValue*) interrelated via constraints. For example, one constraint of the ontology is invoked in Figure 4 and it states that all temperature sensors are sensors. Bridging the conceptual (ontology) level with the real-world data (relational databases) is handled by mappings that roughly say how the concepts and properties are populated by objects from the database.

A simple mapping for the static (i.e., non-temporal) concept sensor is given with the following rule:

?x ns1 : *type TemperatureSensor* ⟵

```
SELECT f(SID) as x FROM sensor
WHERE sensor.name ~'^TC*'
```

The right-hand side is an SQL query that selects all sensors identified by an *SID* and having a name that

```
PREFIX ns1 : <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ns2 : <http://www.siemens.com/ontology/gasturbine/>
CREATE PULSE pulseA WITH FREQUENCY = "PT1s"^^xsd:duration

CREATE STREAM S_out AS
SELECT { ?_val0 ?Train_c1 ?Turbine_c2 ?Generator_c3 ?BearingHouse3_c4
  ?JournalBearing_c5 ?TemperatureSensor_c6 }
FROM STREAM measurement
  [NOW - "PT10s"^^xsd:duration, NOW]->"PT1s"^^xsd:duration
USING PULSE pulseA
WHERE {
  ?Train_c1 ns1:type ns2:Train.
  ?Turbine_c2 ns1:type ns2:Turbine.
  ?Generator_c3 ns1:type ns2:Generator.
  ?BearingHouse3_c4 ns1:type ns2:BearingHouse3.
  ?JournalBearing_c5 ns1:type ns2:JournalBearing.
  ?TemperatureSensor_c6 ns1:type ns2:TemperatureSensor.
  ?Train_c1 ns2:hasTurbine ?Turbine_c2.
  ?Train_c1 ns2:hasGenerator ?Generator_c3.
  ?Generator_c3 ns2:hasBearingHouse3 ?BearingHouse3_c4.
  ?BearingHouse3_c4 ns2:hasJournalBearing ?JournalBearing_c5.
  ?JournalBearing_c5 ns2:isMonitoredBy ?TemperatureSensor_c6.
  ?Turbine_c2 ns2:hasName "Bearing Assembly"^^xsd:string.
}
SEQUENCE BY StdSeq AS seq
HAVING EXISTS i IN seq
  ( GRAPH i { ?TemperatureSensor_c6 ns2:hasValue ?_val0 } )
```

Fig. 4. An example diagnostic task in STARQL.

starts with "TC" (for temperature sensors). The attribute *SID* is used as a template to generate (virtually) all assertions saying that an object with identifier *SID* is an instance of the concept *TemperatureSensor*. The *SID* is wrapped into a functional term *f(SID)* in order to overcome the so-called impedance mismatch between the real-world data (such as the values of *SID*) and that of abstract objects (real temperature sensor, represented by *f(SID)*).

The mappings for temporal concepts and properties are slightly more complex. In STARQL, temporal concepts and roles (such as *hasValue*) are used in the HAVING clause in states. One first needs to specify how non-temporal data are mapped in a general mapping schema and then instantiate them with concrete window parameters.

For example, a classical mapping relates the temporal property *hasValue* to the table *measurement* for storing sensor IDs and measurement values as follows:

```
?x ns2:hasValue ?y ←
  SELECT measurement.sid as ?x,
         measurement.value as ?y
  FROM measurement
```

The concrete mapping using the window parameters then is the following:

```
GRAPH i { ?x hasVal ?y } ←
  SELECT sid as ?x , sval as ?y
  FROM Slice(measurement,i,r,sl,st)
```

The left-hand side contains a template on a state and the right-hand side extends the mapping for *hasValue* by applying the function *Slice* describing the relevant finite temporal slice of the table *measurement* from which the ontology level assertions are produced and using the window parameters such as range *r*, slide *sl*, the sequencing strategy *st* and the index *i* (see Neuenstadt et al. [29] for further details).

In classical OBDA, queries are answered by transforming the query according to the ontology and the mappings. This is done automatically by the engines underlying the VQS according to a correct and complete algorithm. The outcome of the transformation may become considerably larger than the more abstract STARQL query. This is illustrated by a simple transformation to PostgreSQL in the evaluation section. For

more information on STARQL itself, readers are referred to Özçep et al. [30].

3.2. OptiqueVQS

OptiqueVQS is a visual query system rather than a *visual query language* (VQL), that is, it is based on a *system of interactions* rather than a formal visual *syntax* and *notation* (cf. [13]). This allows OptiqueVQS to offer familiar and informal presentation and interaction styles, while constraining the user interaction so as to enforce the formulation of valid queries.

OptiqueVQS is designed as a *widget-based user-interface mashup* (UI mashup) [44,42]. *Widgets* are standalone, full-fledged, and re-usable applications, which are put together in a common graphical space in order to build a new interface augmenting the user experience [41]. Widgets communicate with each other by broadcasting event messages as a user interacts with them and widgets react automatically to the events depending on their signatures. Such an approach offers *flexibility*, *modularity*, and *adaptability* and enables us to combine multiple *representation paradigms*, such as *forms*, *diagrams* and *icons*, and *interaction paradigms*, such as *schema navigation*, *range selection*, and *matching* [8]. A multi-paradigm approach is important to support different user and task types as stated earlier [39,47].

Since the HAVING clauses of typical STARQL queries are comparatively complex, in OptiqueVQS stream query formulation is separated into two tasks: *(i)* selecting the exact data stream to query (which values from which sensors, etc.), and *(ii)* what to do with these values. The second is done by letting the user choose one from a set of templates. Technically, a *template* computes a HAVING clause, and a list of selected variables for output, based on a variable from the WHERE part, a *dynamic property* (i.e., whose extension is time dependent) of that variable, and possible parameters instantiated by the user (range intervals, etc.). Some available templates are “echo" for copying values to the result stream, “range" for checking that values are within a given parametrisable interval, and “gradient" for checking that the derivate does not exceed a given value.

OptiqueVQS with stream-temporal querying presents five widgets to a user:

(i) the first widget (W1), see the bottom left side of Figure 5, is a menu-based widget and allows the user to navigate through concepts of an ontology by selecting relationships between them;

(ii) the second widget (W2), see the top part of Figure 5, is a diagram-based widget presenting typed variables as nodes and object properties as arcs to give an overview of the query formulated so far;

(iii) the third widget (W3), see the bottom right side of Figure 5, is a form-based widget and presents the attributes of a selected concept for selection and projection operations;

(iv) the fourth widget (W4), see Figure 6, is a form-based widget and supports selection of parameters, such as slide (i.e., frequency at which the window content is updated/moves forward) and window width interval, for stream-temporal queries;

(v) the fifth widget is (W5), see Figure 7, is a tabular widget and allows selecting a template for each stream attribute, which is by default “echo” (this widget is normally used for displaying example results in SPARQL mode).

In a typical scenario[7] [46], W1 initially lists all domain concepts and the user first selects a starting concept (i.e., *kernel concept*) from W1. The selected concept appears on the diagram (i.e., W2) as a typed *variable node* and becomes the active node (i.e., *pivot*). W3 displays the attributes of selected variable node as text fields, range sliders, etc. W3 relies on *propagation of property restrictions* [**?**], that is, properties of sub/super concepts are also presented for a given concept. The user can put constraints on attributes and a constrained attribute appears over the corresponding node in the query diagram with letter “c”. The user can select attributes for output by using the “eye” button. An attribute selected for output appears on the corresponding variable node with a letter “o”. The user can further refine the type of variable node from W3, by selecting appropriate subclasses, which are treated as a special attribute (named “Type”). However, currently other attributes presented in W3 remains unchanged after refinement. Once there is a pivot node, each item in W1 represents a combination of a possible relationship – range concept pair pertaining to the pivot. Then, if the user selects an item from W1, it triggers a join between the pivot and the new variable node (of type range concept) over the specified relationship. The new variable node becomes the pivot. The user can change the pivot by clicking on any variable node in the query diagram (i.e., W2) and expand the query form there.

[7] Demo video: `https://youtu.be/TZTxujz5hCc`

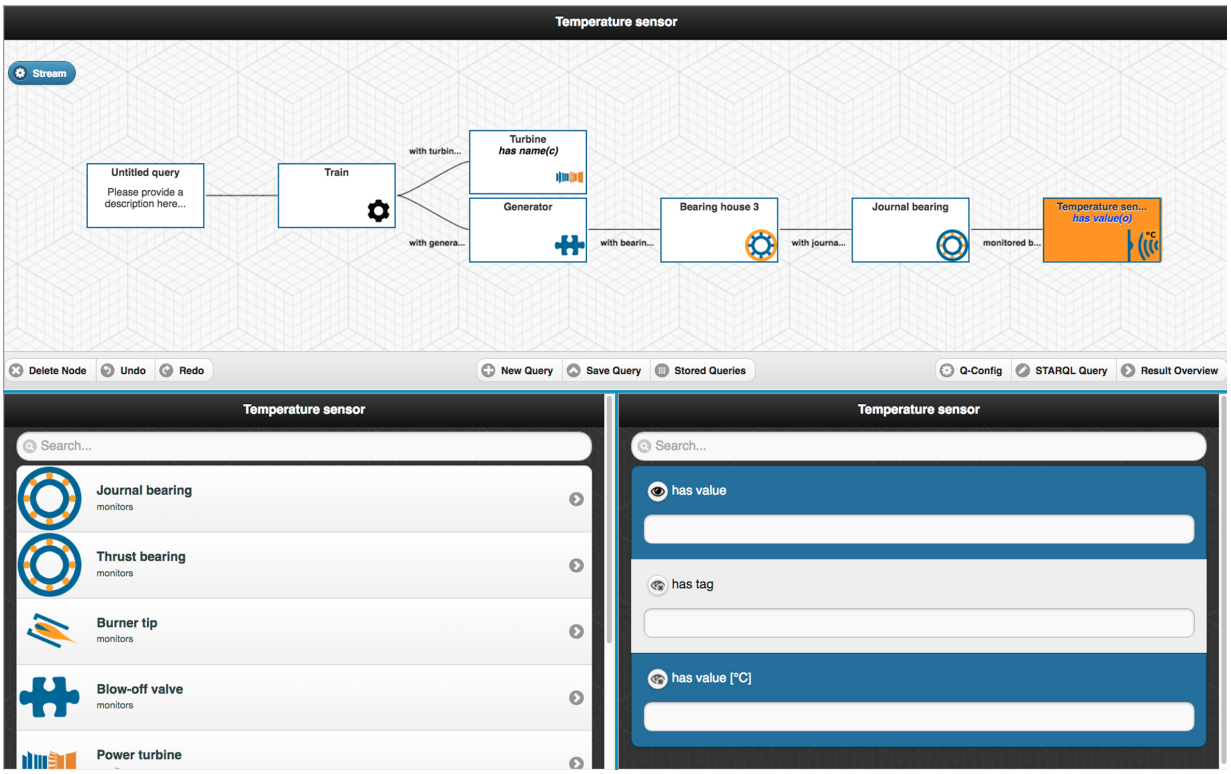

Fig. 5. OptiqueVQS with stream-temporal querying.

In a stream-temporal scenario, dynamic properties are presented in blue by W3. As soon as a dynamic property is selected, OptiqueVQS switches to STARQL mode. A stream button appears on top of the W2 and lets the user to activate W4 for parameter configuration. If the user clicks on the "Result Overview" button, W5 appears for selecting a template for each dynamic attribute. Finally, the user can *register the query* through W5 by clicking on the "Register query" button. The user can also save and load queries to the query catalogue, that is particularly important since users could use and extend queries written by others as well (i.e., *passive collaboration* [27]). The example query depicted in Figure 5, Figure 6 and Figure 7 presents the query example given in Figure 4.

OptiqueVQS has a backend composed of several components [43]. The core component is *graph projector* [**?**] enabling a graph-based navigation over an ontology during the query formulation process. It adapts a technique called *navigation graph* to extract a suitable graph-like structure from a set of OWL 2 axioms [**?**]. A *data sampler* component is also a part of the backend and it is used to enrich a given ontology with additional axioms to capture values from data that are frequently used and rarely changed. This includes the list of values and numerical ranges in an OWL data property range. Such an approach allows presenting attributes in different form elements, such as sliders, multi-select boxes, date pickers etc, with respect to the underlying data. Moreover, backend harvests the query log for *ranking and suggesting* query extensions as the user formulates a query, that is, the W1 and W3 lists concepts and properties adaptively with respect to the *partial query* the user has formulated so far [**?**].

OptiqueVQS is free of any SPARQL or OWL jargon and its usability is based on several design choices. W1, W2, and W3 provide a fine combination of *ontology exploration* (gradual and on-demand) and *query formulation*. W1 presents valid object property and range concept combinations in pairs for the pivot concept in order to reduce the number of navigational steps. W2 employs a tree-shaped query representation, rather than an arbitrary graph representation, to improve comprehensibility, and inverted object properties to restrict arcs to a single direction (i.e., left to right). Finally, W3 simplifies the type refinement by

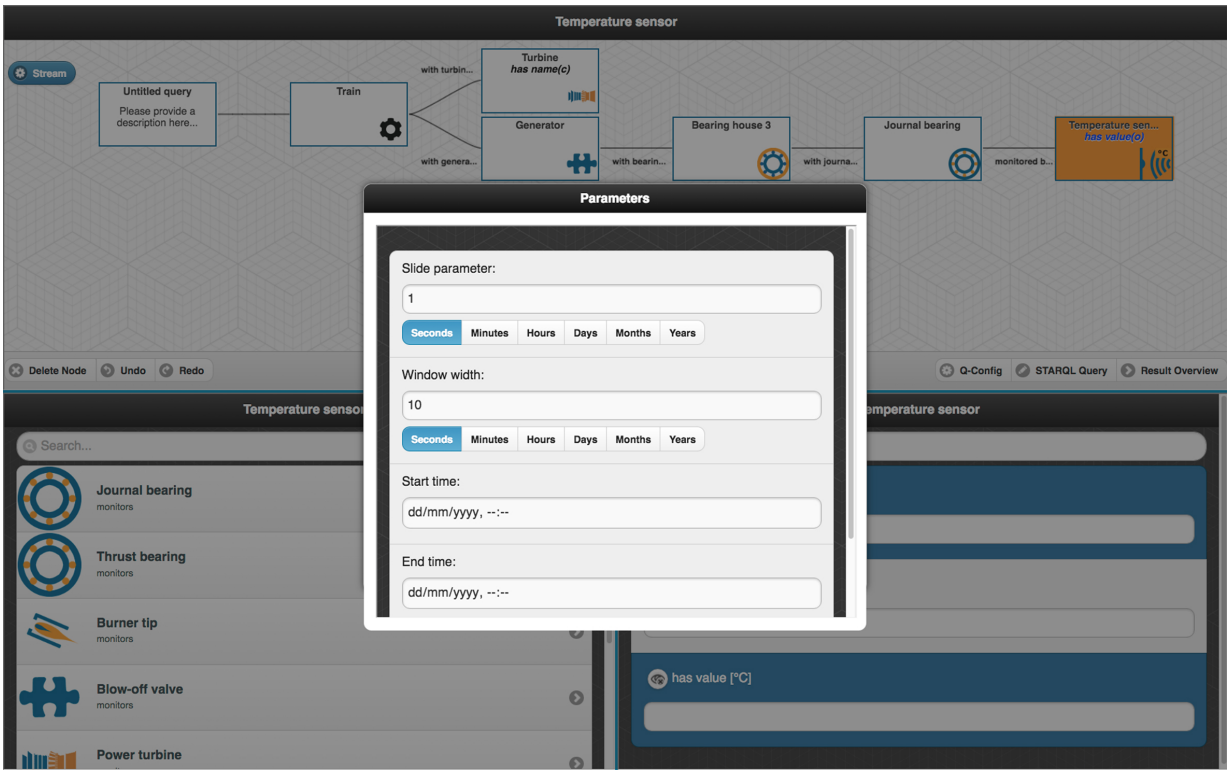

Fig. 6. OptiqueVQS with stream-temporal querying – parameter selection.

presenting the subclasses of a pivot in an ordinary form element. Regarding the expressiveness, OptiqueVQS currently supports tree-shaped conjunctive queries (including aggregation and excluding negation) and a restricted fragment of STARQL (e.g., no support for correlating multiple dynamic properties).

4. Evaluation

A *query catalogue* involving 40 representative queries for Siemens has been established (non-disclosable). An analysis of the query catalogue shows that 70% of queries are tree-shaped conjunctive queries and 65% are tree-shaped conjunctive queries excluding negation and including aggregation. This means that Siemens' engineers could potentially formulate 65% of their information needs with OptiqueVQS.

The solution presented in this article has been further evaluated in twofold:

(i) a proof of concept implementation has been realised for the transformation of STARQL queries to PostgreSQL and query execution times have been measured;

(ii) a user study has been conducted with domain experts from Siemens in order to measure the *effectiveness* and *efficiency* of OptiqueVQS.

4.1. Transformation of STARQL queries to PostgreSQL

The proof of concept implementation presented in this section is meant to show that the evaluation of STARQL is even possible on standard SQL engines like PostgreSQL. The evaluation strategy is based on the implementation of a system for answering historical queries. This system relies on recorded data and evaluates all temporal windows, which are created by a sliding widow operator, in a single calculation step. Therefore, the implemented transformation process consists of two phases. First, a view is generated that represents the temporal data linked to an additional window identifier (called *windowID*) that is produced by a window and pulse operator. In the second step the WHERE and HAVING clauses are translated into a second SQL view that evaluates the respective view based on the data aligned to each *windowId*.

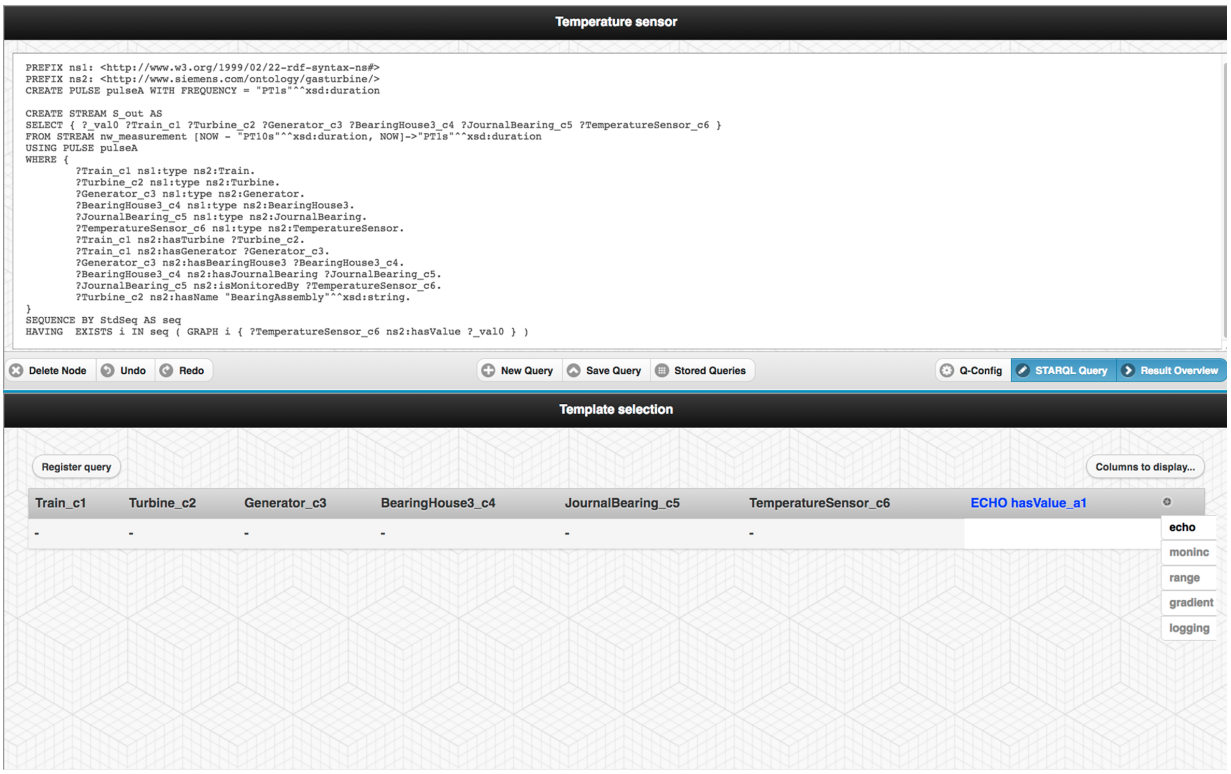

Fig. 7. OptiqueVQS with stream-temporal querying – template selection.

In the following, the translation result of the window operator is explained first.

WindowFunction: The window operator F^{winOp} takes a temporal table with a timestamp column and additional data columns as input and generates a SQL view of windows as output. The view groups all timestamp columns into a window group that is referenced by an additional column of *windowID*s. The window operator view can be seen as three applied functions, namely $F^{seqMeth}(F^{dataJoin}(F^{windowGen}))$. By the inner function $F^{windowGen}$, temporal borders are generated for each window, based on window parameters width w and slide sl that are defined in the window expression of each stream. The resulting view includes a sequence of *windowID*s with additional columns for temporal borders, i.e., the left and right borders of each window. Timestamps of the input table are used for a reference to the start and endpoint of each window, where start and end of the current window is given for every timestamp t by $t^{start} = \lfloor t/sl \rfloor \times sl$ and $t^{end} = max\{t^{start} - w, 0\}$.

As PostgreSQL includes functions for time series generation, the window generation step can be evaluated directly by the PostgreSQL processor (see view *Measurement_window* in the example of Figure 8).

JoinStream: In general, the implementation for a join on different input streams js can be seen as a simple join in SQL, but as there are different window parameters for each stream, they cannot be simply joined on the *windowID*. Furthermore, there is a need for a function that chooses at each *pulse* in time the related *windowID* of each input stream. Therefore, an additional global pulse function is calculated, which is defined by the user in the STARQL query, to synchronise the windows of all input streams.

All input streams are finally joined together based on the pulse frequency *pulse_freq* (in the example of Figure 8 from view *pulse_pulseA*) into the new recalculated joined pulse *pWindowID* that references the same windows for all streams (see view *Measurement_window_pulse* in the example of Figure 8). The recalculation is defined as follows:

$$pWindowID = \left\lfloor \frac{WindowID \times sl}{pulse_{freq}} \right\rfloor.$$

```
CREATE VIEW pulse_pulseA AS //a time series for the pulse is created
SELECT pulse AS time, row_number() OVER ( ORDER BY pulse ) - 1 AS wid
FROM ( SELECT generate_series ( $start$, $end$, $pulse_slide$ ) AS pulse) series;

CREATE VIEW Measurement_window AS //a time series for the window sequence is created
SELECT row_number() OVER ( ORDER BY time ) - 1 AS wid,
time - ($window_slide$) AS left, lead(time, 1) over(order by time) AS next, time
FROM ( SELECT generate_series ( $start$,$end$, $window_slide$) AS time ) series;

CREATE VIEW Measurement_window_pulse AS //the window sequence is filtered by pulse
SELECT p.wid AS pWid, m.left, m.time
FROM Measurement_window m RIGHT JOIN pulse_pulseA p ON p.time BETWEEN m.time
AND m.next AND m.next > p.time;

CREATE VIEW Measurement_data AS //the original data table is joined on timestamps
SELECT DISTINCT pWid AS wid, tble.*
FROM Measurement_public_window_pulse pj LEFT OUTER JOIN Measurement_public tble
ON tble.timestamp BETWEEN pj.left AND pj.time;

CREATE VIEW Measurement_stream AS //a temporal abox seq column is added
SELECT dense_rank() OVER
( PARTITION BY wid ORDER BY timestamp ASC ) AS abox, * FROM Measurement_public_data;
```

Fig. 8. A PostgreSQL transformation result for the STARQL window operator.

DataJoin: As soon as the historical window sequence is generated by the function $F^{windowGen}$, the second function $F^{dataJoin}$ can be applied on the window view by joining the result of the window generation with the actual input-data of the incoming stream (see view *Measurement_data* in the example of Figure 8).

SequencingFunction: The third function ($F^{seqMeth}$), which serves as a sequencing operator, adds an additional *ABoxID* column to the sliding window view (see view *Measurement_stream* in the example of Figure 8). In the current example this can be seen as a simple recalculation of the windowed timestamps 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 identical ABoxes afterwards.

The schematic transformation result of the transformed window operator for PostgreSQL is shown in Figure 8.

Transformation of WHERE and HAVING clause: The second transformation phase transforms the WHERE and HAVING clauses into a SQL view according to the algorithm presented by Neuenstadt et al. [29]. The resulting view evaluates the calculated windows of the window operator based on the given constraints in the STARQL query. The transformation result of the example from Figure 4 is given in Figure 9 (simplified) by the *S_out_having* view.

The SQL view consists of two parts: the transformation of the WHERE clause (SUB_WHERE) and the transformation of the HAVING clause (SUB_HAVING) that are both connected by a natural join. Both parts rely on mappings to a database schema of different tables on the PostgreSQL server, e.g., the *sensormetadata* table, which stores detailed information on all available sensors; while the HAVING clause also evaluates the previous described view of the window operator transformation result by referring to the *Measurement_stream* view. One can also see how the selection of sensors is restricted in the WHERE clause of the static part to temperature sensors on the *BearingAssembly*, while the transformed HAVING clause itself does not restrict any sensor types.

This section explained the transformation of historical STARQL queries into SQL that can be directly executed on PostgreSQL servers. The execution times of STARQL queries on a Postgres server are evaluated in the following subsection.

4.2. Experimental results

During the evaluation, the STARQL query given in Figure 4 has been taken and transformed based on the Optique prototype into a SQL based result (see Fig-

```
CREATE VIEW S_out_having AS
SELECT DISTINCT wid, TemperatureSensor_c6, _val0, Turbine_c2
FROM
  ( SELECT * FROM
    ( SELECT * FROM
      (
        SELECT "TemperatureSensor_c6", "Turbine_c2"
        FROM (
          SELECT DISTINCT
          ('http://www.siemens.com/ont[...]' || qview1."id") AS "Turbine_c2",
          ('http://www.siemens.com/ont[...]' || qview6."tagid") AS "TemperatureSensor_c6"
          FROM
            assembly qview1,
            sensormetadata qview6,
            partof qview7,
            sensor qview8
          WHERE
           (qview1."name" = 'BearingAssembly') AND
           (qview6."property" = 'Temperature') AND
           (qview6."location" = qview7."partid") AND
           (qview6."tagid" = qview8."id") AND
           (qview1."id" = qview8."assembly") AND
          ) SUB_QVIEW
        ) SUB_TRIPLE1
      ) SUB_WHERE /* end of WHERE-clause transformation */
    NATURAL JOIN
      (SELECT wid, __val0, _TemperatureSensor_c6 FROM
        (SELECT * FROM
          (SELECT wid, abox AS i, "TemperatureSensor_c6", "_val0"
           FROM
            (SELECT DISTINCT  qview2.wid,  qview2.abox, qview2."value" AS "_val0",
             ('http://www.siemens.com/ont[...]' || qview1."tagid") AS "TemperatureSensor_c6"
             FROM
               sensormetadata qview1,
               measurement_stream qview2
             WHERE
               (qview1."tagid" = qview2."sensor") AND
               (((((('Position' = qview1."property") OR
               ('PositionDemand' = qview1."property")) OR
               (qview1."property" LIKE 'Axial%')) OR
               (qview1."property" LIKE 'Rotation%')) OR
               ('Temperature' = qview1."property")) AND
                qview2."value" IS NOT NULL AND
                qview1."tagid" IS NOT NULL
             )SUB_QVIEW
           ) SUB_TRIPLE0
         )SUB_QVIEW
       )SUB_HAVING /* end of HAVING-clause transformation */
     )SUB_FROM;
```

Fig. 9. A PostgreSQL transformation result for STARQL WHERE and HAVING clauses.

ure 9). It has been executed on a reference dataset that includes the signals of 25 sensors. These sensors emit values irregularly over a time of approximately one month. The dataset, which is sampled in seconds, can be seen as a representation of real-world scenarios that also occur in Siemens' data centres.

The query has been evaluated over a PostgreSQL installation on a machine with a i7 2.4 GHz core and 8GB of RAM. The standard SQL database executes queries in a process based way, and therefore, cannot distribute its execution directly over cores or machines. The execution time has been measured in two steps. First, the time that is used by the window operator to generate the complete sequence of windows on the whole input data set has been measured and second, the evaluation of all windows with respect to the given queries has been measured.

The results for different window and data sizes can be found in Table 1. They basically show that the query can even be executed on standard SQL engines in the range of one minute, while a larger amount of time is used for the execution of the window operator. That is as expected, because the PostgreSQL engine does not natively support the execution of a window operator.

Furthermore, STARQL has been also tested on distributed and scalable environments having a native window operator like Exareme[8] [21].

4.3. User study

A user experiment has been conducted at Siemens for OptiqueVQS with stream-temporal support [45**?**]. The experiment was designed as a *think-aloud study*, since the goal of the experiment was not purely *summative*, but to a large extent *formative*. The experiment was built on a "turbine ontology" with 40 concepts and 65 properties.

Three participants, who cover the relevant occupation profiles, have taken part in the experiment; the profiles of participants are summarised in Table 2. A brief introduction on the topic and tool was delivered to the participants along with a simple example. Then they were asked to fill in a *profile survey*. The survey asks users about their age, occupation and level of education, and asks them to rate their technical skills, such as on programming and query languages, knowledge on the semantic web technologies, and their familiarity with similar tools on a Likert scale (i.e., 1 for "not familiar at all," 5 for "very familiar"). Participants were then asked to formulate a series of information needs into actual queries with OptiqueVQS, given at most three *attempts* for each query. Each participant performed the experiment in a dedicated session, while being observed by a surveyor. Participants were instructed to think aloud, including any difficulties they encountered, while performing the given tasks. Table 3 lists the tasks representing the information needs used in the experiment (tasks 3-5 are stream queries).

Once users were done with the tasks, they were asked to fill in an *exit survey* asking about their experiences with the tool in order to measure *user satisfaction*. The survey asks users to rate whether the questions were easy to do with the tool (*S1*), the tool was easy to learn (*S2*), was easy to use (*S3*), gave a good feeling of control and awareness (*S4*), was aesthetically pleasing (*S5*), was overall satisfactory (*S6*), and was enjoyable to use (*S7*) on a Likert scale (again, 1 for "strongly disagree" and 5 for "strongly agree"). Users were also asked to comment on what they did like and dislike about the tool and to provide any feedback which they deem important.

Since OptiqueVQS is a *data retrieval* (DR) tool, where a single missing or irrelevant object implies a total failure contrary to *information retrieval* (IR) [**? ?**], effectiveness is measured in terms of a binary measure of success (i.e., correct/incorrect query) and efficiency is measured in terms of total time taken to formulate a query [39,47]. The results of the experiment are presented in Table 4. In total, 15 tasks were completed by the participants with 100 percent correct completion rate and 66 percent first-attempt correct completion rate; and, on average, a task took 143 seconds to complete in 1.3 attempts. First-attempt correct completion refers to the cases where a user formulates a correct query for a given task at his/her first attempt. One should be aware that query formulation is an *iterative process* [50,27]; therefore, *query reformulation* is a natural step. Nevertheless, the results indicate that domain experts could formulate queries with high efficiency and effectiveness by using OptiqueVQS. The feedback provided by the participants through the exit survey is presented in Table 5 and Table 6. The usability scores given by participants are quite high and their comments suggest that they did like the design of interface. Users generally praised the capabilities and the design of OptiqueVQS.

R&D engineers asked for advanced operators such as "OR" and negation, since they, compared to diagnosis engineers, often need to formulate more complex

[8] `http://madgik.github.io/exareme`

Table 1
Result overview for window operator and query evaluation.

Window Width	Data Set	Window Gen Time	Window Eval Time
1 Second	1 Day	2.4 sec	0.468 sec
	1 Week	28.1 sec	3.5 sec
	1 Month	1 min 14 sec	1 min 17 sec
10 Seconds	1 Day	14.9 sec	18.2 sec
	1 Week	1 min 38 sec	35.2 sec
	1 Month	2 min 44 seec	2 min 54 sec

Table 2
Profile information of the participants.

#	Age	Occupation	Education	Technical skills	Semantic Web	Similar tools
P1	37	R&D engineer	PhD	4	1	1
P2	54	Diagnostics Engineer	Bachelor	5	1	3
P3	39	Engineer	PhD	5	1	2

Table 3
Information needs used in the user experiment.

#	Information need
T1	Display all trains that have a turbine and a generator.
T2	Display all turbines together with the temperature sensors in their burner tips. Be sure to include the turbine name and the burner tags.
T3	For the turbine named “Bearing Assembly”, query for temperature readings of the journal bearing in the compressor. Display the reading as a simple echo.
T4	For a train with turbine named “Bearing Assembly”, query for the journal bearing temperature reading in the generator. Display readings as a simple echo.
T5	For the turbine named “Burner Assembly”, query for all burner tip temperatures. Display the readings if they increase monotonically.

Table 4
The results of the user experiment (*c* for complete, *t* for time in seconds, and *a* for attempt count).

	T1			**T2**			**T3**			**T4**			**T5**			**Av.**		
#	*c*	*t*	*a*	*c*	*t*	*a*	*c*	*t*	*a*	*c*	*t*	*a*	*c*	*t*	*a*	*c*	*t*	*a*
P1	1	120	1	1	150	1	1	130	1	1	70	1	1	60	1	1	106	1.0
P2	1	120	1	1	180	2	1	240	2	1	60	1	1	180	1	1	156	1.4
P3	1	45	1	1	40	2	1	40	2	1	60	2	1	60	1	1	49	1.6
Av.	1	95	1	1	123	1.6	1	136	1.6	1	63	1.3	1	100	1	1	103	1.3

Table 5
The results of the exit survey.

Question	P1	P2	P3	Avg.
“I think that I would like to use this system frequently.”	5	4	4	4.3
“I found the system unnecessarily complex.”	1	3	2	2.0
“I thought the system was easy to use.”	5	4	5	4.6
“I think that I would need the support of a technical person to be able to use this system.”	1	1	1	1.0
“I found the various functions in this system were well integrated.”	4	4	4	4.0
“I thought there was too much inconsistency in this system.”	2	2	2	2.0
“I would imagine that most people would learn to use this system very quickly.”	5	4	4	4.3
“I found the system very cumbersome to use.”	1	2	2	1.6
“I felt very confident using the system.”	4	4	3	3.6
“I needed to learn a lot of things before I could get going with this system.”	2	1	1	1.3

Table 6
The feedback given by the participants.

What did you like about the tool?	**Person**
"Easy to learn"	P1
"Nice user interface"	P1
"Possibility to see the original query"	P1
"Very comfortable to use"	P2
"UI looks really nice"	P3
"The floating tree shows exactly what kind of situation I am looking for. Gives a nice overview."	P3
"The interaction between the buttons and the tree work really well."	P3
"The turbine structure is really useful to find sensors quickly."	P3
"Very nice icons for the turbine parts."	P3
"Especially complex queries appear easy to understand."	P3
What didn't you like about the tool?	**Person**
"Should be possible to extend search for things not directly connected to the current concept."	P1
"When selecting a turbine name, the turbine box in the tree does not show me the turbine name but only c. I find this confusing."	P2
"Did not always know where to click for the stream part. E.g., the little circle on in the column."	P3
"It may be confusing to have to run the query before specifying it further. Could it be run automatically, e.g., after each change to the tree?"	P3
"Did not know what the start time and end time field means for a stream. Is that automatically registering / de-registering the query at a certain time point?"	P3
"I do not understand what the numbers on the buttons mean. Is that the number of instances of the item (i.e., turbine.)"	P3
"I find the order of items confusing. This is not alphabetical and also does not make sense from the structure of the turbine."	P3
"Why am I offered sensors that don't exist at certain locations? For instance, I see 'RotationSpeed' for the burners?"	P3

queries. The participants wished for the ability to combine multiple queries and connect concepts that are not directly linked (i.e., *non-local navigation* [**?** 47]). Participants also wanted attributes and attribute values in W3 to be filtered automatically with respect to previously selected constraints (including type refinement) – i.e., similar to *faceted-search* (cf. [**?**]). Non-local navigation and faceted-search like filtering are challenging with large ontologies and data sources. This is because, for the former, most relevant connections between two concepts must be found, and for the latter, selected constraints must be checked against data to filter out attributes and attribute values.

Overall, the high completion and satisfaction rates suggest that domain experts are quite comfortable with OptiqueVQS's stream-temporal capabilities. The study also shows that the learnability of OptiqueVQS is high as participants did not receive any substantial training. Earlier, three other user experiments were conducted: one with casual users [44] and two others with domain experts at Statoil and Siemens on non-streaming scenarios [43,20]. These studies revealed similar results and confirmed that OptiqueVQS could address various user types and different scenarios with high efficiency, effectiveness and user satisfaction.

5. Related Work

A high majority of work on stream processing is realised in the context of *data stream management systems* (DSMSs). They mainly extend relational model to support continuous queries with declarative languages analogous to SQL such as CQL [2], TelegraphCQ [9], Aurora/Borealis [19], and PIPES [23]. Notable examples of stream query languages in the Semantic Web are Streaming SPARQL [**?**], C-SPARQL [3], SPARQLstream [4], RSP-QL [11], CQELS [25], and EP-SPARQL [**?**]. However, they either have no implemented and/or optimised engine or the engine is still not fully developed [**?**]. These approaches usually extend SPARQL with a window operator whose content is a multi-set of variable bindings for the open variables in the query. Among them, only SPARQLstream and STARQL support ontology based data access approach. However, SPARQLstream query language does not support historic data and its engine provides no reasoning support, while STARQL does. STARQL also offers more advanced user-defined functions from the Optique backend system like Pearson correlation [21]. Özcep et al. [30] compare and discuss advantages and disadvantages of different approaches.

Visual query formulation is a long-standing endeavour, and as such it has accumulated a considerable number of studies over years. Majority of these studies are within the relational database community (cf. [8]), for instance, query by example (QBE) [52]. Attempts in the semantic web community is quite recent; however, several visual tools exist for SPARQL (cf. [47]). A variety of such approaches could be classified as VQLs, such as RDF-GL [18], Nitelight [37], and QueryVOWL [16]. However, VQLs are still comparable to formal textual query languages as users need to have knowledge and skills to understand the underlying visual notation and syntax. VQSs offer a good balance between usability and expressiveness; examples include OZONE [49], Konduit VQB [1], and gFacet [17]. However, none of these approaches support querying stream-temporal data sources. Only example supporting stream-temporal querying is SPARQL/CQELS visual editor designed for Super Stream Collider framework [33]. The tool follows the jargon of the underlying language closely and; therefore, it is not appropriate for end users without technical skills.

OptiqueVQS is valuable also conceptually as an instance of the *end-user programming* [26] paradigm in pervasive environments, which aims to empower end users to orchestrate data and objects (e.g., sensors, actuators, and appliances) distributed across the digital ecosystem on their own, such as for *activity recognition* [24] and *self-monitoring* [31]. This is because the abundance of data and internet-connected objects render it difficult for IT experts to consider all possible eventualities and develop solutions addressing broader contexts. The OBDA platform underlying OptiqueVQS [15,14] presents an example where ontologies could help building scalable and efficient architectures for data retrieval, integration, and access in pervasive environments.

6. Conclusion

OptiqueVQS with stream-temporal querying has been developed in an industrial context with real requirements. Its main design goal is to provide a fine balance between usability and expressiveness. Domain experts at Siemens have used OptiqueVQS for querying streaming sensor data with high efficiency and effectiveness. The underlying formal textual query language, STARQL, and OBDA platform, Optique, are mature enough to promote ontologies and OptiqueVQS as a realistic solution for querying dynamic industrial scale data sources.

The future work involves extending the functionality of OptiqueVQS to cover a larger fragment of STARQL, while maintaining its high usability. This includes stream-temporal specific functionalities, such as ability to correlate multiple dynamic properties; and generic functionalities, such as simpler forms of negation and disjunction (e.g., only over data properties). The current widget-based architecture and design of OptiqueVQS provide us with a sufficient room for a sustainable evolution, where new functionality could be distributed to different widgets and complex functionality could be hidden behind layers.

References

[1] O. Ambrus, K. Möller, and S. Handschuh. Konduit VQB: A Visual Query Builder for SPARQL on the Social Semantic Desktop. In *Proceedings of the Workshop on Visual Interfaces to the Social and Semantic Web (VISSW 2010)*, volume 565 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2010.

[2] A. Arasu, S. Babu, and J. Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *The VLDB Journal*, 15(2):121–142, 2006.

[3] 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 (WWW 2009)*, pages 1061–1062. ACM, 2009.

[4] 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 (ISWC 2010)*, volume 6496 of *LNCS*, pages 96–111. Springer, 2010.

[5] D. Calvanese, G. 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.

[6] 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. *Semantic Web*, in press.

[7] T. Catarci. What happened when database researchers met usability. *Information Systems*, 25(3):177–212, 2000.

[8] T. Catarci, M. F. Costabile, S. Levialdi, and C. Batini. Visual query systems for databases: A survey. *Journal of Visual Languages and Computing*, 8(2):215–260, 1997.

[9] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous Dataflow Processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 2003)*, pages 668–668. ACM, 2003.

[10] 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. *Proceedings of the VLDB Endowment*, 6(12):1314–1317, 2013.

[11] D. Dell'Aglio, E. Della 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 & Information System*, 10(4):17–44, 2014.

[12] A. Doan, A. Halevy, and Z. Ives. *Principles of Data Integration*. Morgan Kaufmann, 1st edition edition, 2012.

[13] R. G. Epstein. The TableTalk Query Language. *Journal of Visual Languages and Computing*, 2(2):115–141, 1991.

[14] M. Giese, D. Calvanese, P. Haase, I. Horrocks, Y. Ioannidis, H. Kllapi, M. Koubarakis, M. Lenzerini, R. Möller, O. Özcep, M. Rodriguez-Muro, R. Rosati, R. Schlatte, M. Schmidt, A. Soylu, and A. Waaler. Scalable End-user Access to Big Data. In R. Akerkar, editor, *Big Data Computing*. CRC Press, 2013.

[15] M. Giese, A. Soylu, G. Vega-Gorgojo, A. Waaler, P. Haase, E. Jimenez-Ruiz, D. Lanti, M. Rezk, G. Xiao, O. Ozcep, and R. Rosati. Optique – Zooming In on Big Data Access. *IEEE Computer*, 48(3):60–67, 2015.

[16] F. Haag, S. Lohmann, S. Siek, and T. Ertl. QueryVOWL: A Visual Query Notation for Linked Data. In *Proceedings of the Satellite Events of the 12th European Conference on the Semantic Web (ESWC 2015)*, volume 9341 of *LNCS*, pages 387–402. Springer, 2015.

[17] P. Heim and J. Ziegler. Faceted Visual Exploration of Semantic Data. In *Proceedings of the First IFIP WG 13.7 International Workshop on Human Aspects of Visualization (HCIV 2009)*, volume 6431 of *LNCS*, pages 58–75. Springer, 2011.

[18] F. Hogenboom, V. Milea, F. Fransincar, and U. Kaymak. RDF-GL: A SPARQL-Based Graphical Query Language for RDF. In *Emergent Web Intelligence: Advanced Information Retrieval*, Advanced Information and Knowledge Processing, pages 87–116. Springer, 2010.

[19] J. H. Hwang, Y. Xing, U. Cetintemel, and S. Zdonik. A Cooperative, Self-Configuring High-Availability Solution for Stream Processing. In *Proceedings of the IEEE 23rd International Conference on Data Engineering (ICDE 2007)*, IEEE, pages 176–185, 2007.

[20] 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 *Proceedings of the 13th International Semantic Web Conference (ISWC 2014)*, volume 8796 of *LNCS*, pages 601–619. Springer, 2014.

[21] 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. In *Proceedings of the 15th International Semantic Web Conference (ISWC 2016)*, 2016.

[22] M. R. Kogalovsky. Ontology-Based Data Access Systems. *Programming and Computer Software*, 38(4):167–182, 2012.

[23] J. Krämer and B. Seeger. Semantics and Implementation of Continuous Sliding Window Queries over Data Streams. *ACM Transactions on Database Systems*, 34(1):4:1–4:49, 2009.

[24] N. C. Krishnan and D. J. Cook. Activity Recognition on Streaming Sensor Data. *Pervasive and Mobile Computing*, 10: 138–154, 2014.

[25] D. Le-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 *Proceedings of the 10th International Conference on The Semantic Web (ISWC 2011)*, volume 7031 of *LNCS*, pages 370–388. Springer, 2011.

[26] H. Lieberman, F. Paterno, and V. Wulf, editors. *End User Development*, volume 9 of *Human-Computer Interaction Series*. Springer, 2006.

[27] G. Marchionini and R. White. Find What You Need, Understand What You Find. *International Journal of Human-Computer Interaction*, 23(3):205–237, 2007.

[28] C. Martinez-Cruz, I. J. Blanco, and M. Amparo Vila. Ontologies versus relational databases: are they so different? A comparison. *Artificial Intelligence Review*, 38(4):271–290, 2012.

[29] C. Neuenstadt, R. Möller, and Özgür. L. Özçep. OBDA for Temporal Querying and Streams with STARQL. In *Proceedings of the First Workshop on High-Level Declarative Stream Processing (HiDeSt 2015)*, volume 1447 of *CEUR Workshop Proceedings*, pages 70–75. CEUR-WS.org, 2015.

[30] O. L. Ozcep, R. Moller, and C. Neuenstadt. A Stream-Temporal Query Language for Ontology Based Data Access. In *The 37th Annual German Conference on Artificial Intelligence (KI 2014)*, volume 8736 of *LNCS*, pages 183–194. Springer, 2014.

[31] D. Pavel, V. Callaghan, and A. K. Dey. Looking Back in Wonder: How Self-Monitoring Technologies Can Help Us Better Understand Ourselves. In *Proceedings of the 2010 Sixth International Conference on Intelligent Environments (IE 2010)*, pages 289–294. IEEE Computer Society, 2010.

[32] A. Poggi, D. Lembo, D. Calvanese, G. De Giacomo, M. Lenzerini, and R. Rosati. Linking data to ontologies. *Journal on Data Semantics X*, 10:133–173, 2008.

[33] H. N. M. Quoc, M. Serrano, D. L. Phuoc, and M. Hauswirth. Super Stream Collider: Linked Stream Mashups for Everyone. In *Proceedings of the Semantic Web Challenge at ISWC2012*, 2012.

[34] R. W. Revie, editor. *Oil and Gas Pipelines: Integrity and Safety Handbook*. John Wiley & Sons, Inc., 2015.

[35] L. Ruiz-Garcia, L. Lunadei, P. Barreiro, and J. Ignacio Robla. A Review of Wireless Sensor Technologies and Applications in Agriculture and Food Industry: State of the Art and Current Trends. *Sensors*, 9(6):4728–4750, 2009.

[36] 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. .

[37] P. R. Smart, A. Russell, D. Braines, Y. Kalfoglou, J. Bao, and N. Shadbolt. A Visual Approach to Semantic Query Design Using a Web-Based Graphical Query Designer. In *Proceedings of the 16th International Conference on Knowledge Engineering and Knowledge Management (EKAW 2008)*, volume 5268 of *LNCS*, pages 275–291. Springer, 2008.

[38] K. Soon Low, W. N. N. Win, and M. Joo Er. Wireless Sensor Networks for Industrial Environments. In *Proceedings of the International Conference on Computational Intelligence for Modelling, Control and Automation and International Conference on Intelligent Agents, Web Technologies and Internet Commerce (CIMCA-IAWTIC 2006)*, pages 271–276. IEEE, 2005.

[39] A. Soylu and M. Giese. Qualifying Ontology-based Visual Query Formulation. In *Proceedings of the 11th International Conference Flexible Query Answering Systems (FQAS 2015)*, volume 400 of *Advances in Intelligent Systems and Computing*, pages 243–255. Springer, 2015.

[40] A. Soylu, P. De Causmaecker, D. Preuveneers, Y. Berbers, and P. Desmet. Formal modelling, knowledge representation and reasoning for design and development of user-centric pervasive software: a meta-review. *International Journal of Metadata, Semantics and Ontologies*, 6(2):96–125, 2011.

[41] A. Soylu, F. Wild, F. Mödritscher, P. Desmet, S. Verlinde, and P. De Causmaecker. Mashups and widget orchestration. In *Proceedings of the International Conference on Management of Emergent Digital EcoSystems (MEDES 2011)*, pages 226–234. ACM, 2011.

[42] A. Soylu, F. Moedritscher, F. Wild, P. De Causmaecker, and P. Desmet. Mashups by orchestration and widget-based personal environments: Key challenges, solution strategies, and an application. *Program: Electronic Library and Information Systems*, 46(4):383–428, 2012.

[43] A. Soylu, E. Kharlamov, D. Zheleznyakov, E. Jimenez-Ruiz, M. Giese, and I. Horrocks. Ontology-based Visual Query Formulation: An Industry Experience. In *Proceedings of the 11th International Symposium on Visual Computing (ISVC 2015)*, volume 9474 of *LNCS*, pages 842–854. Springer, 2015.

[44] A. Soylu, M. Giese, E. Jimenez-Ruiz, G. Vega-Gorgojo, and I. Horrocks. Experiencing OptiqueVQS – a multi-paradigm and ontology-based visual query system for end-users. *Universal Access in the Information Society*, 15(1):129–152, 2016.

[45] A. Soylu, M. Giese, R. Schlatte, E. Jimenez-Ruiz, O. Ozcep, and S. Brandt. Domain Experts Surfing on Stream Sensor Data over Ontologies. In *Proceedings of the 1st International Workshop on Semantic Web Technologies for Mobile and Pervasive Environments (SEMPER 2016)*, volume 1588 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2016.

[46] A. Soylu, M. Giese, R. Schlatte, E. Jimenez-Ruiz, O. Ozcep, and S. Brandt. A Visual Query System for Stream Data Access over Ontologies. In *Proceedings of the Satellite Events of the 13th European Conference on the Semantic Web (ESWC 2016)*, volume 9989 of *LNCS*. Springer, 2016.

[47] A. Soylu, M. Giese, E. Kharlamov, E. Jimenez-Ruiz, D. Zheleznyakov, and I. Horrocks. Ontology-based End-user Visual Query Formulation: Why, what, who, how, and which? *Universal Access in the Information Society*, in press.

[48] D.-E. Spanos, P. Stavrou, and N. Mitrou. Bringing relational databases into the Semantic Web: A survey. *Semantic Web*, 3 (2):169–209, 2012.

[49] B. Suh and B. B. Bederson. OZONE: A Zoomable Interface for Navigating Ontology Information. In *Proceedings of the Working Conference on Advanced Visual Interfaces (AVI 2002)*, pages 139–143. ACM, 2002.

[50] V. Uren, Y. Lei, V. Lopez, H. Liu, E. Motta, and M. Giordanino. The Usability of Semantic Search Tools: A Review. *The Knowledge Engineering Review*, 22(4):361–377, 2007.

[51] Y. Yang, X. Wu, and X. Zhu. Combining Proactive and Reactive Predictions for Data Streams. In *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining (KDD 2005)*, pages 710–715. ACM, 2005.

[52] M. M. Zloof. Query-by-example: A Data Base Language. *IBM Systems Journal*, 16(4):324–343, 1977.