Demo: Enabling Semantic Access to Static and Streaming Distributed Data with Optique*

E. Kharlamov[1] S. Brandt[2] M. Giese[3] E. Jiménez-Ruiz[1] Y. Kotidis[4] S. Lamparter[2] T. Mailis[5]
C. Neuenstadt[6] Ö. Özçep[6] C. Pinkel[7] A. Soylu[8] C. Svingos[4] D. Zheleznyakov[1]
I. Horrocks[1] Y. Ioannidis[4] R. Möller[6] A. Waaler[3]

[1] University of Oxford [2] Siemens CT [3] University of Oslo [4] Athens University of Economics and Business
[5] University of Athens [6] University of Lübeck [7] fluid Operations AG [8] Norwegian Uni. of Science and Technology

ABSTRACT

Real-time processing of data coming from multiple heterogeneous data streams and static databases is a typical task in many industrial scenarios such as diagnostics of large machines. A complex diagnostic task may require a collection of up to hundreds of queries over such data. Although many of these queries retrieve data of the same kind, such as temperature measurements, they access structurally different data sources. In this work, we show how Semantic Technologies implemented in our system OPTIQUE can simplify such complex diagnostics by providing an abstraction layer—ontology—that integrates heterogeneous data. In a nutshell, OPTIQUE allows complex diagnostic tasks to be expressed with just a few high-level semantic queries, which can be easily formulated with our visual query formulation system. OPTIQUE can then automatically enrich these queries, translate them into a large collection of low-level data queries, and finally optimise and efficiently execute the collection in a heavily distributed environment.

CCS Concepts

•**Information systems** → **Mediators and data integration;**

Keywords

Data Access, Information Integration, Ontologies, Streaming Data

1. INTRODUCTION

Motivation. *Siemens* runs service centres dedicated to diagnostics of thousands of power-generation appliances across the globe. One typical task for these centres is to detect in real-time potential failure events caused by, e.g., an abnormal temperature and pressure increase. Such tasks require simultaneous processing of *(i)* sequences of digitally encoded coherent signals produced and transmitted from thousands of gas and steam turbines, generators, and compressors installed in power plants, and *(ii)* static data that include the structure of relevant equipment, history of its exploitation and repairs, and even weather conditions. These data are scattered across a large number of heterogeneous data streams in addition to static DBs with hundreds of TBs of data.

Even for a single diagnostic task, such as checking if a given turbine might develop a fault, Siemens engineers have to analyse streams with temperature and other measurements from up to $2,000$ sensors installed in different parts of the turbine, analyse historical temperature data, compute temperature patterns, compare them to patterns in other turbines, compare weather conditions, etc. This requires to pose a collection of hundreds of queries, the majority of which are semantically the same (they ask about temperature), but syntactically different (they are over different schemata). Formulating and executing so many queries, and then assembling the computed answers, takes up to 80% of the overall diagnostic time [8].

Our Proposal. In order to streamline the diagnostic process at Siemens, we propose a data integration approach based on Semantic Technologies. In this paper we will refer to our approach as *Ontology-Based Stream-Static Data Integration* (*OBSSDI*). It follows the classical data integration paradigm that requires the creation of a common 'global' schema that consolidates 'local' schemata of the integrated data sources, and mappings that define how the local and global schemata are related [3]. In *OBSSDI* the global schema is an *ontology*: a formal conceptualisation of the domain of interest that consists of a *vocabulary*, i.e., names of classes, attributes and binary relations, and *axioms* over the terms from the vocabulary that, e.g., assign attributes of classes, define relationships between classes, compose classes, class hierarchies, etc. *OBSSDI* mappings relate each ontological term to a set of queries over the underlying data. For example, the generic ontology attribute *temperature-of-sensor* is mapped to all specific data and procedures that return temperature readings from sensors in dozens of different turbines and DBs storing historical data, thus, all particularities and varieties of how the temperature of a sensor can be measured, represented and stored are captured in these mappings. In *OBSSDI* the integrated data can be accessed by posing queries over the ontology, i.e., *ontological queries*. These queries are *hybrid*: they refer to both streaming and static data. Evaluation of an ontological query in *OBSSDI* has three stages: *(i)* in the *enrichment* stage ontology axioms are used to expand the ontological query in order to access as much of relevant data as possible; *(ii)* in the *unfolding* stage the mappings are used to translate the enriched ontological query into (possibly many) queries over the data; and *(iii)* in the *execution* stage the unfolded data queries are executed over the data.

Our Contributions. We developed a system OPTIQUE that implements *OBSSDI* and has the following novel components:

(C1) Semi-automatic support to construct high quality ontologies and mappings over relational and streaming data.

(C2) Query language over ontologies that combines streaming and static data, and allows for efficient enrichment and unfolding that preserves the semantics of ontological queries.

*This paper extends our earlier accepted demo [4] with a more detailed demo scenario and the STREAMVQS system.

DEBS '16 June 20-24, 2016, Irvine, CA, USA

ACM ISBN 978-1-4503-4021-2/16/06.

DOI: http://dx.doi.org/10.1145/2933267.2933290

(C3) End-user oriented query formulation support to construct continuos ontological queries.
(C4) Backend for optimising large numbers of queries automatically generated via enrichment and unfolding, and efficiently execute them over distributed streaming and static data.

The component C1 is practically important since such support can dramatically speed up deployment and maintenance (e.g., adjustment to new query requirements) of *OBSSDI* systems. The component C2 is crucial since, to the best of our knowledge, no dedicated query language for hybrid semantic queries has the required properties. The component C3 is essential since it allows for fast and easy data access for non-experts to state-of-the-art technologies. The component C4 is vital since even in the context where the data is only static and not distributed, query execution without dedicated optimisation techniques performs poorly since the queries that are automatically computed after enrichment and unfolding can be very inefficient, e.g., they may contain many redundant joins and unions [2]. See Section 2 for more details on the OPTIQUE components.

Demo Overview. Attendees will see how OPTIQUE simplifies diagnostics for Siemens: how to set and monitor continuous diagnostic tasks, how the system can handle more than a thousand complex diagnostic tasks, and how to deploy OPTIQUE over Siemens data. See Section 3 for more details on demo scenarios.

2. OPTIQUE SYSTEM

OPTIQUE is an integrated system that consist of multiple components to support *OBSSDI* end-to-end [7, 9, 10]. For IT specialists OPTIQUE offers support for the whole lifecycle of ontologies and mappings: semi-automatic bootstrapping from relational data sources, importing of existing ontologies, semi-automatic quality verification and optimisation, cataloging, manual definition and editing of mappings. For end-users OPTIQUE offers tools for query formulation support, query cataloging, answer monitoring, as well as integration with GIS systems. Query evaluation is done via OPTIQUE's query enrichment, unfolding, and execution backends that allow to execute up to thousands complex ontological queries in highly distributed environments. We now give some details of the C1-C4 OPTIQUE components.

Deployment Support. In order to support OPTIQUE's deployment we developed a BOOTOX [6, 13] system for "bootstrapping" (i.e., extracting) W3C standardised OWL 2 ontologies and R2RML mappings from static and streaming relational schema and data. Consider, e.g., a class *Turbine*; a mapping for it is an expression of the form: $\mathit{Turbine}(f(\vec{x})) \leftarrow \exists \vec{y}\ \mathtt{SQL}(\vec{x}, \vec{y})$, that can be seen as a view definition, where $\mathtt{SQL}(\vec{x}, \vec{y})$ is an SQL query, $\vec{x}$ are its output variables, $\vec{y}$ are its variables that are projected out and f is a function that converts tuples returned by SQL into identifiers of objects populating the class *Turbine*. Intuitively, mapping bootstrapping of BOOTOX boils down to discovery of 'meaningful' queries $\exists \vec{y}\ \mathtt{SQL}(\vec{x}, \vec{y})$ over the input data sources that would correspond to either a given element of the ontological vocabulary, e.g., the class *Turbine* or attribute *temperature-of-sensor*, or to a new ontological term. BOOTOX employs several novel schema- and data-driven query discovery techniques. The ontological terms bootstrapped by with BOOTOX provide the vocabulary for the formulation of STARQL ontological queries. We now discuss STARQL queries and how we process them.

Diagnostic Queries. In order to formulate diagnostic tasks as semantic queries that blend streaming with static data, we developed a query language STARQL [12]. The syntax of STARQL extends so-called *basic graph patterns* of W3C standardised SPARQL query language for RDF databases. STARQL queries can express basic graph patterns, and typical mathematical, statistical, and event pattern features needed in real-time diagnostic scenarios. Moreover, STARQL queries can be nested, in the sense that the result of one query may be used in the construction of another query. STARQL has a formal semantics that combines open and closed-world reasoning and extends snapshot semantics for window operators [1] with sequencing semantics that can handle integrity constraints such as functionality assertions.

STARQL has favourable computational properties [12]: despite its expressivity, answering STARQL queries is efficient since they can be efficiently enriched and then unfolded into efficient relational stream queries. STARQL query enrichment is polynomial-time in the size of the input ontology if the ontology is expressed in the OWL 2 QL ontology language and the queries are essentially conjunctive with value comparison and aggregates. STARQL assumes global-as-view mappings [3] of ontological terms to underlying data and thus unfolding of STARQL queries is linear-time in the size of both mappings and query and enriched STARQL queries can be unfolded into relational stream queries. We developed a dedicated STARQL2SQL$^{(+)}$ translator that unfolds STARQL queries to SQL$^{(+)}$ queries, i.e. SQL queries enhanced with the essential operators for stream handling.

Streaming and Static Relational Data Processing. Relational queries produced by the STARQL2SQL$^{(+)}$ translation, are handled by EXASTREAM, OPTIQUE's high-throughput distributed *Data Stream Management System* (*DSMS*). The EXASTREAM *DSMS* is embedded in EXAREME, a system for elastic large-scale dataflow processing in the cloud [11, 16]. In the following, we present some key aspects of EXASTREAM.

EXASTREAM is built as a streaming extension of the SQLite *DBMS*, taking advantage of existing Database Management technologies and optimisations such as *query planners*. It provides a declarative language, namely SQL$^{(+)}$, for querying data streams and relations that conform to the CQL semantics [1]. EXASTREAM natively supports *User Defined Functions* (*UDFs*) with arbitrary user code. The engine blends the execution of *UDFs* together with relational operators using JIT tracing compilation techniques speeding up the execution time. *UDFs* allow to express very complex dataflows using simple primitives. For OPTIQUE we used *UDFs* to implement communication with external sources, window partitioning on data streams, data mining algorithms such as the *Locality-Sensitive Hashing* technique [5] for computing the correlation between values of multiple streams. More importantly, the main operators that incorporate the algorithmic logic for transforming SQLite into a *DSMS* are implemented as *UDFs*.

In order to enable efficient processing of data streams of very high velocity we have implemented a number of optimisations in the stream processing engine, such as *adaptive indexing*. With this technique EXASTREAM collects statistics during query execution and, adaptively, decides to build main-memory indexes on batches of cached stream tuples in order to expedite query processing.

Visual Query System. Most diagnostic engineers cannot be expected to learn a formal query language like STARQL. OPTIQUE therefore contains a visual query system, STREAMVQS (a variant of OPTIQUEVQS [14, 15]), that makes it easy for users without IT background to formulate the most commonly needed queries. Due to usability considerations STREAMVQS supports only the essential, i.e., the most frequently used fragment of STARQL that corresponds to tree-shaped conjunctive queries with aggregates and stream related constructors such as window width and slide parameters. STREAMVQS allows domain experts to construct and register continuous queries by combining query-by-navigation and facet refinement over multiple representation paradigms including range and gradient checks and spikes.

3. DEMONSTRATION SCENARIOS

Demo Overview. For the demonstration purpose we selected 20 diagnostic tasks typical for Siemens service centres and expressed these tasks in STARQL and STREAMVQS. Then, we prepared a demo data set of streaming and static data from 950 gas and steam turbines in the time from 2002 to 2011. This data is anonymised in a way that preserves the patterns needed for demo diagnostic tasks. During the demo we will 'play' the streaming data and thus emulate real time streams. Then, we distributed the demo-data in several installations with different number of nodes (VMs) ranging from 1 to 128, where each node has 2 processors and 4GB of main memory. To demonstrate diagnostics results we prepared a dedicated monitoring dashboard for each diagnostic task in the catalog. Dashboards show diagnostics results in real time, as well as statistics on streaming answers, relevant turbines, and other information that is typically required by the service engineers at Siemens . Finally, we deployed OPTIQUE over the Siemens data by bootstrapping ontologies and mappings and then manually post-processing and extending them so that they reach the required quality and contain necessary terms and mappings to cover 20 Siemens diagnostic tasks.

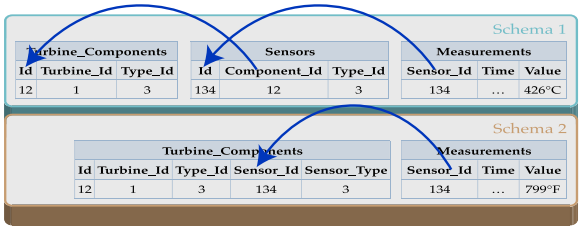
During the demo OPTIQUE will be available in three scenarios:

[S1] *Diagnostics with user's deployment*: the attendees will be able to deploy OPTIQUE over the Siemens data by bootstrapping ontologies and mappings, saving them, and observing and possibly improving them in dedicated editors. Then, they will query their deployed instance with diagnostic tasks either from the Siemens catalog or their own, i.e., they will be able to formulate such tasks in STREAMVQS as parametrised continuous queries and register concrete instances of these tasks over specific data streams.

[S2] *Diagnostics with our deployment*: The attendees will be able to query our preconfigured (high quality) Siemens deployment using diagnostic tasks either from the Siemens catalog and their own.

[S3] *Performance showcase of our deployment*: the attendees will be able to run various tests over our deployment using one of 128 preconfigured Siemens distributed environments and one of 10 test sets of queries. While running the tests they will monitor the throughput and progress of parallel query execution processes.

Use Case. We now illustrate the three scenarios above on a use case inspired by Siemens. Assume that the relational data about turbines that we want to access via an ontology is stored in two alternative relational schemata:



Schema 1 contains the following tables and their attributes:

(i) `Turbine_Components`: This table is used to store *static* information about turbines' *components* and some relevant metadata. The attribute `Id` is the component's unique identifier, the attribute `Turbine_Id` identifies the turbine to which this component belongs to, and the attribute `Type_Id` identifies the components type, for example a component of type 3 is a burner tip.

(ii) `Sensors`: This table is used to store *static* information about *sensors*. The attribute `Id` is the sensor's unique identifier, the attribute `Component_Id` is used to determine the component that uses this sensor, and the attribute `Type_Id` is used to identify sensor's type, i.e. a sensor of type 1 measures temperature.

(iii) `Measurements`: This table is used to store *archived streaming* sensor readings. The attribute `Sensor_Id` identifies the sensor that took the measurement, the attribute `Time` contains the temporal index of each measurement, and the attribute `Value` the actual value that was assessed.

Figure 1: Monitoring dashboards

Schema 2 is designed to store the same information as Schema 1 with the difference that `Turbine_Components` and `Sensors` of Schema 1 are merged into one table `Turbine_Components`. Also the temperature in `Measurements` is measured in Fahrenheit and not Celsius degrees. Note that the tables `Measurements` of Schemata 1 and 2 correspond to the archived parts of streams, while their live parts can be accessed as the EXASTREAM's virtual table `Live_Measurements`. As a rule of thumb, we will assume that the prefix '`Live_`' is used to differentiate between the live and the archived part of each stream and that the live and archived part are described by identical attributes.

Consider a Siemens inspired diagnostics task:

> *'Detect a real-time fault in a burner tip turbine component caused by a temperature increase within 10 sec'.*

Finally, consider fragments of Siemens ontology [8]: two classes

`sie:BurnerTip` and `sie:TemperatureSensor`,

as well as an object and a data property:

`sie:monitoredBy` and `sie:hasValue`,

where the prefix `sie` stands for the URI of the Siemens ontology.

We will now illustrate the three scenarios above using the schemata, diagnostic task and the ontology above.

Scenario S1: For this scenario, by using BOOTOX the attendees will automatically create their ontologies and mappings from the schemata above. Then, they will modify them appropriately in order to provide homogeneous access on the two different data sources. Thus, differences between the two schemata will be hidden by the ontological terms and *OBSSDI* mappings. In particular, the result of bootstrapping will contain the following mapping:

`sie` : `TemperatureSensor`$(Id) \leftarrow$

`SELECT` *Id* `FROM` *Sensors* `WHERE` $Type_{Id} = 3$ `UNION`

`SELECT` *Id* `FROM` *Turbine_Components* `WHERE` *Sensor_Type* $= 3$

Scenario S2: For this scenario, we will explain how the example diagnostics task above can be expressed over the Siemens ontology whose fragments we presented earlier. In Figure 2 (left) the task is expressed as STARQL, in Figure 2 (right) it is visualised in STREAMVQS, and in Figure 1 there are results of task's evaluation visualised in OPTIQUE dashboards. An output stream `S_out`

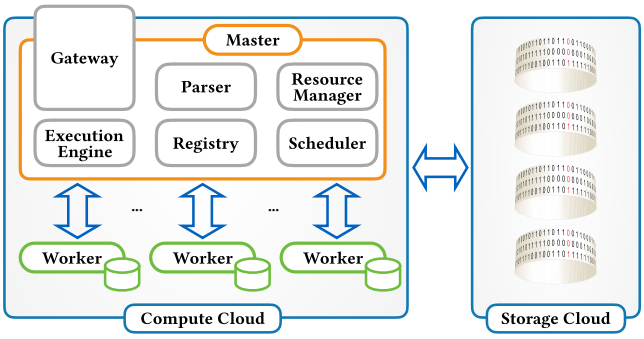
```
CREATE STREAM  Str_out AS
CONSTRUCT      GRAPH NOW { ?c2 rdf:type :MonInc }
FROM           STREAM Str_Msmt [NOW-"PT10S"^^xsd:duration, NOW]->
                              "PT1S"^^xsd:duration,
               STATIC DATA <http://www.optique-project.eu/siemens/
Static>,
               ONTOLOGY <http://www.optique-project.eu/siemens/
Ontology>
USING          PULSE WITH START = "00:10:00CET", FREQUENCY = "1S"
WHERE          {?c1 a sie:BurnerTip. ?c2 a sie:TemperatureSensor.
                ?c1 sie:monitoredBy ?c2.}
SEQUENCE BY    StandardSequencing AS seq
HAVING         MONOTONIC.HAVING(seq, ?c2, sie:hasValue)
```

```
CREATE AGGREGATE MONOTONIC.HAVING ($seq, $var, $attr) AS
       HAVING EXISTS ?k IN $seq: GRAPH ?k {$var sie:showsFault "true"}
       AND FORALL ?i, ?j  IN $seq:
           IF ( ?i < ?j < ?k  AND GRAPH ?i {$var $attr ?x}
           AND GRAPH ?j {$var $attr ?y}) THEN ?x < ?y
```

Figure 2: Example monitoring task in STARQL (on the left) and STREAMVQS (on the right)

is defined by the following language constructs: The CONSTRUCT specifies the format of the output stream, here instantiated by RDF triples asserting that there was a monotonic increase. The FROM clause specifies the resources on which the query is evaluated: the ONTOLOGY, STATIC DATA, and input STREAM(s), for which a window operator is specified with window range (here 10 seconds) and with slide (here 1 second). The PULSE declaration specifies the output frequency. In the WHERE clause bindings for sensors (attached to the turbine) are chosen. For every binding, the relevant condition of the diagnostic task is tested on the window contents. Here this condition is abbreviated by MONOTONIC.HAVING(?c, sie:hasValue) using a macro that is defined at the bottom of Figure 2 in an AGGREGATE declaration. In words, the conditions asks whether there is some state ?k in the window s.t. the sensor shows a failure message at ?k and s.t. for all states before ?k the attribute value ?attr (in the example instantiated by sie:hasValue) is monotonically increasing.

Scenario S3: For this scenario, the corresponding STARQL queries are transformed to the appropriate SQL$^{(+)}$ format and executed in a distributed EXAREME environment. EXASTREAM supports *parallelism* by distributing processing across different nodes in a distributed environment. Consider the architecture of EXASTREAM:



Queries are registered through the Asynchronous Gateway Server. Each registered query passes through the EXAREME parser and then is fed to the Scheduler module. The Scheduler places stream and relational operators on worker nodes based on the node's load. These operators are executed by a Stream Engine instance running on each node. In order to display the importance of parallelism the attendees will be able to run the aforementioned queries using one to 128 nodes. The users will see that OPTIQUE is able to process up to $1,024$ complex Siemens diagnostic tasks in real time with a throughput of up to $3,6 \times 10^6$ tuples/sec by executing the tasks in parallel in a highly distributed environment with up to 128 nodes.

Acknowledgements. This research has been partially supported by the EU project Optique (FP7-IP-318338), the Royal Society, the EPSRC grants Score!, DBonto, and MaSI^3.

4. REFERENCES

[1] A. Arasu, S. Babu, and J. Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. In: *VLDBJ* (2006).

[2] 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. In: *Sem. Web. Journal* (2015).

[3] A. Doan, A. Y. Halevy, and Z. G. Ives. *Principles of Data Integration*. Morgan Kaufmann, 2012.

[4] E. Kharlamov, S. Brandt, E. Jimenez-Ruiz, Y. Kotidis, S. Lamparter, T. Mailis, C. Neuenstadt, Ö. Özçep, C. Pinkel, C. Svingos, D. Zheleznyakov, I. Horrocks, Y. Ioannidis and R. Möller. Ontology-Based Integration of Streaming and Static Relational Data with Optique. In: *SIGMOD demo* (2016).

[5] N. Giatrakos, Y. Kotidis, A. Deligiannakis, V. Vassalos, and Y. Theodoridis. In-network approximate computation of outliers with quality guarantees. In: *Information Systems* 38.8 (2013).

[6] E. Jiménez-Ruiz, E. Kharlamov, D. Zheleznyakov, I. Horrocks, C. Pinkel, M. G. S. veland, E. Thorstensen, and J. Mora. BootOX: Practical Mapping of RDBs to OWL 2. In: *ISWC*. 2015.

[7] E. Kharlamov et al. Enabling Ontology Based Access at an Oil and Gas Company Statoil. In: *ISWC*. 2015.

[8] E. Kharlamov et al. How Semantic Technologies Can Enhance Data Access at Siemens Energy. In: *ISWC*. 2014.

[9] E. Kharlamov et al. Optique: Ontology-Based Data Access Platform. In: *ISWC Posters & Demos*. 2015.

[10] E. Kharlamov et al. Optique: Towards OBDA Systems for Industry. In: *ESWC (Selected Papers)*. 2013.

[11] H. Kllapi, P. Sakkos, A. Delis, D. Gunopulos, and Y. Ioannidis. Elastic Processing of Analytical Query Workloads on IaaS Clouds. In: *arXiv preprint arXiv:1501.01070* (2015).

[12] Özgür Özçep, R. Möller, and C. Neuenstadt. A Stream-Temporal Query Language for Ontology Based Data Access. In: *KI*. 2014.

[13] C. Pinkel, C. Binnig, E. Jiménez-Ruiz, W. May, D. Ritze, M. G. Skjæveland, A. Solimando, and E. Kharlamov. RODI: A Benchmark for Automatic Mapping Generation in Relational-to-Ontology Data Integration. In: *ESWC*. 2015.

[14] A. Soylu, M. Giese, E. Jiménez-Ruiz, E. Kharlamov, D. Zheleznyakov, and I. Horrocks. OptiqueVQS: towards an ontology-based visual query system for big data. In: *MEDES*. 2013.

[15] A. Soylu, E. Kharlamov, D. Zheleznyakov, E. Jimenez-Ruiz, M. Giese, and I. Horrocks. Ontology-based Visual Query Formulation: An Industry Experience. In: *ISVC*. 2015.

[16] M. M. Tsangaris et al. Dataflow Processing and Optimization on Grid and Cloud Infrastructures. In: *IEEE Data Eng. Bull.* 32.1 (2009).