Research online Publishing
www.ronpub.com

Open Access

*Open Journal of Databases (OJDB)*
*Volume X, Issue X, 20XX*
http://www.ronpub.com/ojdb
**ISSN 2199-3459**

# Ontology-based Data Access to Big Data

Simon Schiff, Ralf Möller, Özgür L. Özçep

Institute of Information Systems (IFIS), University of Lübeck, Germany
simon.schiff@student.uni-luebeck.de, {moeller,oezcep}@ifis.uni-luebeck.de

## Abstract

*Recent approaches to ontology-based data access (OBDA) have extended the focus from relational database systems to other types of backends such as cluster frameworks in order to cope with the four Vs associated with big data: volume, veracity, variety and velocity (stream processing). The abstraction that an ontology provides is a benefit from the enduser point of view, but it represents a challenge for developers because high-level queries must be transformed into queries executable on the backend level. In this paper we discuss and evaluate an OBDA system that uses STARQL (Streaming and Temporal ontology Access with a Reasoning-based Query Language), as a high-level query language to access data stored in a SPARK cluster framework. The development of the STARQL-SPARK engine show that there is a need to provide a homogeneous interface to access both, static, and temporal as well as streaming data because, usually, cluster frameworks lack such an interface. The experimental evaluations show that building a scalable OBDA system that runs with SPARK is more than plug-and-play as one needs to know quite well the data formats and the data organisation in the cluster framework.*

## Type of Paper and Keywords

Regular research paper: *streams, OBDA, big data, RDF, cluster framework, SPARK*

## 1 Introduction

The information processing paradigm of ontology-based data access (OBDA) [11] has gained much attention in research groups working on description logics, the semantic web, Datalog, and database systems. But it has become of interest also for the industry [17], mainly due to recent efforts of extending OBDA for handling temporal data [6, 3] and stream data [13, 8, 28, 26, 17] as well as efforts of addressing the needs for enabling statistical analytics: aggregation on concrete domains, temporal operators, and operators for time-series analysis etc. [16].

In an OBDA system, different components have to be set up, fined-tuned, and co-ordinated in order to enable robust and scalable query answering: A query-engine which allows formulating ontology-level queries; a reformulation engine, which rewrites ontology-level queries into queries covering the entailments of the tbox; an unfolding mechanism that unfolds the queries into queries of the backend data sources, and, finally, the backend sources which contain the data.

Whereas in the early days of OBDA, the backend sources were mainly simple relational database systems, recent theoretical and practical developments on distributed storage systems and their extensive use in industry, in particular for statistical analytics on big data, have also raised interest in using cluster frameworks as potential backends in an OBDA system. As of now, a lot of cluster frameworks and data stream management systems for processing streaming and static data have been established. These provide APIs to programming languages such as Java, Scala, Python but sometimes also to declarative query languages such as SQL. However, not all cluster frameworks are appropriate backends

for an OBDA system with analytics. Because of this there are only few publications dealing with OBDA for non-relational DBs, even fewer systems using non-relational (cluster) frameworks, and actually no OBDA system working with cluster frameworks supporting real-time stream processing.

One of the current cluster frameworks that has attracted much attention is the open source Apache framework SPARK[1]. It is mainly intended for batch processing big static data and comes with various extensions and APIs (in particular an SQL API [2]) as well as useful libraries such as a machine learning library. Recently added extensions of SPARK (such as SPARKStream and SPARKStructuredStreaming) are intended for designing systems for processing real-time streams.

In this paper, we present our insights in designing and experimentally evaluating an OBDA system that uses SPARK as a backend system and the query language STARQL [23, 26, 27, 22] as ontology-level query language. We built a small prototype testing SPARK as a potential backend for the STARQL query engine based on the SPARK SQL API and evaluated it with sensor-measurement data. The main scenarios were real-time (continuous) querying and historical querying. In historical querying one accesses historical, aka temporal data, in a sequential manner from the backend source (here SPARK). Historical querying can be used for the purpose of reactive diagnostics where real-time scenarios are reproduced by simulating a stream of data read from the historical DB in order to diagnose potential causes of faulty or erroneous behavior of monitored systems. A detailed description of the results can be found in the project deliverable 5.4 [25]. The software as well as the underlying data are publicly available[2].

The main insights are the following: 1. It means only moderate efforts to adapt an OBDA engine that works with relational DBs or relational data stream management systems to other backends if these provide a robust SQL API. More concretely: the STARQL OBDA engine developed in the OPTIQUE project[3], which works with ExaStream [29, 19] as backend, and the standalone STARQL prototype working with PostGreSQL as backend were easily adapted to work with SPARK as backend. 2. The resulting STARQL-SPARK query engine shows similar performance in processing historical data as the STARQL-ExaStream engine developed in the OPTIQUE project and the STARQL-PostGreSQL prototype. Nonetheless, reaching this performance also depends on finding the right configuration parameters when setting up the cluster. Even then, SPARK showed memory leaks which we explain by the fact that intermediate tables are materialized and not maintained as views. 3. The stream processing capabilities of SPARK 2.0.0 and its extensions are either very basic, not fully specified in their semantics or not fully developed yet. In particular, we saw that a stream extension of SPARK, called SPARKStream, offers only very basic means for stream processing. It does not even provide declarative means for specifying window parameters. As it does not allow applying the SQL API, window parameters have to be programmed by hand. The SPARKStructuredStreaming extension on the other hand, offers a new data structure on top of SPARKStream that can be used together with the SPARK SQL API. Hence, SPARKStructuredStreaming is an appropriate streaming backend with a declarative interface. But in its current stadium in SPARK 2.0.0, it lacks still most of the functionality, so that we could not test it with the same parameters as used for the STARQL-Ontop-ExaStream system. All in all one has to deal separately with the access to historical data and the access to real-time data. So it is a real benefit to have an OBDA query language (such as STARQL) with a semantics that works in the same way for historical and streaming data.

[1] http://spark.apache.org/

[2] https://github.com/SimonUzL/STARQL

[3] http://optique-project.eu/

## 2 OBDA WITH STARQL

STARQL (Streaming and Temporal ontology Access with a Reasoning-based Query Language) is a stream-temporal query framework that was implemented as a submodule of the OPTIQUE software platform [14, 16, 17] and in various stand-alone prototypes described in [20, 22]. It extends the paradigm of ontology-based data access OBDA [11] to temporal and streaming data.

The main idea of OBDA query answering is to represent the knowledge of the domain of interest in a declarative knowledge, aka ontology, and access the data via a high-level query that refers to the ontology's vocabulary, aka signature. The non-terminological part of the ontology, called the abox, is a virtual view of the data produced by mapping rules. Formulated in a description logic, the abox can have many different first-order logic (FOL) models that represent the possible worlds for the domain of interest. These can be constrained to intended ones by the so-called tbox, which contains the terminological part of the ontology.

In classical OBDA, query answering w.r.t. the ontology consists mainly of three steps. The ontology-level query is rewritten into a new query in which the consequences of the tbox are compiled into the query. Then, the rewritten query, which is an FOL query, is unfolded w.r.t. the mapping rules into a query of the data source, e.g., a relational database. This query is

```
1  PREFIX : <http://www.siemens.com/Optique/OptiquePattern#>
2  CREATE PULSE pulseA WITH
3  START = "2015-11-21T00:00:00CET"^^XSD:DATETIME
4  FREQUENCY = "PT1M"^^XSD:DURATION
5
6  CREATE STREAM Sout AS
7  CONSTRUCT GRAPH NOW { ?s a :RecentMonInc }
8  FROM STREAM Meas [ NOW - "PT6M"^^XSD:DURATION, NOW ] -> "PT1M"^^XSD:DURATION,
9       STATIC ABOX <http://www.siemens.com/Optique/OptiquePattern/Astatic>,
10      TBOX <http://www.siemens.com/Optique/OptiquePattern/tbox>
11 USING PULSE pulseA
12 WHERE {?s a :TemperatureSensor}
13 SEQUENCE BY StdSeq AS SEQ1
14 HAVING FORALL i, j IN SEQ1 ?x,?y(
15   IF ((GRAPH i { ?s :hasVal ?x } AND GRAPH j { ?s :hasVal ?y }) AND i < j)
16   THEN ?x <= ?y)
```

**Figure 1: STARQL Query Monotonic Increasing**

evaluated and the answers are returned as answers of the original query.

In the following, we illustrate the different OBDA aspects that are implemented in STARQL with a small example query which was also used (in a slightly simpler form) in our experimental evaluation in a measurement scenario as query *MonInc*. Thereby we will recapitulate shortly the main bits of the syntax and semantics of STARQL. Detailed descriptions of the syntax and its denotational semantics can be found in [26, 23, 24].

The STARQL query in Figure 1 formalizes a typical information need: Starting with the 21st of November 2015, output every minute those temperature sensors in the measurement stream `Meas` whose value grew monotonically in the last 6 minutes and declare them as sensors with a recent monotonic increase.

Many keywords and operators in the STARQL query language are borrowed—and hence should be known—from the standard web language SPARQL[4], but there are some specific differences, in particular w.r.t. the `HAVING` clause in conjuction with a sequencing strategy.

Prefix declarations (l. 1) work in the same way as in SPARQL. Streams are created using the keyword `CREATE STREAM`. The stream is given a specific name (here `Sout`) that can be referenced in other STARQL queries. The `CONSTRUCT` operator (l. 7) fixes the required format of the output stream. STARQL uses the named-graph notation of SPARQL for fixing a basic graph pattern (BGP) and for attaching a time expression to it, either `NOW` for the running time as in the `CONSTRUCT` operator, or a state index $i, j$ as in the `HAVING` clause (l. 15).

The resources to which the query refers are specified using the keyword `FROM` (l. 8). Following this keyword one may specify one or more input streams (by names or further stream expressions) and, optionally, URIs references to a tbox and one or more static aboxes. In this example, only one stream is referenced, the input stream named `Meas`. The tbox contains terminological knowledge, in particular, it contains axioms stating that all temperature sensors are sensors and that all burner-tip temperature sensors are temperature sensors. Factual knowledge on the sensors is stored in the (static) aboxes. For example, the abox may contain assertions { `:tcc125 a BttSensor, :tcc125 :attached :c1, c1 :loc assembly1` } stating that there is a burner tip temperature sensor named tcc125 that is attached to some component c1 located at assembly1. There is no explicit statement that tcc125 is a temperature sensor, this can be derived only with the axioms of the tbox—hence rewriting the query is needed in order to capture all relevant answers.

The input streams consist of timestamped RDF tuples (again represented by named-graphs). The measurement stream `Meas` here consists of timestamped BGPs of the form `GRAPH t1 { ?s :hasVal ?y }` stating that `?s` has value `?y` at time `t1`. The input streams can either be materialized RDF streams or, following the classical OBDA approach, virtual RDF streams: They are defined as views via mapping rules on relational streams of the backend system. For example, assuming a relational measurement stream `Measurement (time,sensor,value)` a mapping rule as shown in Figure 2 generates a (virtual) stream of timestamped RDF triples of the mentioned form.

The window operator `[ NOW - "PT6M", NOW ] -> "PT1M"` following the input stream gives snapshots of the stream with the slide of 1 minute and range of 6 minutes (all stream elements within last 6 minutes).

The `WHERE` clause (line 12) specifies the sensors `?s` that the information need asks for, namely temperature

[4] `https://www.w3.org/TR/rdf-sparql-query/`

```
GRAPH t { s :hasVal v } ⟵
  select sensor as s, time as t,
  value as v from Measurement
```

**Figure 2: Example mapping rule**

sensors. It is evaluated against the static abox(es) only. The stream-temporal conditions are specified in the `HAVING` clause (lines 14–16). In this example the condition is the formalization of the monotonic increase of the values. A sequencing method (here the built-in standard sequencing `StdSeq`) maps an input stream to a sequence of aboxes (annotated by states `i, j`) according to a grouping criterion. In standard sequencing all stream elements with the same timestamp are put into the same state mini abox. Testing for conditions at a state is done with the SPARQL sub-graph mechanism. So, e.g., `GRAPH i {?s :hasVal ?x }` (l. 15) asks whether `?s` shows value `?y` at state `i`.

The evolvement of the time `NOW` is specified in the pulse declaration (l. 4). It is meant to describe the times on which data are put into the output stream. The role of the pulse is to synchronize the different input streams, which may have different slides attached to them. In our example, the information need is meant to be applied on historical data, i.e., data stored in a static database with a dedicated time column. Hence one can specify a `START` date (l. 3) from which on to start the streaming. But sometimes the same information need is required on real-time data. In this case, in essence, the same STARQL query can be used by dropping the `START` keyword. In particular STARQL offers the possibility to integrate real-time data with historic data (as described in [15]). Such a homogeneous interface is a real benefit for engineers which aim at sophisticated predictions on real-time data based on recorded streams.

## 3 Apache Spark Clusters

Apache SPARK is a cluster computing framework which has recently gained much interest because it shows scalability and robustness performances in the range of MapReduce [12] (or outperforms it according to [30]) and because it comes with a useful set of APIs, in particular two APIs used in our experimental evaluations: SPARK SQL, which provides an API to relational data with queries written in SQL, and SPARKStream which allows accessing streams from Kafka, Flume, HDFS, TCP ports or the local file system. In the following we sketch the necessary bits of the SPARK architecture and its extensions that are needed to understand our experimental evaluations.

A SPARK cluster consists of one master and many workers that communicate with the master via SSH. Applications on a cluster are initiated by a script. The so-called *driver program*, which is running on the master node, coordinates and manages the process on the workers. It starts the main method of the application program. The driver program requests all available executors via the cluster manager which runs on the workers. Subsequently, the program code is transmitted to the executor and tasks are started. Results of the workers are received back to the driver program. In order to process the data, the executor must have access to a shared file system. In our experiments, we used the Hadoop File System (HDFS) which provides a sophisticated blockwise storage of data on the workers.

Unlike applications that were written for a Hadoop cluster and that use MapReduce, within a SPARK cluster interim results can be kept in main memory. This prevents slow read/write operations from/to the hard disk. Furthermore lost intermediate results can be calculated again in parallel by other nodes in case a worker node fails. SPARK provides an abstraction model called *Resilient Distributed Datasets* (RDD) which hides from the developer potential node failures. An RDD is a very basic data structure divided into partitions. The partitions are distributed to the worker nodes and can be processed in parallel. RDDs can be generated from data stored in a file system or can be the result of applying operations to other RDDs. Those operations are either *transformations* or *actions*. The main difference is that SPARK only remembers transformations in a lineage but does not compute them. Only if an action has to be processed does the cluster become active and starts calculating all transformations up to the action (inclusively). Examples of transformations are *map(f)*, which maps every element $e$ to $f(e)$ in the new RDD, or *filter(f)*, which filters all elements according to a Boolean condition $f$, and many more. Examples of actions are *collect()*, which sends all elements of an RDD to the driver program, or *count()*, which returns the number of elements in an RDD.

The API SPARK SQL uses DataFrames as the abstraction model in the same way SPARK uses RDDs. DataFrames can be regarded as RDDs of row objects. Internally, however, these are stored column wise and the row objects are calculated only if the user wants to access them via the respective Java, Scala or Python API. This storage type is much more compact than that of using Java/Python objects, which is a big advantage for in-memory processing. DataFrames can be obtained from existing RDDs or from various sources. Unlike the RDDs, they have a schema similar to a table in a database. All common SQL data types are sup-

ported, such as Double, Decimal, String, Timestamp and Boolean. Similar to RDDs, DataFrames are calculated only when actions are applied. The resulting optimizations are handled for DataFrames with a special optimizer called Catalyst.

The main abstract data model of the API SPARKStream is a DStream which is defined as a (potentially infinite) sequence of RDDs. A DStream can be built from various resources such as a TCP port, Kafka, Flume or from HDFS. The grouping of elements into a RDD is specified with a time interval. Moreover, SPARKStream provides a window operator with a range (width of window) and a slide (update frequency) parameter.

SPARKStream has several drawbacks. DStreams consists of a sequence of RDDs which are low level data structures. In particular, RDDs do not have schemes associated with them so they are not directly available for SQL processing. Hence, they would have to be transformed to DStreams with a specified schema. Another drawback is that SPARKStream does not handle asynchronous streams. Because of these reasons a new streaming library called SPARKStructuredStreaming was developed. It is part of the SPARK 2.0.0 release and was in alpha stadium when we experimented with it. SPARKStructuredStreaming still relies on DataFrames. But note that DataFrames can be generated not only from static data but also from streaming data. Unfortunately, the set of operations provided for DataFrames that are produced from streams does not cover (yet) all operations for DataFrames that are produced from static data. So, e.g., it is still not possible to join two DataFrames coming from streams. SPARKStructuredStreaming provides a window operator with a range and a slide parameter. But now the contents of the window operator are determined by the timestamps of the data elements and not by their arrival order.

## 4 STARQL-SPARK ENGINE: IMPLEMENTATION & TESTS

We implemented a prototypical application for a stream-temporal query answering system using STARQL as the query language, Ontop [10] for rewriting (and partly for unfolding) and SPARK 2.0.0 as the backend system. As in the case of the sub-module of the OPTIQUE platform, this software allows answering historical queries as well as continuous queries over realtime streams.

All tests were conducted with 9 virtual machines (VMs) where one was the master and all others were workers. The master runs on a PowerEdge R530 server which has two Intel Xeon E5-2620 v3 processors 2,4GHz with 6 Core / 12 threads and 64 GB DDR4-SDRAM. 8 worker VMs are run on a PowerEdge C6320 with four data nodes. The data nodes have 2 Intel Xeon E5-2620 v3 processors 2,4GHz, 6 Core / 12 threads and 32 GB DDR4-SDRAM, resp. On all data nodes VMWare ESXi 6.0 is run. The ESXi is booted by SD (R530), SSD (C6320), resp. Every data node may use 2TB (2x2TB as RAID 1) for virtual data file systems (VMFS). The RAID controller are Dell PERC H330. Additionally, every VM may access 1 TB storage as RAID 0. The data nodes are connected via an 10 Gbit ethernet to the server. As switch a Netgear XS708E is used. All VMs use VLAN with MTU 9000. The master has 8 cores and 8 GB ram. Each worker VM has 4 cores and 8 GB ram. On every data node two VMs are running. For the tests we used the Hadoop File System. Though replication is possible in Hadoop, for our tests we did not replicate data on the nodes in order to save space. This caused no problem as no node was down in the tests.

Within the tests we used four different STARQL queries three of which are linear and one is quadratic. The listings for the queries can be found on the website of this engine[5]. Here we describe them shortly:

- *Filter*: The linear threshold query asks for all sensors with name TC258 and temperature value smaller than 999.
- *Max*: The maximum query asks for the current maximum value and all maximum values within the last 5 minutes for all sensors.
- *TempPeaks*: The linear peak query asks for all temperature peaks in all sensors.
- *MonInc*: The quadratic monotonic increase query asks for all sensors showing a monotonic increase of the temperature.

For testing historical processing we used a PostGreSQL DB with a simple schema given in Fig. 4.

The sensor data for the `Measurement` table were generated randomly with a java method. We produced four different sized CSV files in plain ASCII text with 17 sensors and temperature values between $4°C$ and $126°C$ for every minute. As in other OBDA based systems one has to specify next to the data source also mappings and the ontology. These can be found on the accompanying website to this paper. The ontology is in DL-lite and covers a simple hierarchy of sensors and values. The data are read in via a SPARK API from a PostGreSQL DB and are stored in HDFS. For the latter, the so-called Parquet data format with Snappy compression[6] is used. The Snappy compression is tailored towards time minimization and not towards space minimization. Nonetheless, within the tests Snappy was able to compress the data to 25 % of the original size. All data such

[5] https://github.com/SimonUzL/STARQL

[6] https://google.github.io/snappy/

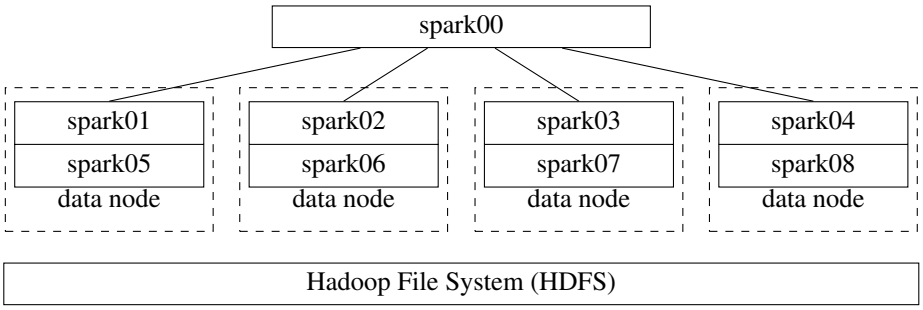



**Figure 3: Spark cluster configuration for tests**

```
Assembly(Id,Name)                    Assemblypart(Id,Name,Part)
Sensor(Id,Assemblypart,Name,Type)    Measurement(Timestamp,Sensor,Value)
```

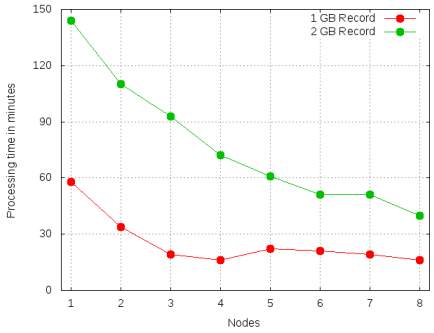
**Figure 4: Schema for Sensor Data**

as those from the PostGreSQL table `Measurement` are registered via a name in a catalog such that they can be referenced within SQL queries. Then, all SQL queries resulting from a transformation of the STARQL queries are executed in a loop. All interim results of the SQL queries are calculated and stored with their name in the catalog. Only for the measurement data a non-sql construct was used: In order to group the data w.r.t. the specified window intervals, we relied on the SPARKStructuredStreaming window described before.

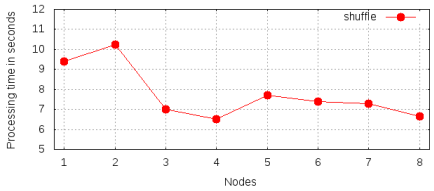
For an adequate comparison of SPARK SQL with PostGreSQL w.r.t. query answering times we set up next to the SPARK cluster configuration mentioned above also a SPARK configuration using only one core of the processor on the master VM because PostGreSQL can use only one core per session. Moreover, PostGreSQL was also installed on the master VM.

For the comparison we used two different files with randomly generated measurements, a 2,7 MB file and a 1 GB file. As can be seen from Table 1, SPARK manages to process the 1 GB data file faster than PostGreSQL does—even if configured to use one core only. Only in case of the *Filter* query, PostGreSQL is faster than SPARK with one core. An explanation for this is that there is an index over the data with which PostGreSQL finds relevant data quite faster than SPARK—SPARK does not provide means of indexing. This latter fact of SPARK being slower than PostGreSQL in answering the *Filter* query holds also for the smaller data file. Even more it is also slower regarding the *TempPeaks* query. If one uses the whole cluster then SPARK in general is slower than PostGreSQL due to the overhead produced by scheduling, starting the tasks, and moving the data around within the cluster.

We tested the scalability of the SPARK cluster by rising the number of worker VMs. For this, SPARK




(a) Scalability w.r.t. STARQL query TempPeak




(b) SQL query with a Group By

**Figure 5: Scalability test results**

was configured such that on every VM one worker with 4 executors was started. Every executor is assigned one of the four available cores. In order to assign also the operating system ram, only 6 GB of the 8 GB was assigned to the worker. Only 1TB hard disk of SPARK was used to store interim results from the ram. So, no two VMs have written jointly on a disk.

As illustrated in Figure 5(a) the query answering times decrease with increasing number of worker VMs up to some limit number. In case of the GB data file this limit is given by 4 nodes. Using more than 4 nodes makes the query answering times even worse—which may be due

| Query | PostGreSQL | SPARK with 1 core | SPARK cluster | data size |
|---|---|---|---|---|
| *Filter* | 12min 33sec | 20min 41sec | 5min 24sec | |
| *MonInc* | 4h 17min 7sec | 1h 31min 34sec | 11min 29sec | |
| *Max* | $>$ 40h | 2h 5min 9sec | 16min 56sec | 1 GB |
| *TempPeaks* | 4h 3min 58sec | 1h 43m 23sec | 10min 13sec | |
| *Filter* | 2sec | 12sec | 17sec | |
| *MonInc* | 34sec | 25sec | 36sec | |
| *Max* | 3min 45s | 26sec | 34sec | 2,7 MB |
| *TempPeaks* | 10sec | 20sec | 27sec | |

**Table 1: Using PostGreSQL vs. SPARK SQL as backend for 1 GB & 2,7 MB data**

to the order in which the worker VMs were chosen. Pairs of workers are running on a data node. During the test the VMs were chosen such that no two of them access the data on the data node at the same time. The pairs of workers have a common hard disk controller and use the same network adapter.

Figure 5(b) shows the results of running a simple SQL query (Fig. 6) on the 1 GB file with `Measurement` data: This query leads to heavy data load in the cluster

```
SELECT      sensor, avg(value),
            max(value), count(value)
FROM        measurement
GROUP BY    sensor
```

**Figure 6: Test SQL query on measurement data**

network. Here we used the same order of choosing the workers as for the experiment from Figure 5(a). Indeed, starting from 4 nodes the response times increase. For larger data files (2 GB say) this is mitigated.

Whereas the tests for historical reasoning reported above were conducted on randomly generated measurement data, the results reported in the following concern a fragment of the large data set which was provided by SIEMENS on a hard disk in the OPTIQUE project. For the tests with SPARK we took a 69 GB file containing anonymized measurement data of 3900 sensors in a range of 6 years. Next to the concrete query answering times for the 69 GB data set, we give in Table 2 rough estimations of the required query answering times interpolated to the 1.5TB data set, the full set of SIEMENS data. We used the four STARQL queries mentioned before.

Considering the query answering times, one can see that there are still opportunities for optimizations of the STARQL + Ontop + SPARK engine. In particular, for the big data set we realized that we could not use the configuration that was used in case of the PostGreSQL backend. Successful query answering without crashes over the 69 GB data set was possible only with a new configuration. A look in the logs revealed that some partitions could not be found. The reason was that some of the nodes were overloaded with processing their jobs so that they could not react to requests of other nodes in time. Because of this fact we configured the SPARK cluster such that every executor is allowed to use only 3 of 4 cores. Furthermore every VM was given 12 GB RAM instead of 8 GB so that the operating system could use 4 GB and rely on one core.

For the queries *Filter*, *MonInc*, and *TempPeaks* we made further configuration changes: The `spark.reducer.maxSizeInFlight` specifies the buffer size of each task. It was decreased from 48m to 4m. The `spark.default.parallelism` parameter determines the possible number of partitions of the results. It was set to 10000.

For the Max query even these adaptations could not prevent memory out of bound exceptions. Hence `spark.default.parallelism` was increased to 30000 and `spark.shuffle.partitions` was set to 3000. With the latter, smaller partitions are kept within the shuffle phase in the working memory.

SPARKStream provides an API to realtime data. As mentioned before, a drawback of SPARKStream is the fact that it supports only RDDs and not DataFrames, which are required in order to apply SPARK SQL. Hence, first, one has to transform RDDs to DataFrames, second, query the DataFrames with SPARK SQL querying and then retransform into RDDs. But as DataFrames have schemes this means that one has to invent a schema before the SPARK application can be run.

In order to test the streaming application, we wrote a small temperature value software that generates every minute some random temperature value where the number of sensors can be chosen by the user of the generator. For all queries the window was specified with a one-minute update. The query answering times for queries *MonInc* and *TempPeaks* are proportional to the number of sensors.

| Query | SPARK with 69 GB | Estimation for SPARK with 1.5 TB |
|---|---|---|
| *Filter* | 5h 27m 43s | 5d |
| *MonInc* | 25h 25m 8s | 23d |
| *Max* | 19h 36m 9s | 18d |
| *TempPeaks* | 26h 51m 34s | 25d |

**Table 2: Query answering times for SIEMENS measurement data**

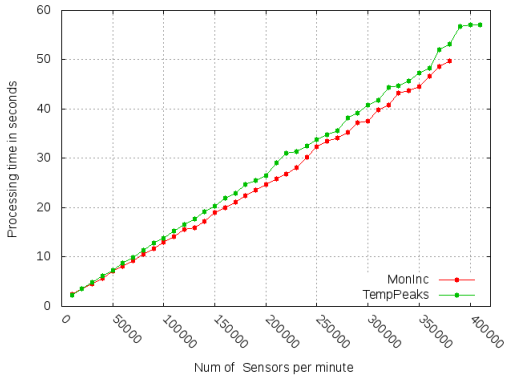



**Figure 7: Query answering times depending on number of sensors**

## 5 Related Work

With its unique features partly illustrated above, namely its sequencing operation, its support of time-series functions, and its specific (window) semantics, previous STARQL engines complemented the collection of state-of-the-art RDF stream processing engines, among them the engines for the languages C-SPARQL [4], CQELS [28], SPARQLStream [8], EP-SPARQL [1], TEF-SPARQL [18] and StreamQR [9]. An overview of all features supported by the STARQL in comparison to other RDF stream engines can be found in [17].

With the new OBDA system on the basis of STARQL and SPARK we provide one of the few OBDA implementations that use a non-relational database system as backend. [7] reports on a OBDA system using the NoSQL MongoDB. [21] and [5] give theoretical considerations on how to handle NoSQL DBs that are based on key-value records. Our system is unique in that it exploits the streaming capabilities of a cluster framework used as backend system.

## 6 Conclusion

This paper described a proof-of-concept implementation of an OBDA system that uses a cluster framework as a backend. As we relied on the SQL API of the SPARK framework, the adaptation of an already present OBDA system is easy. But guaranteeing scalable query answering requires tuning of various parameters of the cluster. And even then, it is not guaranteed to have achieved the possible optimum which would require using native operators on the backend instead of the SQL API. In future work we plan to address a direct compilation of STARQL to native SPARK functions on RDDs. An additional item for future work is to use SPARKStructuredStreaming instead of SPARKStream as backend.

## References

[1] D. Anicic, P. Fodor, S. Rudolph, and N. Stojanovic, "Ep-sparql: a unified language for event processing and stream reasoning," in *WWW*, 2011, pp. 635–644.

[2] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark SQL: Relational data processing in spark," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: ACM, 2015, pp. 1383–1394.

[3] A. Artale, R. Kontchakov, F. Wolter, and M. Zakharyaschev, "Temporal description logic for ontology-based data access," in *IJCAI 2013*, 2013, pp. 711–717.

[4] D. F. Barbieri, D. Braga, S. Ceri, E. D. Valle, and M. Grossniklaus, "C-sparql: a continuous query language for rdf data streams," *Int. J. Semantic Computing*, vol. 4, no. 1, pp. 3–25, 2010.

[5] M. Bienvenu, P. Bourhis, M. Mugnier, S. Tison, and F. Ulliana, "Ontology-mediated query answering for key-value stores," in *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, C. Sierra, Ed. ijcai.org, 2017, pp. 844–851. [Online]. Available: https://doi.org/10.24963/ijcai.2017/117

[6] S. Borgwardt, M. Lippmann, and V. Thost, "Temporal query answering in the description logic DL-Lite," in *FroCos 2013*, ser. LNCS, vol. 8152, 2013, pp. 165–180.

[7] E. Botoeva, D. Calvanese, B. Cogrel, M. Rezk, and G. Xiao, "OBDA beyond relational DBs: A study for MongoDB," in *Proceedings of the 29th International Workshop on Description Logics (DL 2016)*, ser. CEUR Electronic Workshop Proceedings, vol. 1577. CEUR-WS.org, 2016.

[8] J.-P. Calbimonte, O. Corcho, and A. J. G. Gray, "Enabling ontology-based access to streaming data sources," in *Proceedings of the 9th international semantic web conference on The semantic web - Volume Part I*, ser. ISWC'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 96–111. [Online]. Available: http://dl.acm.org/citation.cfm?id=1940281.1940289

[9] J.-P. Calbimonte, J. Mora, and O. Corcho, "Query rewriting in rdf stream processing," in *Proceedings of the 13th International Conference on The Semantic Web. Latest Advances and New Domains - Volume 9678*. Berlin, Heidelberg: Springer-Verlag, 2016, pp. 486–502.

[10] 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*, vol. 8, no. 3, pp. 471–487, 2017. [Online]. Available: https://doi.org/10.3233/SW-160217

[11] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, A. Poggi, M. Rodríguez-Muro, and R. Rosati, "Ontologies and databases: The DL-Lite approach," in *5th Int. Reasoning Web Summer School (RW 2009)*, ser. LNCS. Springer, 2009, vol. 5689, pp. 255–356.

[12] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *OSDI*, 2004, pp. 137–150.

[13] E. Della Valle, S. Ceri, D. Barbieri, D. Braga, and A. Campi, "A first step towards stream reasoning," in *Future Internet – FIS 2008*, ser. LNCS. Springer, 2009, vol. 5468, pp. 72–81.

[14] M. Giese, A. Soylu, G. Vega-Gorgojo, A. Waaler, P. Haase, E. Jiménez-Ruiz, D. Lanti, M. Rezk, G. Xiao, Ö. L. Özçep, and R. Rosati, "Optique: Zooming in on big data," *IEEE Computer*, vol. 48, no. 3, pp. 60–67, 2015. [Online]. Available: http://dx.doi.org/10.1109/MC.2015.82

[15] E. Kharlamov, S. Brandt, E. Jiménez-Ruiz, Y. Kotidis, S. Lamparter, T. Mailis, C. Neuenstadt, Ö. L. Özçep, C. Pinkel, C. Svingos, D. Zheleznyakov, I. Horrocks, Y. E. Ioannidis, and R. Möller, "Ontology-based integration of streaming and static relational data with optique," in *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, F. Özcan, G. Koutrika, and S. Madden, Eds. ACM, 2016, pp. 2109–2112. [Online]. Available: http://doi.acm.org/10.1145/2882903.2899385

[16] E. Kharlamov, Y. Kotidis, T. Mailis, C. Neuenstadt, C. Nikolaou, Ö. L. Özçep, C. Svingos, D. Zheleznyakov, S. Brandt, I. Horrocks, Y. E. Ioannidis, S. Lamparter, and R. Möller, "Towards analytics aware ontology based access to static and streaming data," in *The Semantic Web - ISWC 2016 - 15th International Semantic Web Conference, Kobe, Japan, October 17-21, 2016, Proceedings, Part II*, ser. Lecture Notes in Computer Science, P. T. Groth, E. Simperl, A. J. G. Gray, M. Sabou, M. Krötzsch, F. Lécué, F. Flöck, and Y. Gil, Eds., vol. 9982, 2016, pp. 344–362.

[17] E. Kharlamov, T. Mailis, G. Mehdi, C. Neuenstadt, O. L. Özçep, M. Roshchin, N. Solomakhina, A. Soylu, C. Svingos, S. Brandt, M. Giese, Y. Ioannidis, S. Lamparter, R. Möller, Y. Kotidis, and A. Waaler, "Semantic access to streaming and static data at Siemens," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 44, pp. 54–74, 2017.

[18] J.-U. Kietz, T. Scharrenbach, L. Fischer, M. K. Nguyen, and A. Bernstein, "Tef-sparql: The ddis query-language for time annotated event and fact triple-streams," University of Zurich, Department of Informatics (IFI), Tech. Rep. IFI-2013.07, 2013.

[19] H. Kllapi, P. Sakkos, A. Delis, D. Gunopulos, and Y. Ioannidis, "Elastic processing of analytical query workloads on iaas clouds," *arXiv preprint arXiv:1501.01070*, 2015.

[20] R. Möller, C. Neuenstadt, and Özgür. L. Özçep, "Deliverable D5.2 – OBDA with temporal and stream-oriented queries: Optimization techniques," EU, Deliverable FP7-318338, October 2014.

[21] M. Mugnier, M. Rousset, and F. Ulliana, "Ontology-mediated queries for NOSQL databases," in *Proceedings of the 29th International Workshop on Description Logics, Cape Town, South Africa, April 22-25, 2016.*, ser. CEUR Workshop Proceedings, M. Lenzerini and R. Peñaloza, Eds., vol. 1577. CEUR-WS.org, 2016. [Online]. Available: http://ceur-ws.org/Vol-1577/paper_27.pdf

[22] C. Neuenstadt, R. Möller, and Özgür. L. Özçep, "OBDA for temporal querying and streams with STARQL," in *HiDeSt '15—Proceedings of the*

*First Workshop on High-Level Declarative Stream Processing (co-located with KI 2015)*, ser. CEUR Workshop Proceedings, D. Nicklas and Özgür. L. Özçep, Eds., vol. 1447. CEUR-WS.org, 2015, pp. 70–75.

[23] Ö. L.. Özçep and R. Möller, "Ontology based data access on temporal and streaming data," in *Reasoning Web. Reasoning and the Web in the Big Data Era*, ser. Lecture Notes in Computer Science, M. Koubarakis, G. Stamou, G. Stoilos, I. Horrocks, P. Kolaitis, G. Lausen, and G. Weikum, Eds., vol. 8714., 2014.

[24] Ö. L. Özçep, R. Möller, C. Neuenstadt, D. Zheleznyakov, and E. Kharlamov, "Deliverable D5.1 – a semantics for temporal and stream-based query answering in an OBDA context," EU, Deliverable FP7-318338, October 2013.

[25] Ö. L. Özçep, C. Neuenstadt, and R. Möller, "Deliverable d5.4—optimizations for temporal and continuous query answering and their quantitative evaluation," EU, Deliverable FP7-318338, October 2016.

[26] Özgür. L. Özçep, R. Möller, and C. Neuenstadt, "A stream-temporal query language for ontology based data access," in *KI 2014*, ser. LNCS, vol. 8736. Springer International Publishing Switzerland, 2014, pp. 183–194.

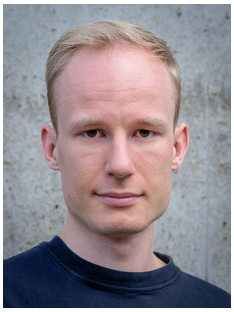
[27] Özgür. L. Özçep, R. Möller, and C. Neuenstadt, "Stream-query compilation with ontologies," in *Poceedings of the 28th Australasian Joint Conference on Artificial Intelligence 2015 (AI 2015)*, ser. LNAI, B. Pfahringer and J. Renz, Eds., vol. 9457. Springer International Publishing, 2015.

[28] D. L. Phuoc, M. Dao-Tran, J. X. Parreira, and M. Hauswirth, "A native and adaptive approach for unified processing of linked streams and linked data," in *The Semantic Web - ISWC 2011 - 10th International Semantic Web Conference, Bonn, Germany, October 23-27, 2011, Proceedings, Part I*, ser. Lecture Notes in Computer Science, L. Aroyo, C. Welty, H. Alani, J. Taylor, A. Bernstein, L. Kagal, N. F. Noy, and E. Blomqvist, Eds., vol. 7031. Springer, 2011, pp. 370–388.

[29] M. M. Tsangaris, G. Kakaletris, H. Kllapi, G. Papanikos, F. Pentaris, P. Polydoras, E. Sitaridi, V. Stoumpos, and Y. E. Ioannidis, "Dataflow processing and optimization on grid and cloud infrastructures." *IEEE Data Eng. Bull.*, vol. 32, no. 1, pp. 67–74, 2009.

[30] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 2–2. [Online]. Available: http://dl.acm.org/citation.cfm?id=2228298.2228301

## AUTHOR BIOGRAPHIES



Simon Schiff is a master student in Computer Science at the Institute of Information Systems (University of Lübeck) mentored by Ralf Möller. The results of his bachelor thesis are the main contributions to this paper. He is preparing his master thesis for optimizing stream query processing within the STARQL engine using an incremental window update algorithm.

Ralf Möller is Full Professor for Computer Science at University of Lübeck and heads the Institute of Information Systems. He was Associate Professor for Computer Science at Hamburg University of Technology from 2003 to 2014. From 2001 to 2003 he was Professor at the University of Applied Sciences in Wedel/Germany. In 1996 he received the degree Dr. rer. nat. from the University of Hamburg and successfully submitted his Habilitation thesis in 2001 also at the University of Hamburg. Prof. Möller was a co-organizer of several international workshops and is the author of numerous workshop and conference papers as well as several book and journal contributions (h-index 33). He served as a reviewer for all major journals and conference in the knowledge representation and reasoning area, and has been PI in numerous EU projects. In the EU FP7 project Optique (www.optique.org), in which abstraction for data access involving ontologies and first-order mapping rules have been investigated in the context of integrating high-pace streaming and high-volume static data, he was the leader of the work package on time and streams.

Özgür Lütfü Özçep is a member of the Information Systems Institute at University of Lübeck since 2014. He worked as a postdoc researcher at Hamburg University of Technology (TUHH) from 2010 to 2014. Before joining TUHH he did his PhD at University of Hamburg as a researcher in the Institute for Knowledge and Language Processing and has taught different course on logics, software programming and knowledge based systems. His PhD thesis dealt with aspects of belief revision, a highly interdisciplinary research topic lying in the intersection of logics, computer science, theory of sciences, and philosophy. After his PhD thesis he contributed to research on combining/extending description logics with other knowledge representation formalisms such as spatial logics—as done in the DFG funded project GeoDL—and to research on ontology-based stream processing—as done in the EU FP7 project Optique. Currently he is habilitating on representation theorems in computer science.