FOrTÉ: A Federated Ontology and Timeseries query Engine

Charbel El Kaed, Matthieu Boujonnier, Stephen Dillon
Schneider Electric: Digital Services Platform
Email: {first.last} @schneider-electric.com

***Abstract*—The adoption of the Internet of things and cloud-connected objects promoted the proliferation of high-level applications aiming to analyze IoT generated data in order to propose value-added services. Such applications distinguish between at least two types of data: contextual information and timeseries. The contextual one, or graph, captures specific information regarding the connected things and their environment, while timeseries provide sampled values over time. Different storage technologies have emerged targeting exclusively graphs and ontologies or massive data which is more suited to timeseries data. However, combining the two worlds in order to provide a semantic scalable data storage seems to be required. We introduce FOrTÉ, a scalable federated query engine capable of bridging the gap between both storage technologies.**

***Index Terms*—Internet-of-Things; Data Lakes; Ontologies**

I. Introduction

The proliferation of connected sensors, devices, and systems accelerated the adoption of massive distributed storage technologies. Such adoption is sustained by the ambition of promoting applications, which will make use of the collected data to drive business opportunities and added values to customers. Business applications are expected to provide several levels of data analysis ranging from monitoring and visualization to prediction through machine learning. In addition, advanced applications are expected to offer facility control and optimization to achieve a larger objective such as an energy management strategy in a smart district or city.

In industrial domains, connected things are of heterogeneous types and range from low-end devices such as sensors and actuators to more capable items such as devices, gateways, and systems. Such diversified connected things embed different resources: CPU, memory, and bandwidth which limit their communication mediums, protocols, and data models. Indeed, the generated information can be abstracted and expressed in data models with human-friendly names such as 'Active-Energy-into-the-Load' or 'Temperature'. However, such naming conventions are human-dependent and syntax-based which lead to inconsistency and incompatibility. For example, a system might expose temperature data as 'Temperature' while another system will refer to it as 'T.' or 'Temp'. In addition, units, scales, and precision bring more challenges. Due to the heterogeneity in various data models, data interpretation and analysis has become seriously challenging.

Semantic Web technology [1] is gaining more popularity over the years. In industrial environments, Semantic Web and ontologies have proved to be a potent solution for issues such as data interoperability [2]. They are particularly useful to represent contextual information in an ontology and help to infer additional knowledge making applications more aware of the context. In addition to the timeseries data (timestamp, value) generated by the things, contextual information is no longer considered as a second class citizen of the data realm. For example, to an energy optimization strategy, the energy consumed by lighting, cooling, heating by zone and by person is more valuable than the total energy consumed by a building.

Different storage technologies have emerged targeting exclusively either Semantic Web and contextual information or massive data which is more fitted to timeseries data. However, combining the two worlds in order to provide a semantic scalable data storage is required for IoT applications.

In this paper, we present FOrTÉ a federated query engine capable of bridging the gap between both storage technologies and relying on the best of both worlds: reasoning from the ontology domain and high scalability from the timeseries domain. The rest of the paper is organized as follows: section II presents our IoT industrial context and the problem statement. Section III describes our proposed solution in detail. Sections IV and V cover the implementation along with the evaluation while section VI reviews the related work. Section VII concludes the paper and outlines future steps.

II. Industrial Context & Problem Statement

The scope of this work is partially described in [3]. It is applied to our facility in North America whose systems include a Building Management System (BMS) and a Power Monitoring System (PM). Our goal is to connect the two on-premise systems to the cloud and be able to combine their data sources to drive better insights for energy management as outlined in Fig. 1.

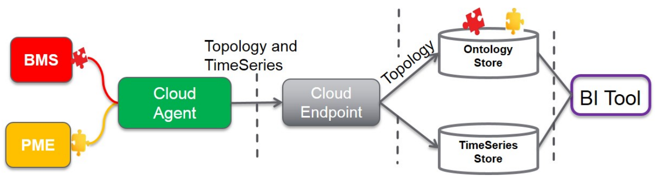
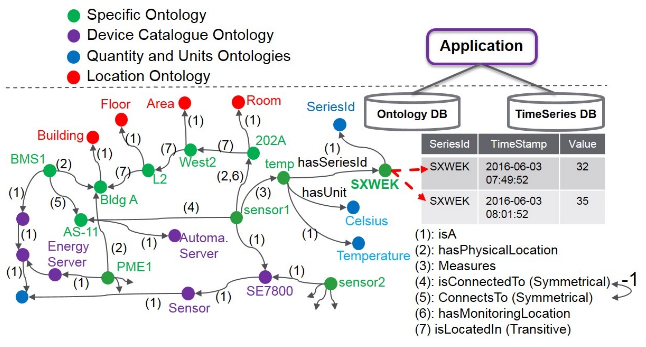


Fig. 1. Overall Architecture

As mentioned in [3], the cloud agent extracts and publishes to the cloud two types of information : topological and timeseries data. The topological data is collected from the Building Management System and the Power Management System then transformed into an ontology as shown in Fig. 2. For example, *sensor1* is an instance of a *SE7800* sensor, connected to the Automation Server *AS-11*. It measures *Temperature* in *Celsius* and it has a physical and monitoring locations, room *202A*. The designated room is located in the *West2* side of the floor *L2* of building *BldgA*. Moreover, *sensor1* has a unique SeriesId: *SXWEK* which uniquely designate its timeseries data.

In addition, the Ontology Web Language [4] allows to declare a property such as *isLocatedIn* as a transitive relation or *connectsTo* and *isConnectedTo* as symmetrical and are inverse of each other. Such properties are exploited by the reasoners during queries. SPARQL [5] a standard query language is used to explore and query data. It is an expressive language, when combined with a reasoner, it allows advanced data exploration, e.g., a request for the list of temperature sensors located in *BldgA* will return *sensor1*. Since sensor1 is located in *202A*, which is transitively connected to the floor and the building, then the returned results will contain sensor1. Also, the BMS (respectively PM) is an instance of *Building Server* (respectively *Energy Server*) is based on a common abstract base class, *Server*. A single query can be used to explore multiple systems due to the abstraction, such as *find all servers with their locations and connected elements along with their measure types and series Id*.



SeriesId	TimeStamp	Value
SXWEK	2016-06-03 07:49:52	32
SXWEK	2016-06-03 08:01:52	35

Fig. 2. Contextual and TimeSeries Data

Timeseries data is sampled and collected over time, therefore, having a low cost and scalable storage is essential. Indeed, applications providing machine learning capabilities are expected to rely on months of data to train their algorithms.

Due to the heterogeneous nature of contextual information and timeseries, the difference in their updating frequencies and the reasoning requirements, we opted for the data architecture and storage as outlined in Fig. 2. The contextual information is persisted in a triple store providing reasoning capabilities, while the timeseries data is inserted in a timeseries store.

Both of the two systems: ontology and timeseries stores have pros and cons. The former offers inference capabilities through reasoners in order to extract new implicit information from what has been explicitly represented. However, such reasoners are not scalable enough to support years of timeseries data. The latter provides scalability, however, it does not support inference capabilities. As shown in Fig. 2, the timeseries data is stored in columns such a SeriesId, timestamp, value or any other parameters changing over time such as data validity, or interval data [6]. The bridge between the semantic representation of the contextual information and the timeseries data is represented by the SeriesId in both worlds.

In our previous work [3], we showed that such data architecture puts a lot of burden on the application or BI tool which aims to join contextual with timeseries data by connecting to two different storage technology. We propose next a solution for the applications to query timeseries and triple stores in a scalable and federated manner.

III. FOrTÉ

We present in this section, our federated query engine, FOrTÉ. First, we overview the requirements which influenced our choices. Then, we go through the supported query language. Last, we detail the architecture and design.

A. Requirements

Based on our industrial context and use cases, we depict some of the requirements to be fullfilled by a federated engine.

(a) **Standard query language**: a federated engine is expected to expose a standard query language for the integration with existing third parties applications and for the ease of adoption by users. SQL and SPARQL [5] are both standards and therefore are potential candidates to be selected as the query language for our engine. SQL is already widely adopted in the relational database domain and gaining adoption in the big data domain with components like Apache Spark[1] which is introduced as a fast and general batch processing engine. It provides a Spark-SQL [7] module for relational processing.

On the other hand, SPARQL is the standard query language in the semantic domain. Many comparative studies depicted the pros and cons of these two languages along with conversion tools from SPARQL-to-SQL [8] or SQL-to-SPARQL [9].

(b) **Standard result format**: in order for the federated engine to be *pluggable* to third parties' applications or visualization tools, the serialization results have to be compliant to a standard result format. SPARQL already supports several formats such as JSON [10], CSV, and TSV [11].

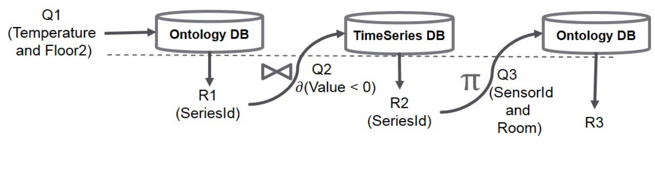


Fig. 3. Multiple Queries Example

(c) **Sub-queries and multiple joins operations**: applications should be able to combine multiple sub-queries in a single query to look up items of interest in the triple store and their corresponding timeseries data. Consider a maintenance application looking to investigate potential malfunctioning temperature sensors on the second floor in order to schedule

[1] spark.apache.org

a maintenance operation. Such request can be divided into the following three sub-queries as shown in Figure. 3. The first query *Q1* requests all the timeseries Ids of the temperature sensors on the second floor. The results of the first query *R1* returns the series id. *Q2* takes the results (R1), the series id, performs a join with the timeseries database, then filters those having negative values. The returned results *R2* represents the timeseries Ids having negative values. *Q3* projects such Ids with the sensor Ids and the room location of the sensors.

(d) **Scalability**: analytics applications require millions of historical records for their algorithms. Thus, the federated engine is expected to return huge amounts of data.

(e) **Reasoning**: the underlying triple store exposes the reasoning feature in its query APIs, thus, the federated query engine must be able to expose such feature and forward the reasoning request to the underlying triple store.

Based on these requirements, we depict in the following the selected query language and the design of our engine.

B. SPARQL Federated Query

In [12], the W3C published a SPARQL Federated Query recommendation allowing data consumers to query data distributed across the Web. This extension enables a query to have several sub-queries which can be directed to several SPARQL endpoints. The federated query processor recieves and combines the returned results.

```
1  SELECT DISTINCT ?sensorId ?Room ?time_Stamp ?temperature{
2   SERVICE <http://stardogserver:5820/
             BOC-XSD/query?reasoning=true> {
3    SELECT DISTINCT ?timeSeriesId ?sensorId ?Room
4     WHERE {
5         ?server a qt:Server.
6         ?server qt:connectsTo+ ?sensor.
7         ?sensor bldgs:hasPoint ?point.
8         ?sensor qt:hasId ?sensorId.
9         ?point qt:hasSeriesId ?timeSeriesId.
10        ?point qt:hasUnitType ?Unit.
11        ?Unit a qt:Celsius.
12        ?point qt:hasMonitoringLocation ?Room.
13        ?Room loc:isLocatedIn ?Floor.
14        ?Room a loc:Room.
15        ?Floor a loc:Floor.
16        FILTER (?Floor = boc:L2)}
17   }
18   SERVICE <http://federated-engine:9090/federated?
                 tenantid=BOC-XSD&db=mssql>{
19    SELECT ?id ?time_Stamp ?temperature
20     WHERE {
21        ?id qt:hasTimeStamp ?time_Stamp.
22        ?id qt:hasValue ?temperature.
23        FILTER(?time_Stamp>'2015-10-08' && ?temperature<0)}
24   }
25   FILTER (?timeSeriesId = ?id)}
```

Listing 1: An Example of a SPARQL Federated Query

The multiple queries (Q1, Q2, Q3) example of Fig. 3, aiming to detect malfunctioning temperature sensors on floor 2, can be written as shown in Listing 1. The SPARQL federated query consists of two sub-queries (Q1) lines 2-17 and (Q2) lines 18-24. The first sub-query (Q1) will be forwarded to the remote SPARQL service endpoint with reasoning feature enabled as depicted in line 2 of Listing 1. This query requests all temperature sensors with Celsius unit located on *L2*. The inference will be applied on the location, the server types, and the connectivity. The sub-query, as depicted in line 3 of Listing 1, selects the timeseries Id of the measure, the sensor Id and the room location of the sensor. The second sub-query, shown in lines 18-24, requests all Ids, timestamps, and negative values after a specific time.

Line 25 of Listing 1 depicts the join operation between the two sub-queries which is the series Id. Line 1 corresponds to (Q3) of Fig. 3, it expresses the projected returned elements: the sensor Id, its room location.

The SPARQL Federated Query [12] does not provide any optimization recommendations to be applied among sub-queries. In Listing 1, the first sub-query returns timeseries Id. However, the second sub-query will return all the timeseries with negative values without any filter on the Ids. Thus, without any optimization strategy, such solution is not well suited to bridge between the IoT graph and timeseries data.

We discuss next our solution relying on an optimised query execution strategy based on query rewriting. Our proposed approach fulfill the requirements depicted in section III-A to federate two different types of data stores in a scalable manner.

C. FOrTÉ's Design

We chose SPARQL as the supported query language for our federated query engine for the following four reasons: first, our industrial context and current architecture depicted in section II already hosts an ontology storage which contains the topology of our building installation along with the systems, devices, and sensors. Second, SPARQL already recommends [12] enabling the federation of multiple queries to be executed on remote distributed endpoints even if it lack of an optimized execution strategy. Third, SPARQL supports several standard results' formats, in particular, CSV and TSV which are preferred by analytics applications and existing BI tools. Fourth and last, with existing popular big data components such as Spark-SQL [7], timeseries stores relying on SQL or NoSQL query interfaces become queryable. In addition, since the timeseries storage contains a limited number of fields such as the SeriesId, timestamp, and value, a translation from SPARQL to another language such as SQL in order to query a timeseries store is feasible [8], [13].

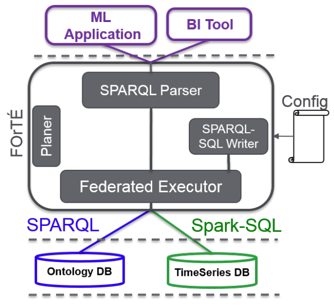


Fig. 4. FOrTÉ Architecture Overview

With SPARQL as its supported query language, FOrTÉ, as shown in Fig. 4, is a query processor which handles a federated SPARQL query from an application, such as Listing 1. It is designed to recognize sub-queries to be executed on remote endpoints with a triple store. In addition, FOrTÉ exposes the timeseries stores as a SPARQL endpoint for applications. The engine's execution plan looks up for the filtering and joins operations in the federated SPARQL query, such as line 25 of Listing 1, and rewrites optimized queries.

In the following, we detail FOrTÉ's components:

Query Planer: handles the flow of execution of a SPARQL federated query according to the order of the sub-queries. It orchestrates the modules of FOrTÉ ensuring the successful execution of the received query and returning the results to the application according to the requested format.

SPARQL Parser: after receiving a SPARQL federated query, such as Listing 1, the parser verifies the query and extracts the three following parts: (a) global select, (b) list of sub-queries and (c) the global filters. These extracted parts are prepared to be optimised and rewritten. The global select expression (a), as shown in Listing 1 line 1, constitutes the final elements to be returned to the application. The list of sub-queries (b) to be forwarded to the triple store and the list of the sub-queries to be rewritten and executed on the timeseries store. The query parser recognizes the timeseries sub-queries based on the reserved keyword *federated* in the Service URL, line 18 of Listing 1. In addition, the *tenantid* parameter specifies the name of the timeseries table to query. Finally, the parser extracts the global filter expression (c) to perform the join operation between sub-queries.

Config: constitutes the configuration parameters of the federated query engine. It exposes several types of information such as the remote endpoint of the timeseries store, along with R2RML [8] configuration. Such attributes will be used in the SPARQL to SQL query translation since it has to match the exact attributes names of the remote timeseries target.

SPARQL-SQL writer: receives SPARQL sub-queries, such as line 19-23 of Listing 1, which are expected to be executed on a timeseries storage. It relies on the configuration parameters when generating SPARQL to SQL queries, more specifically, the timeseries attributes of the timeseries storage. The R2RML [8] configuration file, containing mapping information determine the translation between the queries variables and the timeseries storage attributes. (Seriesid $\equiv$?id, Value $\equiv$?temperature, t $\equiv$?timestamp).

The SPARQL-SQL writer receives additional parameters to be injected such as the list of timeseries Ids from the sub-query executed first on the triple store, lines 3-17, Listing 1. Then, this module generates at least two optimized SQL queries, the first, shown in Listing 2, will be executed on the timeseries store. the second query is the transformation of the SPARQL global select, Listing 1-line 1, and the global filter, Listing 1-line 25, into a SQL query as shown in Listing 3.

```
SELECT SeriesId, Time_Stamp, Value FROM timeseries
WHERE Time_Stamp > '2015-10-08' AND Value < 0
AND SeriesId IN ( '5f05-47ad-d618' ,'42c9-25d2-19da', ... )
```

Listing 2: SPARQL-to-SQL translation: timeseries sub-query

Federated Executor: handles several operations such as the execution of a SPARQL (respectively SQL) query on the ontology (respectively timeseries) store. It also performs join operations between two or more returned results from the sub-queries according to the global filter expression.

```
SELECT Distinct sensorId, Room, time_stamp, value
FROM timeseries WHERE
timeSeriesId = Seriesid
```

Listing 3: SPARQL-to-SQL translation: global select

Next, we discuss a pseudo-sequential execution of the operations carried on by FOrTÉ upon receiving a query.

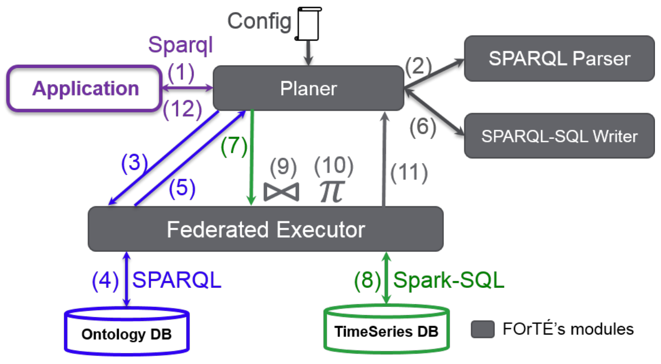


Fig. 5. FOrTÉ Pseudo Sequential Operations

First FOrTÉ receives a federated query such as Listing 1 from the application (1). Second, the planer passes (2) the query to the SPARQL Parser module which extracts the necessary elements as depicted previously. Third, the planer forwards (3) the triple store sub-queries to the Federated Executor which executes (4) them and waits for results. By combining the returned SPARQL results (5), the global filter expression and the timeseries sub-query, lines 19-23, the planer requests (6) from the Writer am optimized SQL query, as shown in Listing 2. Later, the SQL query is passed (7) to the Federated Executor which executes it (8) through the Spark-SQL module on the timeseries storage. The returned timeseries results are then joined (9) with the SPARQL results of step (4). Furthermore, based on the global select operation, a projection is applied (10) to return the requested variables only, as shown in Listing 3. Additionally, the results of (10) are returned (11) to the planer which returns (12) the response format according to the one requested such as CSV [11] or JSON [10].

In the following section, we go through the implementation choices then the performance evaluation.

IV. Implementation

FOrTÉ is implemented as a software component in Java, it connects to any triple store exposing a standard SPARQL endpoint and a timeseries store with a suitable JDBC Spark-SQL compatible connector. The engine exposes a REST service by relying on *nanohttpd* web server.

As an ontology storage, we selected Stardog 4.2[2] with a developer license on a 2 cores, 14 GB memory Linux machine. Our IoT context ontology representing the building installation has 8221 triples. Stardog exposes a REST SPARQL endpoint.

[2]www.stardog.com

Spark-SQL [7] with a set of suitable JDBC Connectors offers high flexibility and allows to connect to various data stores. Therefore, for the timeseries storage, we successfully connected FOrTÉ to a MySQL, Azure SQL, and a MongoDB Database, in addition to a native Hadoop store [14]. Such flexibility offered by Spark-SQL allows our solution to be pluggable to various SQL and NoSQL stores. FOrTÉ expects a configuration file to be loaded at start time containing the remote servers names and credentials along with an R2RML file. Then, through the SPARQL query, the selection of the timeseries database can be set through the tenantId and the type of database, as shown in Line 18 (db=mssql) of Listing 1.

The SPARQL parser, shown in Fig. 4, relies on Antlr4[3] library which takes as input a SPARQL grammar[4] and then generates the required code to verify the queries lexical and syntactic conformity along with an Abstract Syntax Tree (AST). The SPARQL parser relies on the visitor pattern in order to go through the AST tree to extract the timeseries SPARQL sub-query and populate internal structure to be used later for the SPARQL to SQL query conversion. Next, we evaluate FOrTÉ to demonstrate the scalability of our approach.

V. Evaluation

To evaluate FOrTÉ, we looked into available benchmarks such as LUBM [15] and BSBM [16]. However, such benchmarks are not designed to handle timeseries data. One can extend, for example, the LUBM benchmark and add timestamp attributes to the generated data. However, this would also imply changes in the evaluation queries and therefore modifies the whole idea of having a common and replicable benchmark. Thus, we decided to rely on our industrial use case and prepare a reproducible benchmark more suited to the IoT context.

A. *TimeSeries Data Generation*

As mentioned in [3], the Cloud Agent sends, every 15 minutes, 213 records (series Id, timestamp, value) to the cloud. Since, we did not have years of data yet, we implemented a Data Generator which relies on the 213 seriesIds as input and generates a timestamp and a value between $[0..99]$. We created 6 different tables (50m, 150m, 250m, 500m, 750m, 1000m) in Azure SQL database to insert from 50 million to 1 billion timeseries records. The Data Generator fills each table with its correspondent number of records, 50 million records in the 50m table, up to one billion records in the 1000m table.

B. *Three Evaluation Architectures*

These 6 different tables will constitute the common timeseries data for our three evaluation architectures as shown in Fig. 6. We detail next these three architectures: **FOrTÉ**, shown on the left in Fig. 6, is deployed on a single worker node Azure Spark 1.6.1 cluster with a 4 cores (Intel Xeon CPU E5, 2.40GHz) and a 14 GB of RAM. Stardog is installed on a Linux Azure VM with 2 cores and 7 GB of RAM. The topology of our industrial context, with 8221 triples,

[3]www.antlr.org

[4]https://github.com/antlr/grammars-v4

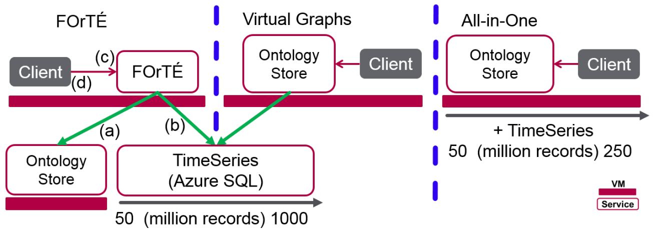


Fig. 6. Evaluation Architectures

is deployed on Stardog 4.2 community edition. The client, *curl* based, is used to query the REST/SPARQL interface of FOrTÉ. The returned records result are saved in a CSV file.

Virtual Graphs, shown in the center of Fig. 6, relies on the R2RML [8] specification which allows triple stores to bind to remote SQL tables in order to formulate queries upon. We deployed Stardog 4.2 with an evaluation license on an 8 cores and a 56 GB Linux Azure VM. Then, we added to Stardog a JDBC Microsoft SQL 4.2 connector. We relied on the virtual graphs import feature with an R2RML file to bind Stardog to the remote Azure SQL timeseries tables. Each R2RML configuration file references one of the 6 tables at a time. The client, *curl* based, is used to query the REST/SPARQL interface of Stardog. Any SPARQL query received by Stardog will be evaluated on the local database and then joined with the remote timeseries data on the Azure SQL. We also deployed Virtuoso 7 with an evaluation license on an equivalent VM.

All-in-One, shown to the right in Fig. 6, consists of inserting the topology 8221 triples and the timeseries data into one database. We deployed Stardog 4.2 with an evaluation license on an 8 cores, a 56GB RAM Linux Azure VM with 1 TB of SSD disk. The client, *curl* based, is used to query the REST/SPARQL interface of Stardog. We also selected Virtuoso 7 with an evaluation license for this architecture and deployed it on an equivalent virtual machine. In order to evaluate this architecture with the same timeseries data sets, we relied on *sqlcmd* to export the 6 tables $[50m..1000m]$ in 6 different CSV files. Once those files exported, we relied on the bulk loading APIs of Stardog and Virtuoso in order to import the timeseries data into one table one by one $[50m, 150m, 250m, 500m, 750m, 1000m]$.

In order to have a stable testing environment and to minimize the network latency and delays, all the virtual machines were deployed in the same virtual network on Azure Cloud. For both Virtual Graphs and All-in-One architectures evaluation, we configured Stardog with 20 GB for the JVM and 32 GB for the Off-heap memory, as recommended by their memory usage documentation regarding the expected number of triples to be loaded. As for Virtuoso, we allocated $2,720,000$ for the Number of buffers and $2,000,000$ for the Max Dirty buffers as recommended by their documentation. Even though, FOrTÉ can connect to a wide variety of stores SQL and NoSQL, our choice for selecting Azure SQL to store timeseries data is motivated by the fact that both architectures, FOrTÉ and Virtual Graphs, are able to query SQL stores.

C. Four Types Of Queries

In order to test the three architectures, we rely on the query shown in Listing 1. The first query (lines 3-17) is expected to return 42 timeseries Ids of all temperature measurements of Floor *L2*. Then, we propose to substitute, lines 19-24, with one of the following sub-queries. Each sub-query is expected to retrieve the data of these 42 timeseries Ids while applying four different filtering and aggregation functions.

```
Select ?id ?time_Stamp ?temperature Where {
 ?id qt:hasTimeStamp ?time_Stamp.
 ?id qt:hasValue ?temperature.
 filter(?temperature < 5).}
```

Listing 4: Sub-query 1 (subq1)

Sub-query 1 (subq1), shown in Listing 4, contains a filter on the values. The number of returned records and time performance are expected to vary based on the number of records of the timeseries table. It is a basic test, expected to pass on each of the three architectures.

```
Select ?id ?time_Stamp ?temperature Where {
 ?id qt:hasTimeStamp ?time_Stamp.
 ?id qt:hasValue ?temperature.
 filter((?temperature < 5)
 && (?time_Stamp < '2014-01-02T12:00:00'^^xsd:dateTime)).}
```

Listing 5: Sub-query 2 (subq2)

In Listing 5, sub-query 2 is depicted where a temporal filter is appended in addition to the filter on values. the number of returned records is expected to remain the same, $1,076,912$. The aim of this query is to evaluate the query time performance when the number of timeseries records vary from 50 million to 1 billion but the result is expected to remain the same.

```
Select ?id (min(?temperature) as ?minVal)
(max(?temperature) as ?maxVal) Where {
 ?id qt:hasTimeStamp ?time_Stamp.
 ?id qt:hasValue ?temperature.}
```

Listing 6: Sub-query 3 (subq3)

Next, we test in Listing 6, aggregation functions such as minimum and maximum by selected timeseries Ids. Thus, the number of returned records is always the same (42). The performance will depend on the size of the timeseries table.

```
Select ?id ?time_Stamp ?temperature Where {
 ?id qt:hasTimeStamp ?time_Stamp.
 ?id qt:hasValue ?temperature.}
```

Listing 7: Sub-query 4 (subq4)

Finally, in sub-query 4, Listing 7, we remove any filter or aggregation function and let the three architectures return all the timeseries data of the 42 timeseries Ids. The aim of this query is to test the load and performance of the three architectures with a large number of returned records.

D. Performance Results

In the following, each of the sub-queries is substituted by lines 19-24 of the SPARQL query shown in Listing 1. We denote by Q1 (respectively Q2, Q3, Q4) the SPARQL query of Listing 1 with subq1 (respectively subq2, subq3, subq4). The queries are executed on the three architectures depicted above with a slight change regarding their supported syntax. For example, on the virtual graph architecture, Stardog allows to invoke a virtual graph through the *graph* keyword instead of *service* in its URL in line 18 of Listing 1. However, on Virtuoso, the keyword *from* followed by the graph URL are used to query a remote virtual graph instead of *service*.

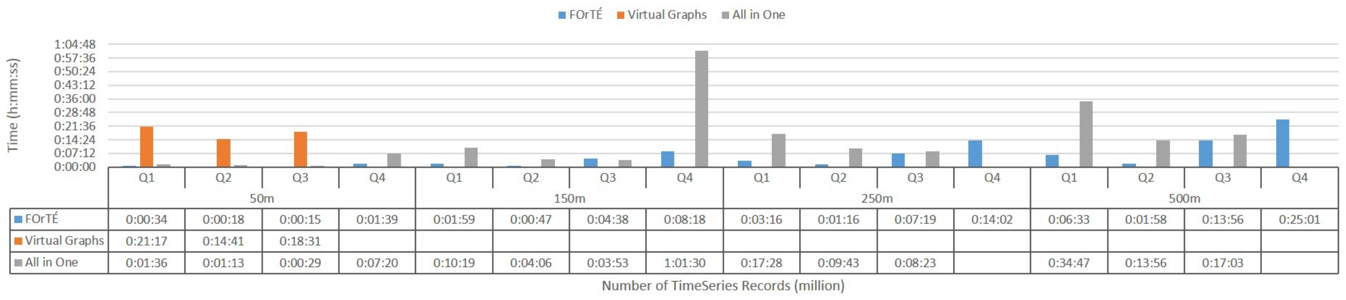
In our evaluation of the Virtual Graphs architecture with Virtuoso, the execution time of Q1 on the timeseries table 50m took 3 hours to complete. Therefore, we decided to eliminate Virtuoso from the Virtual Graphs architecture evaluation. Furthermore, while evaluating the All-in-One architecture, we discovered a limitation in the CSV bulk loading feature of Virtuoso preventing it from loading such large data sets. Therefore, the evaluation below involves Stardog only.

Fig. 7 shows the performance time (end-to-end) of each query executed on each of the three architectures along with the various timeseries table size. All empty blanks indicate the non-completion of the query due to an error in the execution. Fig. 7 shows evaluation up to 500 million records, the maximum number of records that we were able to successfully load in the All-in-One architecture on the given virtual machine.

Q1 is expected to return 2 million records, it executes in 34 seconds by FOrTÉ and in 1 min and 36 sec in the All-in-One architecture. However, on the virtual Graphs architecture, it takes up to 21 minutes to finish. Such slow performance on the Virtual Graphs is due to the queries execution behavior. In fact, the two sub-queries are executed independently, then their results are joined later. In Listing 1, the sub-query in lines 3-17 is executed, it returns 42 timeseries Ids. Then, subq1 in Listing 4 is executed which returns million of records to Stardog having values < 5. Then Stardog applies the joins based on line 25 of Listing 1. Furthermore, Q4 fails to execute resulting with an internal error on the Stardog side. On the other timeseries tables $[150m..1000m]$, the queries fail to execute. We conclude that the Virtual Graphs architecture, with its actual state on Stardog, is not suited for our IoT context.

In the following, we will focus on the comparison between FOrTÉ's and All-in-One architecture. For the Q1 and Q2 execution time, as shown in Fig. 7, FOrTÉ is x4 to x5 times faster than the All-in-One architecture. As for Q3, FOrTÉ and the All-in-One architecture seems to have similar execution time with the aggregation functions (min and max). For the Q4, FOrTÉ is 5x to 6x time faster than the All-in-One architecture, where the query fails to complete when the number of timeseries records is higher than 150 million. The fourth query, Q4, when executed on a 250 million timeseries records table (respectively 500 million), returns 49 million records (respectively 98 million) or 2.99 GB (respectively 4.97 GB) of data, in less than 15 minutes (respectively 25 minutes).

We depicted in Fig. 7 the overall performance of FOrTÉ and its ability to query and join a triple store and a timeseries store efficiently. In order to better understand the performance, we provide in Fig. 8 a more detailed performance execution time of the following operations, shown in Fig. 6 (left):



	50m				150m				250m				500m			
	Q1	Q2	Q3	Q4	Q1	Q2	Q3	Q4	Q1	Q2	Q3	Q4	Q1	Q2	Q3	Q4
FOrTÉ	0:00:34	0:00:18	0:00:15	0:01:39	0:01:59	0:00:47	0:04:38	0:08:18	0:03:16	0:01:16	0:07:19	0:14:02	0:06:33	0:01:58	0:13:56	0:25:01
Virtual Graphs	0:21:17	0:14:41	0:18:31													
All in One	0:01:36	0:01:13	0:00:29	0:07:20	0:10:19	0:04:06	0:03:53	1:01:30	0:17:28	0:09:43	0:08:23		0:34:47	0:13:56	0:17:03	

Fig. 7. Performance Comparison of The Three Architectures

(a) corresponds to the query response time of Stardog with the SPARQL query in Listing 1, lines 3-17. (b) represents the SQL execution time on Azure SQL Server. The SQL query consists of the SPARQL-to-SQL transformation of one of the sub-queries. The execution time of the SQL query is provided by the Query Performance Insight monitoring interface in Azure. (c) measures the overall performance of shuffling and handling retrieved data just before the HTTP server starts sending the records to the client. (d) measures the data transfer alone. Fig. 8 shows that the SPARQL query on Stardog takes 3 seconds to return the 42 series Ids. It also depicts that most of the query time is taken by the SQL query execution. It is also interesting to note that the SQL query time for Q3 and Q4 are almost similar, Q3 requests aggregation (max, min) and returning only 42 records, while Q4 requests the timestamp and value for 42 Ids resulting in million of records. FOrTÉ's operations include table joins, the projection of selected variables along with formatting to CSV [11] can take up to 40% of the overall query execution time in Q4 types queries. Data transfer time is also not to be neglected when dealing with Q4 types of queries returning a dozen of gigabytes corresponding to hundreds of million of records. For instance, Q4 executed on the 1000m returns up to 20% of the table, 197 million records with a size of 11.93 GB.

VI. Related Work

In this section, we briefly overview the state of the art regarding SPARQL query answering by relying on R2RML and triple stores materialization over SQL and NoSQL stores. We divide the related work into two different but related topics.

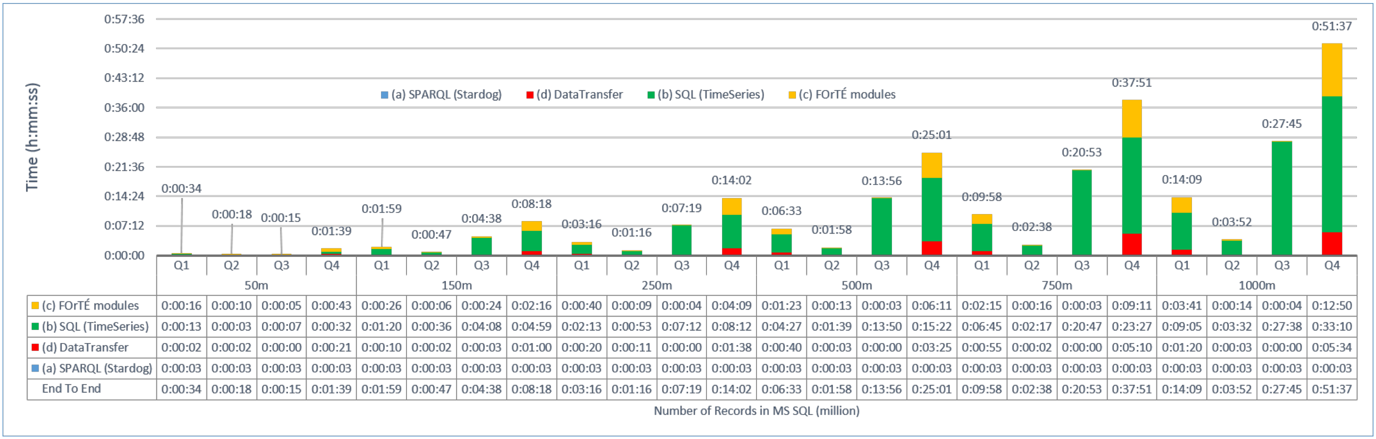
Triple Stores: we came across several proposals pertaining to the execution of SPARQL against SQL and NoSQL storage technology. Husain et al [17] propose a framework for storing and retrieving RDF triples in HDFS utilizing the default Hadoop MapReduce framework. Urbani [18] introduces WebPIE (Web-scale Parallel Inference Engine) to solve the problem of very large scale reasoning on the Web by relying on Map and Reduce operations. Further, DynamiTE [19] offers a parallel system to efficiently computes and incrementally maintain the materialization of a database, which consists of RDF triples. DynamiTE design exploits multi-core hardware and the adoption of data structures that enable fast retrieval of data. Bornea at al [20] propose a storage and query mechanism for RDF which works on top of existing relational representations. They focus on two aspects of storage efficiency, and query evaluation and make a case for using relational data stores en lieu of native triple stores. They examine the challenges of SPARQL query optimization and translation to equivalent SQL queries. Another similar approach such as RDFox [21] and TrinityRDF [22] propose distributed in-memory and distributed reasoning.

All of these approaches rely on rewriting queries translation, indexing strategies, and reasoners on top of the selected underlying stores. In our IoT-oriented approach, we leveraged existing RDF stores along with a federated data architecture. Our main contribution consists in providing a scalable query engine bridging the gap between two storage technologies.

R2RML and Virtual Graphs: R2RML [8] is a W3C recommendation aiming to bridge Relational and RDF worlds by relying on a mapping format. Triple stores vendors such as Stardog and Virtuoso support virtual graphs [5]. Perry *et al* [23] propose SPARQL-ST, an extension of SPARQL for complex spatiotemporal queries. Even though, they present a formal syntax and semantics for SPARQL-ST, however, pushing such extension to the W3C would take time and effort, besides it will violate one of our requirements to expose a standard query language. Michel et al [24] describe xR2RML, a language for expressing customized mappings from various types of databases (XML, object-oriented, NoSQL) to RDF datasets. R2RML addresses the mapping of relational databases to RDF. RML extends R2RML to address the mapping of heterogeneous data formats (CSV, JSON) to RDF.

Such approach seems applicable on a relatively small amount of data stored in a remote RDBMS as shown previously in our evaluation. Moreover, joining data between local RDF stores and remote RDBMS shows to perform very poorly due to the lack of an overall optimized query execution strategy. In our approach, we leverage the standard SPARQL federated queries feature [12] as the basis of our overall query planning and optimization strategy.

[5]http://docs.stardog.com



	50m				150m				250m			
	Q1	Q2	Q3	Q4	Q1	Q2	Q3	Q4	Q1	Q2	Q3	Q4
🟨 (c) FOrTÉ modules	0:00:16	0:00:10	0:00:05	0:00:43	0:00:26	0:00:06	0:00:24	0:02:16	0:00:40	0:00:09	0:00:04	0:04:09
🟩 (b) SQL (TimeSeries)	0:00:13	0:00:03	0:00:07	0:00:32	0:01:20	0:00:36	0:04:08	0:04:59	0:02:13	0:00:53	0:07:12	0:08:12
🟥 (d) DataTransfer	0:00:02	0:00:02	0:00:00	0:00:21	0:00:10	0:00:02	0:00:03	0:01:00	0:00:20	0:00:11	0:00:00	0:01:38
🟦 (a) SPARQL (Stardog)	0:00:03	0:00:03	0:00:03	0:00:03	0:00:03	0:00:03	0:00:03	0:00:03	0:00:03	0:00:03	0:00:03	0:00:03
End To End	0:00:34	0:00:18	0:00:15	0:01:39	0:01:59	0:00:47	0:04:38	0:08:18	0:03:16	0:01:16	0:07:19	0:14:02

	500m				750m				1000m			
	Q1	Q2	Q3	Q4	Q1	Q2	Q3	Q4	Q1	Q2	Q3	Q4
🟨 (c) FOrTÉ modules	0:01:23	0:00:13	0:00:03	0:06:11	0:02:15	0:00:16	0:00:03	0:09:11	0:03:41	0:00:14	0:00:04	0:12:50
🟩 (b) SQL (TimeSeries)	0:04:27	0:01:39	0:13:50	0:15:22	0:06:45	0:02:17	0:20:47	0:23:27	0:09:05	0:03:32	0:27:38	0:33:10
🟥 (d) DataTransfer	0:00:40	0:00:03	0:00:00	0:03:25	0:00:55	0:00:02	0:00:00	0:05:10	0:01:20	0:00:03	0:00:00	0:05:34
🟦 (a) SPARQL (Stardog)	0:00:03	0:00:03	0:00:03	0:00:03	0:00:03	0:00:03	0:00:03	0:00:03	0:00:03	0:00:03	0:00:03	0:00:03
End To End	0:06:33	0:01:58	0:13:56	0:25:01	0:09:58	0:02:38	0:20:53	0:37:51	0:14:09	0:03:52	0:27:45	0:51:37

Fig. 8. Performance Evaluation of FOrTÉ

VII. Conclusion and Future work

In this paper, we propose FOrTÉ, a federated query engine along with a data architecture capable of bridging the gap between the ontology and the timeseries domains in an IoT context. FOrTÉ exposes a SPARQL federated query endpoint and relies on an optimized overall query planning and execution strategy, making it compliant to our list of requirements. For the timeseries storage, FOrTÉ is based on Apache Spark-SQL a parallel query engine which, when combined with the adequate adapter, can query a multitude of SQL and NoSQL stores making our solution pluggable to disparate data sources. In our evaluation, we showed that FOrTÉ's performance is better than existing native RDF stores and Virtual Graphs. FOrTÉ is up to x5 times faster than native RDF stores and up to x21 faster than Virtual Graphs, and can handle larger volume of data up to 1 billion timeseries records. Such performance removes the burden of join and aggregation from third parties applications and delegates them to FOrTÉ thanks to its REST interface and its supported standard results formats. In this work, we selected a SQL store in order to make the evaluation more comparable, however, other existing stores might constitute better candidates for timeseries that we are willing to evaluate in our future work.

References

[1] J. Ye *et al.*, "Semantic web technologies in pervasive computing: A survey and research roadmap," *Pervasive and Mobile Computing*, 2015.

[2] C. E. Kaed, Y. Denneulin, and F. G. Ottogalli, "Dynamic service adaptation for plug and play device interoperability," in *Proceedings of the 7th International Conference on Network and Services Management*. International Federation for Information Processing, 2011, pp. 46–55.

[3] C. E. Kaed *et al.*, "Building management insights driven by a multi-system semantic representation approach," in *IEEE WFIoT*, 2016.

[4] Bechhofer *et al.*, "Owl," http://www.w3.org/TR/owl-features/, 2004.

[5] W3C, "Sparql query language for rdf," http://www.w3.org/TR/rdf-sparql-query/, 2004, sPARQL Query Language for RDF.

[6] C. A. Henson, H. Neuhaus, A. P. Sheth, K. Thirunarayan, and R. Buyya, "An ontological representation of time series observations on the semantic sensor web," in *1st Workshop on the Semantic Sensor Web*, 2009.

[7] M. Armbrust *et al.*, "Spark sql: Relational data processing in spark," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. ACM, 2015, pp. 1383–1394.

[8] D. Souripriya, S. Seema, and C. Richard, "R2rml: Rdb to rdf mapping language, w3c recommendation," https://www.w3.org/TR/r2rml/, 2012.

[9] J. Rachapalli, V. Khadilkar, M. Kantarcioglu, and B. Thuraisingham, "Retro: A framework for semantics preserving sql-to-sparql translation," in *Workshop on Knowledge evolution and ontology dynamics, co-located with the 10th International Semantic Web Conference*, 2011.

[10] W3C, "Serializing sparql query results in json)," https://www.w3.org/TR/rdf-sparql-json-res/, 2013.

[11] ——, "Sparql 1.1 query results csv and tsv formats," https://www.w3.org/TR/sparql11-results-csv-tsv/, 2013.

[12] ——, "Sparql 1.1 federated query," https://www.w3.org/TR/sparql11-federated-query/, 2013.

[13] E. Prud'hommeaux and A. Bertails, "A mapping of sparql onto conventional sql," https://www.w3.org/2008/07/MappingRules/SpringerTemplate/rdb2rdf.pdf.

[14] Hortonworks, "A modern data architecture with apache hadoop the journey to a data lake," http://info.hortonworks.com/rs/h2source/images/Hadoop-Data-Lake-white-paper.pdf, 2014.

[15] Y. Guo *et al.*, "An evaluation of knowledge base systems for large owl datasets," in *In International Semantic Web Conference*, 2004.

[16] C. Bizer and A. Schultz, "The berlin sparql benchmark," *International Journal On Semantic Web and Information Systems*, 2009.

[17] K. T. Husain, Doshi, "Storage and retrieval of large rdf graph using hadoop and mapreduce," in *CloudCom '09 Proceedings of the 1st International Conference on Cloud Computing*. ACM, 2009.

[18] M. H. B. Urbani, Kotoulas, "Owl reasoning with webpie: calculating the closure of 100 billion triples," in *ESWC'10 Proceedings of the 7th international conference on The Semantic Web: research and Applications - Volume Part I*. ACM, 2010, pp. 213–227.

[19] J. H. B. Urbani, Margara, "Parallel materialization of dynamic rdf data," in *12th International Semantic Web Conference*, 2013.

[20] Bornea, Dolby, Kementsietsidis, Srinivas, Dantressangle, Udrea, and Bhattacharjee, "Building an efficient rdf store over a relational database," in *SIGMOD '13 Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2013.

[21] Y. e. a. Nenov, *RDFox: A Highly-Scalable RDF Store*. Springer International Publishing, 2015, pp. 3–20.

[22] W. S. W. Zeng, Yang, "A distributed graph engine for web scale rdf data," in *Proceedings of the VLDB Endowment*. ACM, 2013.

[23] M. Perry, P. Jain, and A. P. Sheth, *SPARQL-ST: Extending SPARQL to Support Spatiotemporal Queries*. Boston, MA: Springer US, 2011.

[24] C. F.-Z. J. M. Franck Michel, Loc Djimenou, "xr2rml: Relational and non-relational databases to rdf mapping language," https://hal.archives-ouvertes.fr/hal-01066663v3, 2015.