SPARQL-to-SQL on Internet of Things Databases and Streams

Eugene Siow(✉), Thanassis Tiropanis, and Wendy Hall

Electronics and Computer Science, University of Southampton, Southampton, UK
{eugene.siow,t.tiropanis,wh}@soton.ac.uk

Abstract. To realise a semantic Web of Things, the challenge of achieving efficient Resource Description Format (RDF) storage and SPARQL query performance on Internet of Things (IoT) devices with limited resources has to be addressed. State-of-the-art SPARQL-to-SQL engines have been shown to outperform RDF stores on some benchmarks. In this paper, we describe an optimisation to the SPARQL-to-SQL approach, based on a study of time-series IoT data structures, that employs metadata abstraction and efficient translation by reusing existing SPARQL engines to produce Linked Data 'just-in-time'. We evaluate our approach against RDF stores, state-of-the-art SPARQL-to-SQL engines and streaming SPARQL engines, in the context of IoT data and scenarios. We show that storage efficiency, with succinct row storage, and query performance can be improved from 2 times to 3 orders of magnitude.

Keywords: SPARQL · SQL · Query translation · Analytics · Internet of Things · Web of Things

1 Introduction

The Internet of Things (IoT) envisions a world-wide, interconnected network of smart physical entities with the aim of providing technological and societal benefits [9]. However, as the W3C Web of Things (WoT) Interest Group charter[1] states, the IoT is currently beset by product silos and to unlock its potential, an open ecosystem based upon open standards for identification, discovery and interoperation of services is required.

We see a semantic Web of Things as such an information space, with rich descriptions, shared data models and constructs for interoperability that utilises but is not limited to semantic and web technologies to provide an application layer for IoT applications. As Barnaghi *et al.* [3] have proposed, semantic technologies can serve to facilitate interoperability, data abstraction, access and integration with other cyber, social or physical world data.

The semantic WoT does present a set of unique challenges: handling and storing time-series data as RDF, querying with SPARQL on limited IoT devices and distributed usage scenarios. Buil-Aranda *et al.* [5] have examined traditional

[1] https://www.w3.org/2014/12/wot-ig-charter.html.

SPARQL endpoints on the web and shown that performance for generic queries can vary by up to 3–4 orders of magnitude. Endpoints generally limit or have worsened reliability when issued with a series of non-trivial queries. IoT devices have added resource constraints, however, we argue that time-series IoT data and distribution also present the opportunity for specific optimisation.

The contribution of this paper is to present an optimisation of SPARQL-to-SQL query translation for the particular case of time-series data, both historical and streaming, with a novel approach that uses existing SPARQL engines to resolve Basic Graph Patterns and mappings that allow intermediate nodes of observations to be 'collapsed'. This is advised by our study of IoT schemata which exhibits a flat and wide structure. Our approach compares favourably to native RDF storage, SPARQL-to-SQL engines and RDF stream processing engines deployed on compact, resource-constrained devices, showing 2 times to 3 orders of magnitude performance and storage improvements on published sensor benchmarks and IoT use cases like smart homes.

In Sect. 2, we first study the structure of time-series IoT data which leads us, in Sect. 3, to study related work. We then describe the design and implementation of our approach, that employs metadata abstraction through mappings and SPARQL-to-SQL translation for performance, reusing, at the core, any existing SPARQL engine in Sect. 4. Finally, we evaluate our approach against traditional RDF stores, SPARQL-to-SQL engines and streaming engines using an established benchmark and a common IoT scenario in Sect. 5. Results are presented and discussed in Sect. 6 with the conclusion in Sect. 7.

2 Structure of Internet of Things Data

To investigate the structure of data produced by sensors in the Internet of Things, we collected the schemata of 19,914 unique IoT devices from public data streams on Dweet.io[2] over a one month period in January 2016.

Dweet.io is a cloud platform that supports the publishing of time-series data from IoT devices in JavaScript Object Notation (JSON). The schema represented in JSON can be flat (row-like with a single level of data) or complex (tree-like/hierachical with multiple nested levels of data). It was observed from the schemata, removing the 1542 (7.7 %) that were empty, that 18,280 (99.5 %) of the non-empty schemata were flat while only 92 (0.5 %) were complex.

We also analysed the schemata to investigate how wide the IoT data was. Wideness is defined as the number of properties beside the timestamp and a schema is considered wide if there are 2 or more such properties. We found that 92.2 % of the devices sampled had a schema that was wide. The majority (53.2 %) had 4 properties related to each timestamp. We also obtained a smaller alternative sample of 614 unique devices (over the same period) from Sparkfun[3], that only supports flat schemata, which confirmed that most (76.3 %) IoT devices sampled have wide time-series schemata.

[2] http://dweet.io/see.
[3] https://data.sparkfun.com/streams.

We concluded that our sample of over 20,000 unique IoT devices from Dweet.io and Sparkfun contained (1) flat and (2) wide IoT time-series data. It follows that a possible succinct representation of such data is as rows in a relational database with column headings corresponding to properties. SPARQL-to-SQL translation is then a possibility for querying. Investigating column stores was out of the scope of this study, however provides for interesting future work and comparison, as tension between inserts/updates and optimising data structures for reads [15] are reduced for time-series data which are already sorted by time in entry sequence order. The IoT schemata we collected is available on Github[4].

3 Related Work

The fact that we are dealing with time-series sensor data, represented as Linked Data with ontologies like the Semantic Sensor Network (SSN) ontology and Linked Sensor Data [12] for interoperability, prescribes the study of: (i) RDF stores, (ii) R2RML and SPARQL-to-SQL translation with relational databases to improve performance and storage efficiency for time series-data as rows and (iii) streaming engines for efficient processing on real-time streams.

RDF Stores. Virtuoso [8] is based on an Object Relational DBMS optimised for RDF storage while Jena Tuple Database (TDB) is a native Java RDF store using a single table to store triples/quads. Indexes, like the 6 SPO (Subject-Predicate-Object) permutations that Neumann *et al.* [11] propose often improve query performance on tables by reducing scans. TDB creates 3 triple indexes (OSP, POS, SPO) and 6 quad indexes while Virtuoso creates 5 quad indexes (PSOG, POGS, SP, OP, GS; G is graph). Commercial stores like GraphDB, formerly OWLIM, have also shown to perform well on benchmarks [4] with 6 indexes (PSO, POS, entities, classes, predicates, literals). Indexing, however, increases the storage size and memory required to load them.

Relational Databases (SPARL-to-SQL). Efficient SPARQL-to-SQL translation that improves performance and builds on previous literature has been investigated by Rodriguez-Muro and Rezk [14] and Priyatna *et al.* [13] with state-of-the-art engines ontop and morph respectively. Both engines support R2RML[5], a W3C recommendation based on the concept of mapping logical tables in relational databases to RDF via Triples Maps (the subject, predicate and object in a triple can be mapped to columns in a table). They also optimise query translation to remove redundant self-joins. Ontop, which translates mappings and queries to a set of Datalog rules, applies query containment and semantic query optimisation to create efficient SQL queries. However, (1) R2RML is designed for

[4] https://github.com/eugenesiow/iotdevices/releases/download/data/dweet_release.zip.

[5] http://www.w3.org/TR/r2rml/.

generality rather than abstracting and ‘collapsing’ (*reducing self joins on identifier columns in tables mapping to IRI templates*) intermediate nodes (Sect. 4) (2) Time-series data can be different from relational data (e.g. does not have primary keys) (3) The round-trip to retrieve database metadata (ontop) could be significant on devices with slower disk/memory access.

Streaming Engines. The C-SPARQL [1] engine supports continuous pull-based SPARQL queries over RDF data streams by using Esper[6], a complex event processing engine, to form windows in which SPARQL queries can be executed on an in-memory RDF model. CQELS [10] is a native RDF stream engine, supporting push and pull queries, that takes a ‘white-box’ approach for full control over query optimisation and execution. morph-streams, from $SPARQL_{stream}$ [6], supports query rewriting with R2RML mappings and execution with Esper.

4 Designing a SPARQL-to-SQL Engine for the IoT

Based on the ontologies for integrating time-series sensor data, the SSN ontology[7], Semantic Sensor Web and Linked Sensor Data (LSD) [12] mentioned in the previous section, we observe that semantic sensor data is modelled as (1) IoT *device metadata* like the location and specifications of sensors, (2) IoT *observation metadata* like the units of measure and types of observation (3) IoT *observation data* like timestamps and actual readings. Listing 1.1 shows an example division into the 3 categories from the Linked Sensor Data dataset in RDF Turtle.

Listing 1.1. LSD example, rainfall from Station 4UT01 (abbreviated)

```
@prefix ssw:<http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#>
@prefix weather:<http://knoesis.wright.edu/ssw/ont/weather.owl#>
@prefix wgs:<http://www.w3.org/2003/01/geo/wgs84_pos#>
@prefix time:<http://www.w3.org/2006/time#>
@prefix sen:<http://knoesis.wright.edu/ssw/>
sen:System_4UT01 ssw:processLocation              // Device Metadata
  [wgs:lat "40.82944"; wgs:long "-111.88222"].
_:obs a weather:RainfallObservation;              // Observation Metadata
  ssw:observedProperty weather:_Rainfall;
  ssw:procedure sen:System_4UT01;
  ssw:result _:data; ssw:samplingTime _:time.
_:data a ssw:MeasureData;
  ssw:uom weather:degrees.
_:time a time:Instant;
  time:inXSDDateTime "2003-03-31T12:35:00".       // Observation Data
_:data ssw:floatValue "0.1".
```

Although Linked Data as implemented in RDF is flexible and expressive enough to represent both data and metadata as triples as seen in Listing 1.1, however, given the resource constraints of IoT devices, we make these hypotheses:

1. Storing flat and wide IoT observation data as rows is more efficient than storage as RDF as each field value in a row, under a column header, does not require additional subject and predicate terms (Table 1).

[6] http://www.espertech.com/products/esper.php.
[7] https://www.w3.org/2005/Incubator/ssn/ssnx/ssn.

Table 1. LSD example, abbreviated row from the Table 4TU01

Time	Rainfall	RelativeHumidity	...
2003-03-31T12:35:00	0.1	37.0	...

2. Queries that retrieve more fields from a row (e.g. Rainfall & RelativeHumidity) will require less joins as compared to RDF stores' and perform better.
3. Most device and observation metadata can be abstracted and stored in-memory, with a mapping language that can express this. Metadata triples can be produced 'just-in-time' and intermediate nodes (e.g. ssw:MeasureData in Listing 1.1), if not projected in queries, can be 'collapsed' (reduces joins in RDF stores and self joins on identifier columns in tables that map to intermediate nodes, e.g. _:obs, _:data and _:time for SPARQL-to-SQL).
4. Efficient queries can be produced without relying on primary keys within time-series data and retrieving database schema from IoT devices.

4.1 sparql2sql and sparql2stream

We present, based on our hypotheses, sparql2sql (translates SPARQL-to-SQL) and sparql2stream (translates SPARQL to Event Processing Language (EPL) for streams) engines. They utilise the same core to provide a holistic approach to SPARQL translation for both historical and streaming IoT datasets.

Firstly, to support SPARQL-to-SQL translation, a mapping for IoT data stored as rows is required. R2RML (as in Sect. 3) is designed for generality rather than for specific IoT time-series data. As such, we propose S2SML in Sect. 4.2, an R2RML-compatible mapping language designed for metadata abstraction, collapsing intermediate nodes and in-memory storage.

Next, in Sect. 4.3, we explain how S2SML mappings can be used to translate SPARQL to SQL, reusing any existing SPARQL engine. Finally, in Sect. 4.4, we show how this applies for SPARQL on streams.

4.2 S2SML Mapping

Sparql2Sql Mapping Language (S2SML) mappings serve the dual purpose of providing bindings from rows and abstracting sensor and observation metadata from observation data stored as rows. Mappings are pure RDF and compatible with R2RML (can be translated to and from). Furthermore, S2SML is also designed to support 'collapsing' intermediate nodes of observation metadata through the use of blank nodes or faux nodes, nodes containing identifiers only created on projection. Listings 1.2 and 1.3 show a comparison of S2SML and R2RML from Listing 1.1. R2RML is more verbose and uses the {time} column for IRI templates, which might not be unique and cannot be 'collapsed' (Sect. 6.2).

Listing 1.2. S2SML

```
_:b a weather:RainfallObservation;
 ssw:result _:c.
_:c a ssw:MeasureData;
 ssw:floatValue
 "4UT01.Rainfall"^^<:LiteralMap>.
```

Listing 1.3. R2RML

```
:t1 a rr:TriplesMap; rr:logicalTable :4UT01;
 rr:subjectMap[rr:template "http://...o/{time}";
  rr:class weather:RainfallObservation];
 rr:predicateObjectMap[rr:predicate ssw:result;
  rr:objectMap[rr:parentTriplesMap :t2]].
:t2 a rr:TriplesMap; rr:logicalTable :4UT01;
 rr:subjectMap[rr:template "http://..m/{time}";
 rr:class ssw:MeasureData];
 rr:predicateObjectMap[rr:predicate ssw:floatValue;
  rr:objectMap[rr:column "Rainfall"]].
```

To define S2SML, we adopt the notation introduced by Chebotko *et al.* [7] where I, B, L denote pairwise disjoint infinite sets of IRIs, blank nodes and literals while I_{map}, L_{map}, F are IRI Map, Literal Map and Faux Node respectively. Examples can be found in Table 2. Combinations of these terms (e.g. $I_{map}IBF$) denote the union of their component sets (e.g. $I_{map} \cup I \cup B \cup F$).

Definition 1 (S2SML Mapping, m**).** *Given a set of all possible S2SML mappings, M, an S2SML mapping, $m \in M$, is a set of triple tuples, $(s, p, o) \in (I_{map}IBF) \times I \times (I_{map}IBL_{map}LF)$ where s, p and o are subject, predicate and object respectively.*

Table 2. Examples of elements in (s, p, o) sets

Symbol	Name	Example
I	IRI	<http://knoesis.wright.edu/ssw/ont/weather.owl#degrees>
I_{map}	IRI Map	<http://knoesis.wright.edu/ssw/{sensors.sensorName}>
B	Blank Node	_:bNodeId
L	Literal	“-111.88222”^^ <xsd:float>
L_{map}	Literal Map	“readings.temperature”^^ <s2s:literalMap>
F	Faux Node	<http://knoesis.wright.edu/ssw/obs/{readings.uuid}>

As shown in Table 2, I_{map} are IRI templates that consist of the union of IRI string parts (e.g. http://knoesis.wright.edu/ssw/) and reference bindings to table columns (e.g. {tableName.colName}). L_{map} are RDF literals whose value contains reference bindings to table columns (e.g. ”tableName.colName”) with a datatype of <s2s:literalMap>.

Definition 2 (Faux Node, F**).** *F is defined as an IRI template that consists of the union of a set of IRI string parts, I_p and a set of placeholders, U_{id}, referencing a table, so that $F = I_p \cup U_{id}$ and $|U_{id}| >= 1, |I_p| >= 1$.*

The example F in Table 2 shows how a placeholder is defined in the format of {tableName.uuid} with keyword '.uuid' identifying this as a Faux node.

Listing 1.2 shows an S2SML mapping of an LSD weather station 4UT01 in Salt Lake City. Observation data is referenced from table columns with Literal Maps, L_{map} (e.g. "4UT01.Rainfall"). Observation metadata which serves to connect nodes (e.g. _:c) is 'collapsed' through the use of blank nodes, B, which in R2RML (Listing 1.3) is mapped to {time} columns. The R2RML specification does support blank nodes but none of the other engines support their use yet. Faux nodes in S2SML are used if there is a possibility that the identifier/intermediate node will be projected in queries (described in Sect. 4.3). Finally, device metadata also contains constant Literals, L (e.g. the latitude of the sensor).

Mapping Closures. IoT devices might also have multiple sensors, each producing a time-series with a corresponding S2SML mapping. In Fig. 1, there might be multiple *observations mappings* each in different *readings* tables and a single sensors mapping and *sensors* table all forming a mapping closure.

Definition 3 (Mapping Closure, M_c**).** *Given the set of all mappings on a device, $M_d = \{m_d | m_d \in M\}$, where M is a set of all possible S2SML mappings, a mapping closure is the union of all elements in M_d, so $M_c = \bigcup_{m \in M_d} m$.*

Implicit Join Conditions. Observation data that is represented across multiple tables within a mapping closure might need to be joined if matched by a SPARQL query. In R2RML, one or more join conditions (*rr:joinCondition*) may be specified between triple maps of different logical tables.

In S2SML, these join conditions are automatically discovered as they are implicit within mapping closures from IRI template matching involving two or more tables. We define IRI template matching as follows.

Definition 4 (IRI Template Matching). *Let I_p be the set of IRI string parts in an element of I_{map}. I_{map_1} and I_{map_2} are matching if $\bigcup_{i_1 \in I_{p_1}} i_1 = \bigcup_{i_2 \in I_{p_2}} i_2$ and $\forall i_1 \in I_{p_1}, \forall i_2 \in I_{p_2} : pos(i_1) = pos(i_2)$ where $pos(x)$ is a function that returns the position of x within its I_{map}.*

Given matching I_{map}, join conditions can be inferred. Figure 1 shows a mapping closure consisting of a sensor and observation mapping. An IRI map in each of the mappings, *sen:system*{*sensors.name*} in I_{map_1} and *sen:system*{*readings.sensor*} in I_{map_2}, fulfil a template matching. A join condition is inferred between the columns *sensors.name* and *readings.sensor* as a result.

Compatibility with R2RML. S2SML is compatible with R2RML as they can be mutually translated without losing expressiveness. Triple Maps are translated to triples based on the elements in Table 2. Table 3 defines additional R2RML predicates and the corresponding S2SML construct. *rr:inverseExpression*, for

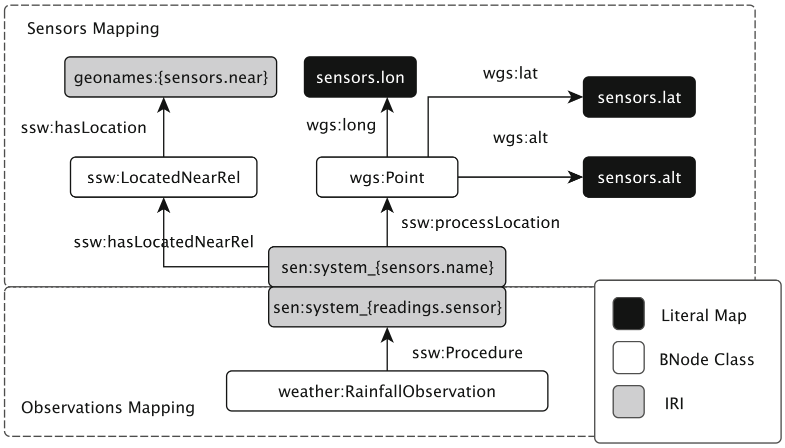


Fig. 1. Graph representation of an implicit join within a mapping closure

example, is encoded within a literal, L_{iv}, with a datatype of <s2s:inverse> and the *rr:column* denoted with double braces {{COL2}}. *rr:sqlQuery* is encoded by generating a context/named graph to group triples produced from that TripleMap and the query is stored in a literal object with context as the subject and <s2s:sqlQuery> as predicate. Faux nodes are translated as IRI templates. A specification of S2SML is available on the sparql2sql wiki on Github.

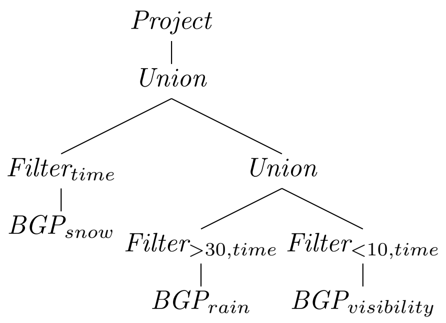
Table 3. Other R2RML predicates and the corresponding S2SML construct

R2RML predicate	S2SML example
rr:language	"literal"@en
rr:datatype	"literal"^^ <xsd:float>
rr:inverseExpression	"{COL1} = SUBSTRING({{COL2}}, 3)"^^ <s2s:inverse>
rr:class	?s a <ont:class>
rr:sqlQuery	<context1> {<sen:sys_{table.col}> ?p ?o.}
	<context1> s2s:sqlQuery "query"

4.3 Translation

Building a Mapping Closure. Following from Definition 3 of a Mapping Closure, M_c, a translation engine needs to perform, $\bigcup_{m \in M_d} m$, a union of all mappings on a device, M_d. To support template matching with any in-memory RDF store and SPARQL engine, as described in Definition 4, we replace all I_{map} within each mapping m with I_p, the union of IRI string parts, and extract C, the set of table column binding strings. C is then stored within map, m_{join}, with I_p as key and C as value. For example, in Fig. 1, <sen:system_> will replace both <sen:system_{sensors.name}> and <sen:system_{readings.sensor}> while m_{join} will store (<sen:system_>, {sensors.name, readings.sensor}).

SPARQL Algebra and BGP Resolution. A SPARQL query, *sparql*, can be translated by the function $trans(M_c, sparql)$. The first step within *trans* is $algebra(sparql) \rightarrow \propto$, where $\propto$ is a SPARQL algebra expression. For example, SRBench [17] query 6[8], which looks for weather stations that have observed low visibility within an hour of time by projecting stations that have either (by union) low visibility, high rainfall or snowfall observations, has $\propto$ as follows.



Basic graph patterns (BGPs) are sets of triple patterns within the query. *trans* walks through $\propto$ from the leaf nodes executing function $\sigma(M_c, BGP)$ on each BGP. As the M_c is pure RDF and represents the graph as it is, it can be loaded into an RDF store, ideally, in-memory. A SPARQL *select * query* containing the BGP within its *where* clause can then be executed on the M_c within the store. Literal datatypes are removed from the query and stored in a map. In the above example, BGP_{snow} and $BGP_{visibility}$ return no results for 4UT01 (Listing 1.1) but BGP_{rain} returns a result from σ. Each result from σ is a map of $(vk, vv) \in V \times (I_{map} IBL_{map} LF)$ where V is a variable in a triple pattern. The (vk, vv) maps are passed to the operator, $\propto_{op}$ above in $\propto$. Eventually, an SQL union is performed at the project operator π for all $|\sigma| > 1$. We have implemented a pluggable BGP resolution interface to show various in-memory RDF stores can be supported, with Jena and Sesame as reference examples.

Syntax Translation. *trans* continues its walk from BGP leaf nodes through $\propto$ to the root. At each node, $\propto_{op}$, a syntax translation $syn(SQL_i, (vk, vv)_i, \propto_{op}) \rightarrow (SQL_o, (vk, vv)_o)$ is performed, producing an updated SQL query, SQL_o. In the example, at the $Filter_{>30,time}$ $\propto_{op}$, SQL_i which consists of a blank SQL *where* clause is updated using $(vk, vv)_i$ to translate restrictions on ?*time* and ?*value* to those with bindings 4UT01.time<...T17:00:00 and 4UT01.Rainfall>30. The SQL *from* clause is also updated with the table 4UT01. An unchanged $(vk, vv)_o$ and the updated SQL_o are output from *syn* and passed upwards.

Table 4 shows a list of common operators $\propto_{op}$ and their corresponding SQL clauses and *syn* descriptions. If an operator uses (vk, vv) for mapping a V and

[8] https://github.com/eugenesiow/sparql2sql/wiki/Q06.

Table 4. Operators $\propto_{op}$ and corresponding SQL Clauses

$\propto_{op}$	SQL clause	Remarks
Project π	Select, From	Restricts relation to subset using (vk, vv)
Extend ρ	Select	Renames an attribute in (vk, vv)
Filter ς	Where, Having, From	Restriction translated using (vk, vv)
Union $\cup$	From	Add unrestricted select of SQL_i in FROM
Group γ	Group By, From	Aggregation translated using (vk, vv)
Slice ς_S	Limit	Add a LIMIT clause
Distinct ς_D	Select	Add a DISTINCT to SELECT clause
Left Join $\bowtie$	Left Join..On, Select	If I add to (vk, vv), else LEFT JOIN

retrieves a I_{map}, L_{map} or F, it adds the table binding to the FROM clause. If there are tables in the FROM without join conditions, a cartesian product (cross join) of two tables is taken. Finally, if faux nodes, F, are encountered in π, an SQL update (UPDATE table SET col=RANDOM_UUID()) is run to generate identifiers and vv in (vk, vv) is updated from {table.uuid} to {table.col}.

4.4 Streaming

The mapping and translation design can be used to translate SPARQL to Event Processing Language (EPL) for streams. Listing 1.4 shows the additional syntax in the SPARQL *from* clause specified in Extended Backus Naur Form.

Listing 1.4. SPARQL FROM Clause Definition for sparql2stream

```
FromClause = FROM NAMED STREAM <StreamIRI> [RANGE Time TimeUnit WindowType]
TimeUnit = ms | s | m | h | d
WindowType = TUMBLING | STEP
```

A *TUMBLING* window is a pull-based buffer that reevaluates at the specified time interval while the *STEP* window is a push-based sliding window extending for the specified time interval into the past. The *syn* function is modified to support EPL as an SQL dialect. Streaming for the IoT is useful for 1) scenarios with high sampling (e.g. accelerometers) or insertion rate (e.g. many sensors to a device/hub) and 2) applications that perform real-time analytics requiring push-based results from queries rather than results at pull intervals.

5 Experiment

To evaluate our approach against RDF stores, SPARQL-to-SQL engines and streaming engines in an WoT context, we selected two unique IoT scenarios using published datasets. Code and experiments can be found on Github[9].

[9] https://github.com/eugenesiow/sparql2sql.

Distributed Meteorological System. The first scenario uses Linked Sensor Data with sensor metadata and observation data from about 20,000 weather stations across the United States. In particular, we used the period of the Nevada Blizzard (100k triples) for storage and performance tests and the largest Hurricane Ike period (300k triples) for storage tests. SRBench [17] is an accompanying analytics benchmark for streaming SPARQL queries but can be applied, with similar effect, to SPARQL queries constrained by time. Queries[10] 1 to 10 were used as they involve time-series sensor data while the remaining queries involved integration or federation with DBpedia or Geonames which was not within the scope of the experiment. Queries are available on Github[11]. The experiment simulates a distributed setup as each station's data is stored on an IoT device as RDF or rows with S2SML or R2RML mappings. Queries are broadcast to all stations, total query time was the maximum time as the slowest station was the limiting factor. Due to resource constraints, we assumed broadcast and individual connection times to be similar over a gigabit switch, hence, distributed tests for the 4700+ stations were run in series, recording individual times, averaging over 3 runs and taking the maximum amongst stations for each query.

Smart Home Analytics Benchmark. This scenario uses smart home IoT data collected by Barker *et al.* [2] over 3 months in 2012. 4 queries[12] requiring space-time aggregations with a variety of data for descriptive and diagnostic analytics were devised. (1) hourly aggregation of temperature, (2) daily aggregation of temperature, (3) hourly and room-based aggregation of energy usage and (4) diagnosis of unattended devices through energy usage and motion, aggregating by hour and room. Time taken for queries were averaged over 3 runs.

Environment and Stores. The IoT devices used were Raspberry Pi 2 Model B+s' with 1GB RAM, 900MHz quad-core ARM Cortex-A7 CPU and Class 10 SD Cards, as they are widely available and relatively powerful. 512mb was assigned to the Java Virtual Machine on Raspbian 4.1. Ethernet connections were used between the querying client (i5 3.2GHz, 8GB RAM, hybrid drive) and the Pis'.

RDF stores compared were TDB (Open Source) and GraphDB (Commercial). Virtuoso 7 was not supported on the 32-Bit Raspbian and Virtuoso 6 did not support SPARQL 1.1 time functions like *hours*. H2[13] (disk mode) was used as the relational store for all SPARQL-to-SQL tests. ontop and morph were tested within the limits of query compatibility and a quantitative evaluation of SQL queries and translation time was done. Native SPARQL streaming engine CQELS was compared for push-based performance. As CQELS already benchmarked against C-SPARQL and push results for real-time analytics helped differentiate streams, we did not compare against C-SPARQL.

[10] http://www.w3.org/wiki/SRBench.

[11] https://github.com/eugenesiow/sparql2sql/wiki.

[12] https://github.com/eugenesiow/ldanalytics-PiSmartHome/wiki/.

[13] http://www.h2database.com/.

6 Results and Discussion

6.1 Storage Efficiency

Table 5 shows the store sizes of different datasets for the H2, TDB and GraphDB setups. As time-series sensor data benefits from succinct storage as rows, H2 outperformed the RDF stores, which also suffered from greater overheads for multiple stores and indexing [16], from about one to three orders of magnitude.

Table 5. Store size by dataset (in MB)

Dataset	#Store(s)	H2	TDB	GraphDB	Ratio
Nevada Blizzard	4701	90	6162	121694	1:68:1352
Hurricane Ike	12381	761	85274	345004	1:112:453
Smart home	1	135	2103	1221	1:15:9

6.2 Query Performance

Figure 2 shows the performance of SRBench queries on the various stores with the Nevada Blizzard dataset. We see that our sparql2sql approach performs better consistently on all queries with stable average execution times. We argue that this was the result of SQL queries produced not having joins as each station was a single time-series (wide) and intermediate nodes not being projected (could be 'collapsed'). GraphDB generally performed better than the TDB store especially on query 9 due to TDB doing a time consuming join operation in the low-resource environment between two subtrees, WindSpeedObservation and WindDirectionObservation. If queries were executed to retrieve subgraphs individually with TDB, each query cost a 100 times less. Query 4 was similar but with TemperatureObservation and WindSpeedObservation subgraphs instead.

Both ontop (v1.6.1) and morph (v3.5.16), at the time of writing, will only support the aggregation operators required for queries 3 to 9 sans query 6 in future versions. morph was also unable to translate queries 6 and 10 as yet while ontop's SQL query 10 did not return from the H2 store on some stations (e.g. BLSC2). ontop performs better than the RDF stores on queries 2 and 6. Although queries 1 and 2 are similar in purpose, query 2 has an OPTIONAL on the unit of measure term, hence as shown in Table 6, ontop generates different structured queries, explaining the discrepancy in time taken.

We did an additional comparison between SPARQL-to-SQL engines in terms of the structure of queries generated and translation time. Table 6 shows the average translation time, t_{trans} of the 3 engines on the client. The plugin BGP resolution engine for sparql2sql (s2s) used was Jena. Both ontop and morph have additional inference/reasoning features and ontop makes an extra round trip to the Pi to obtain database metadata explaining the longer translation times.

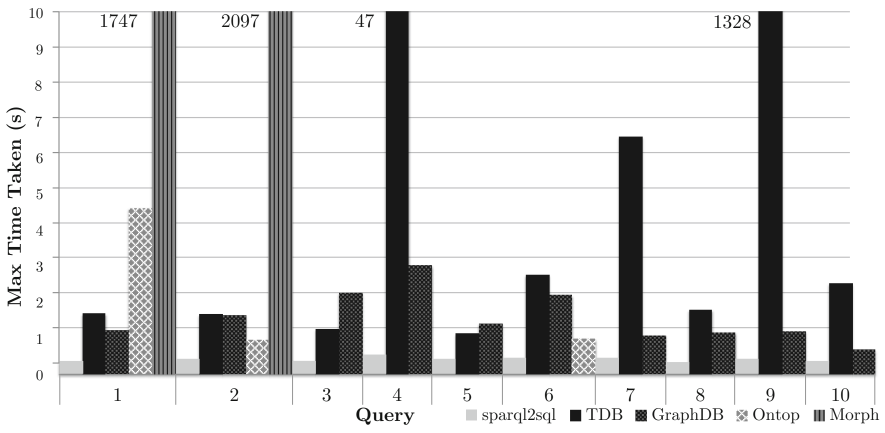


Fig. 2. Max time taken for distributed SRBench queries

In R2RML, as shown in Listing 1.3, in the absence of row identifiers in time-series data, time has to be used in IRI templates for intermediate observation metadata nodes. As timestamps are not unique in LSD (observed from data), they are not suited as a primary key, hence cannot be used to chase equality generating dependencies in the semantic query optimisation ontop does [14]. The resulting queries from ontop and morph both have redundant inner joins on the time column (used to model intermediate IRIs in R2RML).

In the smarthome scenario, sparql2sql query performance on aggregation queries as shown in Fig. 3 is still ahead of the RDF stores. GraphDB also has all-round better performance than TDB. All the queries performed SPARQL 1.1 space-time aggregations, excluding the other SPARQL-to-SQL engines.

Through the experiments, we observe that although other SPARQL-to-SQL engines have reported significant performance improvements over RDF stores on various benchmarks and deployments, there is still room for optimisation for IoT devices and scenarios and perform below RDF stores on Pis' or do not yet support queries relevant to IoT scenarios such as aggregations. sparql2sql with S2SML, utilises the strengths of SPARQL-to-SQL on IoT scenarios and

Table 6. SPARQL-to-SQL translation time and query structure

Q	t_{trans} (ms)			Joins			Join type and structure	
	s2s	Ontop	Morph	s2s	Ontop	Morph	Ontop (qview)	Morph
1	16	702	146	0	6	4	implicit	4 inner
2	17	703	144	0	6	4	5 nested, 1 left outer	4 inner
6	19	703	-	0	5	-	5 implicit	-
10	32	846	-	0	6	-	UNION(2x3 implicit)	-

time-series data and performed better than both RDF stores and SPARQL-to-SQL engines. Table 8 summarises the average query times for all the tests.

6.3 Push-Based Streaming Query Performance

Table 8 shows the average time taken to evaluate a query from the insertion of an event to the return of a push-based result from sparql2stream, t_{s2r} and CQELS, t_{CQELS} with 1 s delays in between. This was averaged over 100 results. For sparql2stream, the one-off translation time at the start (ranging from 16ms to 32ms) was added to the sum during the average calculation. Query 6 of SRBench was omitted due to EPL and CQELS not supporting the UNION operator. The sparql2stream engine (using Esper to execute EPL) showed over two orders of magnitude performance improvements over CQELS. Queries 4, 5 and 9 that involved joining subgraphs (e.g. WindSpeed and WindDirection in 9) and aggregations showed larger differences. It was noted, that although CQELS returned valid results for these queries, they contained an increasing number of duplicates (perhaps from issues in the adaptive implementation) which caused a significant slowdown over time and when averaged over 100 pushes. The experiments are available on Github[14,15].

This ability to answer queries in sub-millisecond average times in a push-based fashion makes sparql2stream a viable option for real-time analytics on IoT devices like medical devices that require reacting instantaneously.

To verify that sparql2stream was able to answer SRBench queries close to the rate they are sent, even at high velocity, we reduced the delay between insertions from 1000 ms to 1 ms and 0.1 ms. Table 8 shows a summary of the average latency (the time from insertion to when query results to be returned) of each query (in ms). We observe that the average latency is slightly higher than the inverse of the rate. The underlying stream engine, Esper, maintains context partition states consisting of aggregation values, partial pattern matches and data

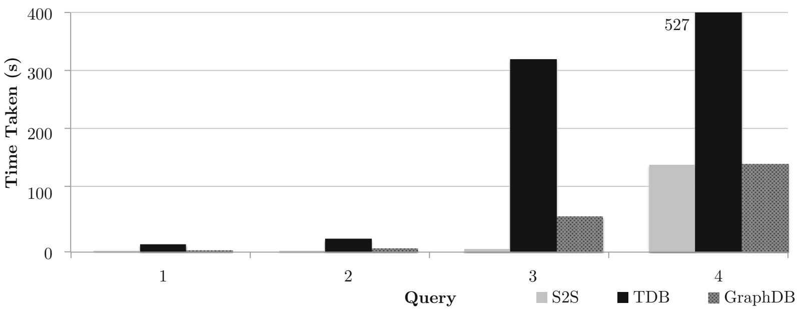


Fig. 3. Average time taken for smarthome analytical queries

[14] https://github.com/eugenesiow/cqels.
[15] https://github.com/eugenesiow/sparql2stream.

Table 7. Average query run times (in ms)

SR_{Bench}	t_{s2s}	t_{TDB}	t_{GDB}	t_{ot}	t_{morph}	Ratio	t_{s2r}	t_{CQELS}	Ratio
1	365	1679	1223	4589	1747702	1:5:3:13:4k	0.47	138	1:294
2	415	1651	1627	945	2097159	1:4:4:2:5k	0.46	119	1:261
3	375	1258	2251	-	-	1:3:6	0.66	202	1:306
4	533	47084	3004	-	-	1:88:6	0.67	186k	1:277k
5	415	1119	1404	-	-	1:3:3	0.63	1476k	1:3243k
6	457	2751	2181	987	-	1:6:5:2	-	-	-
7	455	6563	1082	-	-	1:14:2	0.66	2885	1:5245
8	320	1785	1162	-	-	1:6:4	0.67	282	1:426
9	436	1328197	1175	-	-	1:3k:3	0.67	188k	1:280k
10	354	2514	685	-	-	1:7:2	0.73	72	1:98

Smarthome	t_{s2s}	t_{TDB}	t_{GDB}	Ratio	t_{s2r}	t_{CQELS}	Ratio
1	466	13709	3132	1:29:7	0.64	125	1:196
2	2457	21898	6914	1:9:3	0.77	129	1:167
3	4685	322357	59803	1:69:13	0.81	-	-
4	147649	527184	147275	1:4:1	3.78	-	-

Table 8. Average latency (in ms) at different rates

R	$Q_{\#}$ 1	2	3	4	5	7	8	9	10
1	1.300	1.374	1.279	1.303	1.2561	1.268	1.267	1.295	1.255
10	0.155	0.159	0.143	0.161	0.1291	0.137	0.141	0.155	0.129

R = Rate(rows/ms), $Q_{\#}$ = Query number

windows. At high rates, the engine introduces blocking to lock and protect context partition states. However, Fig. 4 shows the effect of this blocking is minimal as the percentage of high latency events is less than 0.3 % (note that x-axis is 99 % to 100 %) across various rates. This comparison which groups messages by latency ranges is also used in the Esper benchmark and by Calbimonte *et al.* [6].

We also tested the size of data that can fit in-memory for sparql2stream with SRBench Query 8, that uses a long *TUMBLING* window. The engine ran out of memory after 33.5 million insertions. Given a ratio of 1 row to 75 triples within the SSN mapping (each observation type with 10+ triples), by projection, an RDF dataset size of 2.5 billion triples was 'fit' in a IoT devices' memory.

Queries 1 and 2 of the smart home scenario also corroborated the 2 orders of magnitude performance advantage of sparql2stream over CQELS. Queries 3 and 4 were not run on CQELS due to issues with the FILTER operator in the version tested. Query 4 which involved joins on motion and meter streams and an aggregation saw the average latency of sparql2stream increase, though still stay under 4ms. The latency for this query was measured from the insertion time of the last event involved (that trips the push) to that of the push result.

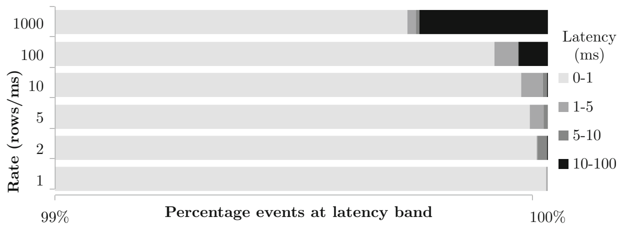


Fig. 4. Percentage latency at various rates for Q1

7 Conclusion

A Web of Things based on open standards and the innovations introduced in the Semantic Web and Linked Data can encourage greater interoperability and bridge product silos. This paper shows how time-series Internet of Things data that is flat and wide, can be stored efficiently as rows on devices with limited resources. By optimising SPARQL-to-SQL translation and 'collapsing' intermediate nodes, performance on smart home monitoring and a distributed meteorological system show storage and query performance improvements that range from 2 times to 3 orders of magnitude. The independence from primary keys and database metadata also resulted in less joins in resultant SQL queries and faster query translation times respectively. Future work will expand experimentation to consider additional datasets, data sizes, queries and include a greater variety of stores and stream processing use cases for time-series data e.g. column stores, stream analytics and compression/approximation.

The limitations of this approach lie in the assumption that the bulk of IoT time-series data is flat and read-only which might change in the future. Exploiting the wideness of time-series data for row access performance is also query dependant. Current state-of-the-art Ontology-Based Data Access (OBDA) systems, which do query translation, support general use cases (web/enterprise relational database mapping) and support reasoning which our approach does not seek to address at the moment.

References

1. Barbieri, D.F., Braga, D., Ceri, S., Valle, E.D., Grossniklaus, M.: Querying RDF streams with C-SPARQL. ACM SIGMOD Rec. **39**(1), 20 (2010)
2. Barker, S., Mishra, A., Irwin, D., Cecchet, E.: Smart*: an open data set and tools for enabling research in sustainable homes. In: Proceedings of the Workshop on Data Mining Applications in Sustainability (2012)
3. Barnaghi, P., Wang, W.: Semantics for the internet of things: early progress and back to the future. Int. J. Semant. Web Inf. Syst. **8**(1), 1–21 (2012)
4. Bishop, B., Kiryakov, A., Ognyanoff, D.: OWLIM: a family of scalable semantic repositories. Semant. Web **2**(1), 33–42 (2011)

5. Buil-Aranda, C., Hogan, A., Umbrich, J., Vandenbussche, P.-Y.: SPARQL web-querying infrastructure: ready for action? In: Alani, H., et al. (eds.) ISWC 2013, Part II. LNCS, vol. 8219, pp. 277–293. Springer, Heidelberg (2013)
6. Calbimonte, J.P., Jeung, H., Corcho, O., Aberer, K.: Enabling query technologies for the semantic sensor web. Int. J. Semant. Web Inf. Syst. **8**(1), 43–63 (2012)
7. Chebotko, A., Lu, S., Fotouhi, F.: Semantics preserving SPARQL-to-SQL translation. Data Knowl. Eng. **68**(10), 973–1000 (2009)
8. Erling, O.: Implementing a sparql compliant RDF triple store using a SQL-ORDBMS. Technical report, OpenLink Software (2001)
9. International Telecommunication Union: Overview of the Internet of things. Technical report (2012)
10. Le-Phuoc, D., Dao-Tran, M., Xavier Parreira, J., Hauswirth, M.: A native and adaptive approach for unified processing of linked streams and linked data. In: Aroyo, L., et al. (eds.) ISWC 2011, Part I. LNCS, vol. 7031, pp. 370–388. Springer, Heidelberg (2011)
11. Neumann, T., Weikum, G.: x-RDF-3X. Proc. VLDB Endow. **3**, 256–263 (2010)
12. Patni, H., Henson, C., Sheth, A.: Linked sensor data. In: Proceedings of the International Symposium on Collaborative Technologies and Systems (2010)
13. Priyatna, F., Corcho, O., Sequeda, J.: Formalisation and experiences of R2RML-based SPARQL to SQL query translation using morph. In: Proceedings of the 23rd International Conference on World Wide Web, pp. 479–489 (2014)
14. Rodriguez-Muro, M., Rezk, M.: Efficient SPARQL-to-SQL with R2RML mappings. Web Semant. Sci. Serv. Agents WWW **33**, 141–169 (2014)
15. Stonebraker, M., Abadi, D., Batkin, A.: C-store: a column-oriented DBMS. In: Proceedings of VLDB, pp. 553–564 (2005)
16. Weiss, C., Karras, P., Bernstein, A.: Hexastore: sextuple indexing for semantic web data management. In: Proceedings of the VLDB Endowment (2008)
17. Zhang, Y., Duc, P.M., Corcho, O., Calbimonte, J.-P.: SRBench: a streaming RDF/SPARQL benchmark. In: Cudré-Mauroux, P., et al. (eds.) ISWC 2012, Part I. LNCS, vol. 7649, pp. 641–657. Springer, Heidelberg (2012)