Optimizing a Semantically Enriched Hypercat-enabled Internet of Things Data Hub (Short Paper)

Ilias Tachmazidis[1], Sotiris Batsakis[3,1], John Davies[2], Alistair Duke[2], Grigoris Antoniou[1], and Sandra Stincic Clarke[2]

[1] University of Huddersfield, Huddersfield, UK
[2] British Telecommunications, Ipswich, UK
[3] Technical University of Crete, Greece

Abstract. Large volumes of data is generated from the increasing number of sensor networks and smart devices. Such data is generated and published in multiple formats, thus highlighting the significance of interoperability for the success of what has come to be known as the *Internet of Things* (IoT). The *BT Hypercat Data Hub* provides a focal point for the sharing and consumption of available datasets from a wide range of sources. In this work, we present a series of optimizations applied on the BT Hypercat Data Hub that enabled scalable SPARQL query answering over relational databases and an access control mechanism that filters SPARQL results based on user's subscriptions.

1 Introduction

The number of interconnected smart devices is constantly increasing, producing a huge amount of data that has to be represented and exchanged using common data formats and protocols that form the *Internet of Things* (IoT), which is used in applications such as smart cities. A smart city is based on the use of technology in order to improve the efficiency, effectiveness and capability of various city services, thus improving the quality of the inhabitants' lives [10]. Such application area is characterized by the vast variety of the technologies used, the types and volumes of data, and the services and applications targeted [2]. Thus, developing successful smart city solutions requires the collection and maintenance of relevant data in the form of IoT data, combined with scalable implementations and efficient access control mechanisms.

Addressing interoperability issues by focusing on how interoperability could be achieved between data hubs in different domains was a major objective for the development of Hypercat [1], which is a standard for representing and exposing Internet of Things data hub catalogues [3] over web technologies. In [8], the semantic enrichment for the core of the Hypercat specification, namely an RDF-based [5] equivalent for a JSON-based catalogue was proposed.

The *BT Hypercat Data Hub* supports access to the enriched data through a SPARQL endpoint [6] combined with reasoning capabilities and the ability

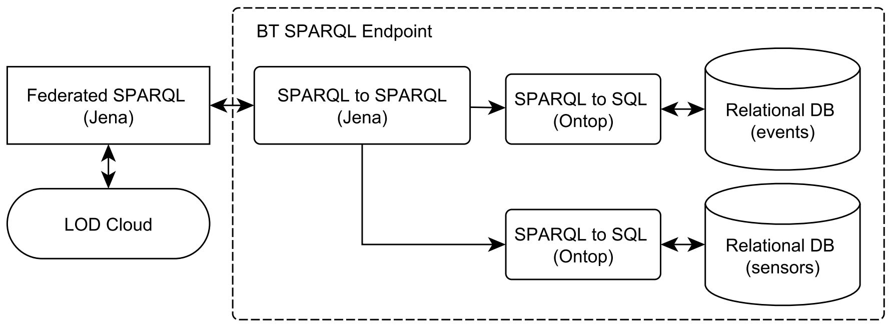

Fig. 1. BT Data Hub Architecture.

to combine external data sources using federated queries [7]. In [7], the data in the *BT Hypercat Data Hub* is stored in relational databases and since this data is frequently updated, a dynamic solution based on a mapping from relational databases to corresponding *BT Hypercat Ontology* concepts has been adopted. Thus, instead of copying the existing data into an RDF triplestore, submitted SPARQL queries are dynamically translated into a set of SQL queries on top of the existing relational databases.

Although this approach was efficient for most querying and reasoning tasks, scalability for certain queries was limited and certain optimization both on the relational database schema and the corresponding ontology had to be employed. In addition, the existing system did not supported an access control mechanism, which is an important part of related systems [4]. In this work, these issues are addresed, and both optimizations and an access control mechanism are proposed and implemented, resulting in a scalable, industrial scale IoT system, integrating Semantic Web technologies and access control.

This work is organized as follows: Section 2 contains background information about the *BT Hypercat Data Hub*. Section 3 contains a description of the applied optimizations to the *SPARQL to SQL* endpoint which enabled scalable SPARQL query answering over relational databases. An access control mechanism over the developed SPARQL endpoints is presented in Section 4, while conclusions and future work are discussed in Section 5.

2 BT Hypercat Data Hub

In this section, we describe the basic components of the *BT Hypercat Data Hub*, which aggregates and catalogues mulitple IoT data sources and exposes them via a uniform RESTful API. Figure 1 presents the architecture of the data hub, more specifically (for more details readers are referred to [7,9]):

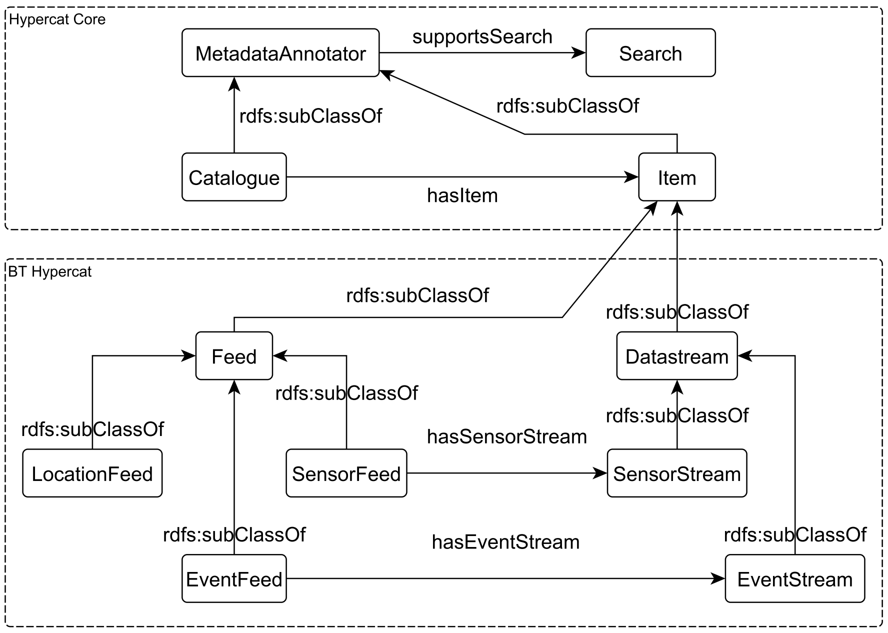

Fig. 2. BT Hypercat Ontology.

– The *BT Hypercat Ontology* (see Figure 2) enables the publication of an RDF-based Hypercat catalogue [8] as well as the translation of data stored in a relational database into RDF format.
– *RDF Adapters* provide internally stored data in N-Triples format following a systematic generation of URIs.
– A *SPARQL to SQL* endpoint enables the dynamic translation of SPARQL queries into SQL queries, using Ontop[4].
– The *BT SPARQL Endpoint* queries internally available *SPARQL to SQL* endpoints and combines SPARQL results, using Apache Jena[5].
– Federated querying is enabled by providing a Jena endpoint that allows federated queries over the *BT SPARQL Endpoint* and SPARQL endpoints that are available through the Linked Open Data cloud.

3 Optimizing a SPARQL to SQL Endpoint

The *BT Hypercat Data Hub* has been successfully deployed as part of two use cases. The first being the *SimplifAI* project, which is aimed at urban traffic management and control in order to reduce traffic and improve air quality, and

[4] http://ontop.inf.unibz.it/
[5] https://jena.apache.org/index.html

the second being *City Concierge*, which is a use case of the *CityVerve* project aiming to increase uptake of walking and cycling as a preferred travel mode in Greater Manchester.

In order to provide the required functionality for both use cases, the *SPARQL to SQL* endpoint needed to be optimized, thus ensuring rapid responses. However, the developed optimizations that are presented in this section, are based on the assumption that the reader has already some understanding of the internal functionality of a *SPARQL to SQL* endpoint (for details see [7,9]).

The first step towards the development of efficient mappings was to optimize the ontology itself, namely:

- Restrict class hierarchy only to classes that are used by Ontop for mappings (faster reasoning, duplicate reduction).
- Restrict property hierarchy only to properties that are used by Ontop for mappings (faster reasoning, duplicate reduction).
- Delete domain and range assertions from properties when each class has a separate Ontop mapping (duplicate reduction).
- Study Ontop's reasoning capabilities, for the given ontology, in order to ensure that reasoning does not lead to SQL plans that generate duplicate results.

A thorough investigation of the developed ontology revealed certain inefficiencies. Thus, the ontology was reduced to a bare minimum (exclusively for Ontop) in order to prevent duplicates. A close examination of the minimal ontology showed that the combination of the developed mappings and Ontop's reasoning capabilities could indeed lead to the unique generation of each RDF resource.

Once the unique generation of each RDF resource is ensured, mappings should be studied for potential inefficiencies in terms of generated SQL plans. More specifically, the use of an SQL function (such as *TO_TIMESTAMP()* for time, *unnest()* for arrays and *ST_AsText()* for PostGIS geometry) in a given mapping, is translated into a separate subquery. Such subqueries are inefficient as they are not indexed and they could lead to unnecessary self-joins over a given SQL table. Thus, columns representing time or PostGIS geometry need to be translated into a simpler form (such as *character*), while columns representing arrays need to be stored in a separate SQL table. For example, the following SQL table:

```
TABLE feed(id uuid NOT NULL, updated bigint, tag character
varying[], the_geom geometry);
```

should be translated into the following SQL tables[6]:

[6] In order to keep the initial database intact, VIEW and MATERIALIZED VIEW were considered. However, both solutions deteriorate the performance because Ontop's reasoner is able to retrieve the initial table schema, and thus, each mapping is translated into a separate subquery (regardless of whether an SQL function is used or not). In order to overcome this issue, new SQL tables need to be defined, which

```
TABLE sparql_feed(id uuid NOT NULL, updated character varying,
the_geom character varying);

TABLE sparql_feed_tag(id uuid NOT NULL, tag character varying
NOT NULL);
```

based on the following data translations:

```
INSERT INTO sparql_feed (id, updated, the_geom)
SELECT id, TO_TIMESTAMP(feed.updated) AS updated,
       ST_AsText(feed.the_geom) AS the_geom
FROM feed

INSERT INTO sparql_feed_tag (id, tag)
SELECT feed_tag.id, feed_tag.tag
FROM (SELECT feed.id, unnest(feed.tag) AS tag
      FROM feed) AS feed_tag
```

Thus, mappings should be based on the SQL tables *sparql_feed* and *sparql_feed_tag* (instead of *feed*), hence retrieving each field without the need for data translation based on SQL functions. Note that the key of *sparql_feed* is *id*, while the key of *sparql_feed_tag* is *id,tag* in order to allow an efficient join operation between the two tables based on *id*.

Based on the following prefixes that are used in order to shorten URIs:

bt-sensors: http://api.bt-hypercat.com/sensors/
bt-hypercat: http://portal.bt-hypercat.com/ontologies/bt-hypercat#

the following mapping maps the data property *feed_updated* of class *Feed*:

Mapping ID	mapping:feed_updated
Target (Triple Template)	bt-sensors:feeds/{sparql_feed.id} bt-hypercat:feed_updated {sparql_feed.updated} .
Source (SQL Query)	SELECT sparql_feed.id, sparql_feed.updated FROM sparql_feed

Note that the SQL variables that are used in order to generate RDF triples following the triple template (see Target) should match the columns that belong to the key (here *sparql_feed.id*) of the corresponding SQL table. This is important in order to eliminate self-joins. If the key contains more columns than those used in the RDF triple pattern to be generated, then self-joins cannot be eliminated and each mapping for the given table will be translated as a separate subquery. Allowing self-joins can be manageable for relatively small tables (containing thousands of rows, provided that the table is indexed), but can be prohibitive for larger tables (containing millions of rows).

provide a manually created view over the existing data in a format that allows fast SPARQL queries.

Finally, the following guidelines should be taken into consideration in order to avoid SPARQL queries that lead to excessive SQL query plans (after the dynamic translation from SPARQL to SQL), namely:

- Avoid generic triple patterns such as "?s ?p ?o" as they will be translated using all available mappings (a UNION of all defined mappings), leading to an excessive SQL query plan. Thus, the predicate within each triple pattern should be specified.
- Avoid using DISTINCT as it deteriorates severely performance since the final results would be sorted and filtered for unique values at the end of the SQL query plan.
- Retrieve specific *Feed*s by filtering (namely using FILTER). Restricting the search space early (e.g., retrieving a feed based on its id) leads to more efficient SQL query plans.
- Avoid OPTIONAL as each OPTIONAL is translated into a LEFT OUTER JOIN. If used, all OPTIONAL should be put at the end of the query in order to improve query translation.
- Use LIMIT as limiting the amount of required results could speed up the query execution.

By applying the aforementioned optimizations to the *BT Hypercat Data Hub*, queries that were unresponsive in the non-optimized system were executed in less than a second by the optimized system. The optimized system was able to handle efficiently queries over multiple SQL tables, containing millions of records.

4 Access Control Mechanism

An access control mechanism has been implemented for the *SPARQL to SQL* endpoint and the *SPARQL to SPARQL* endpoint while retaining the high level of performance achieved by the above mentioned optimizations. The proposed access control mechanism allows user access management at the feed level, namely if users have access to a feed then they also have access to any information contained within that feed.

Access control is enforced in two stages, first the given query is examined syntactically. Since the system supports SPARQL 1.0, only a subset of SPARQL constracts is relevant. Indeed, *WHERE* is the main point of interest in terms of enforcing access control. More specifically, the *WHERE* clause could contain feed URIs, defined as constants, that users have no access to. Consider the following SPARQL query:

```
PREFIX hypercat: <http://portal.bt-hypercat.com/ontologies/bt-hypercat#>
BASE <http://api.bt-hypercat.com/>
SELECT ?f1
WHERE {
  <sensors/feeds/00000000-0000-0000-0000-000000000001>
  hypercat:feed_id ?f1.
}
```

Note that this query should be allowed to execute only if users have access to feed <http://api.bt-hypercat.com/sensors/feeds/00000000-0000-0000-0000-000000000001>.

In the case that query execution cannot be denied by syntactic analysis, the system still needs to ensure that users do not have access to results based on variables that are bound with information from restricted feeds. Consider the following example:

```
PREFIX hypercat: <http://portal.bt-hypercat.com/ontologies/bt-hypercat#>
SELECT ?f1
WHERE {
  ?feed hypercat:feed_id ?f1.
  ?feed hypercat:feed_creator ?f2.
}
```

It is insufficient to check only returned results (here *?f1*) since other variables (here *?feed* and *?f2*) might attempt to extract information from feeds that the user has no access to. Thus, the query is rewritten internally so that the *SELECT* clause would contain all variables in the *WHERE* clause (here *?f1*, *?feed* and *?f2*), even though users will be provided with results only for requested variables (here *?f1*), provided that access is allowed (to all variables). Note that results might be combined from different feeds, while in order to access a triple through Ontop either a fixed URI (syntactic analysis) or a variable (result checking over all variables) would be used within the triple. Moreover, by checking all variables for the given query, manipulating results through *FILTER* becomes ineffective.

Finally, in order to ensure that a given *ASK* query provides the same level of access control, after it is syntactically checked for allowed URIs, it is translated into a *SELECT* query with *LIMIT 1*. Thus, the result of an *ASK* query is *true* if users have access to at least one result of the equivalent *SELECT* query, or *false* otherwise.

It is worth mentioning that integrating the access control mechanism to the optimized system has not affected the performance of the SPARQL endpoints since query times remained almost identical regardless of whether the access control mechanism was enabled or not.

5 Conclusion

In this work, a series of applied optimizations to the *BT Hypercat Data Hub* has been presented, thus scaling up SPARQL query answering over relational databases. In addition, an access control mechanism that filters SPARQL results based on user's subscriptions has been proposed. Both query optimization and access control mechanisms that were addressed in this work are critical factors for the successful deployment of a large scale IoT system. Future work includes further semantic enrichment by enabling GeoSPARQL queries. In addition, spatiotemporal reasoning is a prominent direction that could provide richer knowledge by combining data coming from both the *BT Hypercat Data Hub* and the LOD cloud.

References

1. Pilgrim Beart. Hypercat 3.00 Specification, 2016.
2. Mathieu d'Aquin, John Davies, and Enrico Motta. Smart cities' data: Challenges and opportunities for semantic technologies. *IEEE Internet Computing*, 19(6):66–70, 2015.

3. John Davies and Mike Fisher. Internet of Things - Why Now? *Jnl Institute of Telecommunications Professionals*, 7(3), September 2015.
4. Giorgos Flouris, Irini Fundulaki, Maria Michou, and Grigoris Antoniou. Controlling access to rdf graphs. In *Future Internet Symposium*, pages 107–117. Springer, 2010.
5. Patrick Hayes. RDF Semantics. In *W3C Recommendation*, 2004.
6. Eric PrudHommeaux, Andy Seaborne, et al. SPARQL query language for RDF. *W3C recommendation*, 15, 2008.
7. Ilias Tachmazidis, Sotiris Batsakis, John Davies, Alistair Duke, Mauro Vallati, Grigoris Antoniou, and Sandra Stincic Clarke. A hypercat-enabled semantic internet of things data hub. In *The Semantic Web - 14th International Conference, ESWC 2017, Portorož, Slovenia, May 28 - June 1, 2017, Proceedings, Part II*, pages 125–137, 2017.
8. Ilias Tachmazidis, John Davies, Sotiris Batsakis, Grigoris Antoniou, Alistair Duke, and Sandra Stincic Clarke. Hypercat RDF: Semantic Enrichment for IoT. In *Semantic Technology - 6th Joint International Conference, JIST 2016, Singapore, Singapore, November 2-4, 2016, Revised Selected Papers*, pages 273–286, 2016.
9. Ilias Tachmazidis, Sotiris Batsakis John Davies, Alistair Duke, Mauro Vallati, Grigoris Antoniou, and Sandra Stincic Clarke. A Hypercat-enabled Semantic Internet of Things Data Hub: Technical Report. `https://arxiv.org/abs/1703.00391`, March 2017.
10. A.M. Townsend. *Smart Cities: Big Data, Civic Hackers, and the Quest for a New Utopia.* WW Norton & Company, 2013.