Doctoral Thesis

A Scalable Spatio-temporal Query Processing Engine for Linked Sensor Data

Hoan Nguyen Mau Quoc

February 25, 2020

Supervisors
Prof. John G. Breslin
Dr. Danh Le Phuoc

Insight Centre for Data Analytics
College of Engineering and Informatics, National University of Ireland, Galway

DECLARATION

I declare that this thesis, titled "*A Scalable Spatio-temporal Query Processing Engine for Linked Sensor Data*", is composed by myself, that the work contained herein is my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Galway, February 25, 2020

Hoan Nguyen Mau Quoc

"You can’t finish what you don’t start"

(Gary Ryan Blair)

ABSTRACT

The ever-increasing amount of Internet of Things (IoT) data emanating from sensors and mobile devices is creating new capabilities and unprecedented economic opportunity for individuals, organizations, and states. To fully realize the potential benefits of these sensor datasets, two fundamental requirements need to be addressed, namely interoperability and effective data management system. Fortunately, a suite of technologies developed in the Semantic Web effort, such as the RDF model, Linked Data, and SPARQL, can be used as some of the principal solutions to help sensor data from the challenge of poor interoperability. However, in this context, providing an effective data management system for sensor data that can combine the benefits of Semantic Web principles, and also be able to deal with the ”big spatio-temporal data” nature of sensor data, is still an open challenge. Central to this problem is not only knowing how to store a massive volume of sensor data, but also being able to answer a complex spatio-temporal related query on large-scale sensor datasets in a timely manner.

Researchers in the Semantic Web community have proposed a substantial number of works that use Semantic Web technologies for effectively managing and querying heterogeneous sensor data. However, our research survey revealed that these solutions primarily focused on semantic relationships and paid less attention to the temporal-spatial correlation of sensor data. Moreover, most semantic approaches do not have spatio-temporal support. Some of them have addressed limitations as regards providing full spatio-temporal support but have poor performance for complex spatio-temporal aggregate queries. In addition, while the volume of sensor data is rapidly growing, the challenge of querying and managing the massive volumes of data generated by sensing devices still remains unsolved.

In this work, we propose a scalable spatio-temporal query engine for sensor data based on Linked Data model, called EAGLE. The ultimate goal of our approach is to provide an elastic and scalable system which allows fast searching and analysis on the relationships of space, time and semantic in sensor data. In order to support spatio-temporal computing, we introduce a set of new query operators which is compatible with SPARQL 1.1. For dealing with ”big data” and a high update throughput of sensor data, EAGLE adopts a loosely hybrid architecture that consists of different clustered databases. This flexible architecture not only helps the engine with the overhead of ”big data” processing but also allows us to make use of the existing spatio-temporal query functions provided by the underlying databases. The engine also provides a learning optimization approach that can predict query performance based on historical query execution plans. To demonstrate the advantages of the query processing engine in terms of performance, the thesis provides extensive experimental evaluations. The evaluations cover a comprehensive set of parameters that indicate the performance of spatio-temporal queries over Linked Sensor Data.

ACKNOWLEDGEMENTS

This thesis becomes a reality with the kind support and help of many individuals. I would like to extend my sincere thanks to all of them.

Foremost, I would like to express my sincere gratitude to my two supervisors, Prof. John G. Breslin and Dr. Danh Le Phuoc for the continuous support of my Ph.D study and research. Their guidance helped me in all the time of research and writing of this thesis as well as all the articles that we published together. I could not have imagined having better supervisors for my Ph.D study.

Beside my supervisors, I would like to thank Prof. Manfred Hauswirth for offering me the internship opportunities in his group and leading me working on diverse exciting projects. I am very lucky to have his support in my career path.

My sincere thanks also goes to Dr. Martin Serrano, my unit leader, for his encouragement, insightful comments, and hard questions. Dr. Martin Serrano also has been a big supporter that provides me all resources I need during my Ph.D research.

I thanks to my colleges, Anh Le Tuan, for the stimulating discussions, for all the experiments that we were working together, and for the fun we have had in the last eight years.

My final words go equally to my family whom I want to thank for their unconditional love and guidance in whatever I pursue. Most importantly, thank to my wife, Huynh Quynh Hoa, for her inestimable moral support and her infinite warmth and tenderness. I also want to dedicate this thesis to my little princess, Anna, who always has been the motivation that keeps me going in my life.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

LIST OF ABBREVIATIONS

APIs	Application Programming Interfaces
ASCII	American Standard Code for Information Interchange
BGP	Basic Graph Pattern
CRS	Coordinate Reference System
CS	Coordinate System
DE-9IM	Dimensionally Extended 9-Intersection Model
ECMWF	The European Centre for Medium-Range Weather Forecasts
FTP	File Transfer Protocol
GeoSS	Global Earth Observation System of Systems
GML	Geography Markup Language
GoT	Graph of Things
GPS	Global Positioning System
HTTP	Hypertext Transfer Protocol
IoT	Internet of Things
IRI	Information Resource Identifier
ISH	Integrated Surface Hourly
LD	Linked Data
LOD	Linked Open Data
NCDC	National Climate Data Center
NOAA	National Oceanic and Atmospheric Administration
OGC	Open Geospatial Consortium
OWL	Web Ontology Language
RCC	Region Connection Calculus
RDF	Resource Description Framework
RDFS	RDF Schema
RSS	Really Simple Syndication
SOSA	Sensor, Observation, Sample, and Actuator
SPARQL	Simple Protocol and RDF Query Language
SSN	Semantic Sensor Network
SSW	Semantic Sensor Web
SW	Semantic Web
SWE	Sensor Web Enablement

URI	Uniform Resource Identifier
W3C	World Wide Web Consortium
WGS	World Geodetic System
WKT	Well-known Text

1 INTRODUCTION

1.1 MOTIVATION

The concept of the *"Internet of Things"*(IoT) was first introduced in 2005 in the annual report of the International Telecommunications Union. A common understanding of the IoT is a dynamic and global network infrastructure, in which each *thing* can be uniquely identifiable and be connectable through the Internet. Particularly, the IoT consists of a variety of information sensing devices, such as Radio Frequency Identification (RFID), sensor networks, infrared sensors, a global positioning system (GPS), laser scanners, and so on, and are able to connect to the Internet while making real-time observations about the world as it happens.

By 2020, with the estimate of having 50 billion connected objects [51], there will be an enormous amount of sensing observation data being continuously generated per second. This observation data source, in combination with existing data and services on the Internet, promises a wide range of innovative and valuable applications and services in the areas of Smart Cities, Smart Grid, Industry 4.0, Intelligent Transportation Systems, etc. Moreover, the variety and diversity of these sensor data sources enable the users to not only capture the trends but also to explore new knowledge spanning different scientific domains.

Interoperability is one of the most significant challenges in an IoT smart environment, where different platforms, sensors, and data sets are connected. While sensor data usually lack contextual information such as spatial information, measurement properties, temporal description of sensing data, etc., providing meaningful semantic metadata for sensor is necessary to achieve interoperability. Unfortunately, due to the heterogeneous nature of this sensing data, this task becomes more difficult and labor-intensive. To address this problem, the Semantic Web community has proposed the Linked Data model, for representing sensor data (so called Linked Sensor Data). This data model aims to connect dynamic and heterogeneous data generated from IoT, e.g. sensor readings, with any knowledge base to create a single graph as an integrated database serving any analytical queries on a set of nodes/edges of the graph [175, 40, 139, 110]. The crucial strength of the Linked Data model is in connectivity. Based on standard vocabularies and linking properties, data from different sources can be combined, linked, expanded and built upon to create information models for different domains. Since all data uses the same data model, they can be queried with the same query language (SPARQL). As a result, diverse data from different applications can be brought together easily.

In comparison with traditional data sources, and in combination with other useful information sources, the data generated by sensors are also providing a meaningful spatio-temporal context. They are produced in specific locations at a specific time. Basically, all sensor data items can

be represented in three dimensions: semantic, spatial and temporal. Consider the following real-world example. There is a global weather data provider that receives a request for sending the historical average temperature data of Dublin City in the last 10 years to a customer. To process this request, a query processing engine of the data provider has to execute an aggregate query on the temperature observation data generated by all weather stations in Dublin City. In this query, the implied semantic description of sensor data is temperature measurement. In the meantime, the spatial information describes a place (Dublin city) that the sensors observe. Finally, the temporal dimension specifies the *time* when the temperature values were generated (last 10 years). Apparently, to answer the aforementioned query in a timely manner, the query engine has to have a capability for processing a heavy spatio-temporal computation and also querying the semantic description of sensor data. Unfortunately, in the Linked Sensor Data context, supporting such multidimensional analytical queries is still not fully addressed. This is due to the fact that researchers in this area have paid more attention to processing the semantic relationships of the data rather than investigating a sufficient approach for querying its spatio-temporal correlation. As a consequence, it is difficult for current solutions to efficiently index and query sensor data.

Besides the need to support spatio-temporal computation, it is also imperative to produce an efficient and scalable data processing engine so that it can deal with the "big data" nature of sensor data. This requirement is crucial because sensor data is frequently updated and one can end up with a massive amount of data. Several standalone spatial RDF stores have been introduced for managing sensor data, however, as the size of data increases, such single-machine approaches meet performance bottlenecks, in terms of both data loading and query performance. Other attempts have been proposed [130, 167, 166] for distributed processing of RDF data, however, these are not designed for handling spatial or spatio-temporal data.

These facts motivated us to develop EAGLE, a scalable spatio-temporal query engine for processing Linked Sensor Data. Our approach aims to build a high-performance processing engine for Linked Sensor Data which is able to index, filter and aggregate a high throughput of sensor data together with a large volume of historical data stored in the engine. Following this aim, the EAGLE engine adopts a loosely hybrid architecture that consists of different clustered databases. This flexible architecture not only helps the engine to deal with the overhead of "big data" processing but also allows us to make use of existing spatio-temporal query functions provided by the underlying databases. It is also worth mentioning that while there can be different kinds of spatio-temporal queries (continuous queries, queries on moving objects, queries on historical data, etc), EAGLE focuses only on solving the processing challenges of the queries on historical data. In particular, this is based on the assumption that the spatial information of the sensor is static while the temporal observation data of the sensor is frequently updated. In that sense, for supporting spatio-temporal computing, we introduce a set of new query operators which is compatible with SPARQL 1.1. Furthermore, to provide an insight into how to build an efficient query processing engine for Linked Sensor Data, we conducted the first performance study on Linked Sensor Data processing engines that were developed during the time frame of this thesis.

1.2 PROBLEM STATEMENT

The motivation for the thesis leads to the broader research problems that arise when building an efficient scalable spatio-temporal query processing engine for Linked Sensor Data. The first challenge is to define a unified data model that semantically describes the three aspects of sensor data (spatial, temporal and semantic). Furthermore, the data model should be able to represent not only Linked Sensor Data but also Linked Data in a unified view. For this reason, it is desirable that the spatio-temporal data model should be extended from the Linked Data model so that it can be transparently integrated with other conventional RDF datasets.

Along with the spatio-temporal data model, there is a need for an associated semantic query language that supports spatio-temporal searches. It is important to note that the standard SPARQL query language is only designed for querying semantic RDF data and thus has very limited support for spatio-temporal queries. Therefore, when defining a spatio-temporal query language, the spatial and temporal query operators associated with their special variables have to be defined to specify the meanings of the declarative query patterns. Additionally, because temporal analytics is one of the most popular demands in a Linked Sensor Data context, one feature that should be taken into consideration is having a set of temporal analytical query operators that helps the complex temporal aggregate queries to be easily constructed. Finally, to reduce the learning effort, it is also desirable that the proposed query language is a SPARQL-like query language [140].

During the evolution of RDF management systems in the last decade, many approaches for efficiently managing and querying RDF data have been proposed [3, 2, 30, 133, 196, 30, 199]. In these approaches, the RDF data is well organized and indexed to efficiently and effectively answer the RDF queries. However, since these systems are already well-designed and none of them takes spatio-temporal support into consideration, it is difficult to use them as a Linked Sensor Data management system without great modification. For example, in [29], the authors exploit RDF-3X [131] to build a spatial feature integrated query system. They employ the separated R-tree and RDF-3X indexes for filtering the entities exploiting the spatial and semantic features respectively. Similarly, YAGO2 [113] is extended to support spatial feature over statements and also provides an interface for querying SPARQL-like queries over YAGO2 data. Because such approaches are not intended to be a geographical or temporal database in the first place, they limit the user to use very few hard-code spatial relationships, such as *north_Of*, *east_Of*, *south_Of*, etc, and do not support other complex ones (i.e, *within*, *intersects*). For a temporal query, instead of building a dedicated temporal index, a special handling method is introduced. For that, temporal data are treated as standard RDF literals. To query this data, the temporal filter will be defined in the standard semantic SPARQL FILTER expression, using the basic comparison operators on the time values. This solution is proved to be only effective in the case of simple temporal queries that can be processed by SPARQL. However, for the more complex analytical spatio-temopral queries such as an aggregate query over an explicit spatio-temporal correlation, this solution will be inappropriate due to its poor query performance. Therefore, instead of indexing temporal values as RDF literals, a more sufficient spatio-temporal index solution needs to be investigated.

The aforementioned approaches commonly focus on enabling spatio-temporal query features, but hardly any of them fully address the performance and scalability issues of querying billions of data point. On the light of dealing with this "big data" issue, many centralized and distributed native RDF repositories have been implemented [130, 165, 167]. These RDF repositories are fast and able to scale up to many millions of triples or a few billion triples. Unfortunately, none of these systems takes spatio-temporal feature into consideration. Other hybrid solutions have been proposed such as Strabon [108], SSTDE [201], to support spatio-temporal queries at scale. These approaches adopt a loosely coupled hybrid architecture which includes different relational DBMSs coordinated by a centralized middleware. However, through our performance study, some shortcomings in system performance have been realized, i.e, data loading and analytical query performance. We address this issue by proposing a more efficient hybrid distributed architecture that is not only able to support complicated spatio-temporal queries tailored to managing Linked Sensor Data but also that is capable of dealing with mentioned performance and scalability issues.

Lying at the heart of the query processing engine is the query optimizer. Researchers in the Semantic Web community have proposed a substantial number of works that focus on query optimization in conjunction with supporting spatio-temporal computation. In many of these approaches, for a given query, the query optimizer will firstly analyze the correlation of the spatial, temporal and semantic aspects, and find the best execution plan based on either the complex cost model or by using data statistics and heuristics. However, these approaches have only proven to be effective in a centralized system where the optimizer can easily estimate the cost of all spatio-temporal query execution operations. In the context of a Linked Sensor Data management system, which requires a distributed integration spatio-temporal query processing engine, these existing query optimization techniques will not be sufficient because of the following reasons: (1) defining a cost model which has to express all spatio-temporal aspects of sensor data and is adaptable to changes in the underlying environment (data communication costs, characteristics of the network, cluster configuration, etc.) is a costly task and has not been fully addressed; (2) the spatio-temporal statistics of sensor datasets are often missing due to the (multidimensional) complexity and high-frequency updates of the data. They are expensive to generate and maintain. As coming up with a reliable analytical cost-model is very difficult and often impossible to achieve, a learning query optimizer should be considered.

1.3 THESIS CONTRIBUTIONS

This thesis aims to develop a scalable spatio-temporal query processing engine for Linked Sensor Data. During this process, a number of original contributions were produced as follows.

A performance study of RDF stores for Linked Sensor Data

Before the design of our EAGLE engine, we firstly study and analyze in detail the fundamental requirements for an RDF processing engine that can be applied to Linked Sensor Data. Based

on these requirements, we conduct an extensive performance assessment of five selected and well-known RDF stores for managing Linked Sensor Data. In comparison with existing works, our performance study also focuses on evaluating the spatial, temporal and text data indexing performance of these systems. These evaluations give an insight into the gaps and shortcomings of current RDF stores in managing Linked Sensor Data and so help in the design of our approach. Furthermore, through the performance study, we also identify a query optimization challenge for executing a complex spatio-temporal query over Linked Sensor Data.

Data modeling and publishing process of linked meteorological sensor dataset

The lack of research in the area of spatio-temporal Linked Data will draw more attention when more datasets with such characteristics will be made available in the LOD Cloud. Taking this target into consideration, in this thesis, we present the sensor data modeling and publishing process of our linked meteorological sensor dataset. This RDF spatio-temporal sensor dataset is transformed from Integrated Surface Hour (ISH) weather data, which is originally published by The National Oceanic and Atmospheric Administration. Thanks to Linked Data principles, we were able to transform roughly 350 gigabytes of ISH raw weather data to RDF. This dataset consists of roughly 3.7 billion global meteorological sensor observations from 2008 to 2018. As the data also consists of geospatial and temporal information, making all this data available as Linked Data and interlinking it with semantic connections will provide extremely useful links to the Linked Data Cloud and also with substantial environmental and commercial value.

A SPARQL spatio-temporal extensions for querying Linked Sensor Data

We propose our SPARQL query language extensions to enable querying spatial and temporal aspects of Linked Sensor Data. We adopt the GeoSPARQL syntax [19] into our proposed extensions so that it can be used to query spatial data. For querying temporal data, we introduce a set of novel analytical temporal property functions. Moreover, a full-text search function is also given.

Scalable spatio-temporal query processing engine

We develop EAGLE, a scalable spatio-temporal query processing engine, that is able to store a massive amount of sensor data and also efficiently query them using our proposed query language. The engine serves as a middleware to maintain the hybrid database systems and consists of an implementation of the proposed spatio-temporal query operators. The engine is developed upon Apache Jena, allowing it to manage the semantic data as a native triple store. For the spatial, temporal and text data, the engine stores and maintains them externally in different underlying databases. This architecture demonstrates excellent scalability for very large Linked Sensor Datasets.

Spatio-temporal data partitioning strategy

We propose a spatio-temporal data partitioning strategy that aims to enhance the data loading and query performance of EAGLE. The partitioning strategy makes use of the spatio-temporal correlation features of sensor data so that all the observation data of nearby sensors will be located in the same data partitions and ordered by time. We implement this technique by designing a *rowkey scheme* that concatenates the geohash prefix representing a sensor's location and its temporal and semantic information. Each data partition is assigned to a unique range of rowkey. Therefore, based on our spatio-temporal data partitioning strategy, for a given spatio-temporal query, the query engine can locate quickly what partitions actually have to be processed, thus, reducing the query processing time drastically.

A learning approach for query planning on spatio-temporal Linked Sensor Data

Generating execution plans for a given query is a costly operation for the query engine. An interesting alternative to this operation is to reuse the old execution plans that were already generated by the optimizer for past queries, to execute a new query. In this thesis, we present a learning approach for query planning that uses query similarity identification in conjunction with machine learning techniques to recommend a previously generated query plan to the optimizer for a given query. We present how to model the spatio-temporal query features as feature vectors for machine learning algorithms such as the k-nearest neighbors algorithm (k-NN) and support vector machine (SVM). Additionally, our approach also aims to predict the query execution time for the purposes of workload management and capacity planning. Our extensive experiments indicate the efficiency of our learning approach with an impressive prediction accuracy on test queries.

1.4 ILLUSTRATIVE EXAMPLE

We will give a brief introduction to our querying approach to illustrate some of our query engine's major concepts. This example is taken from a scenario that The European Centre for Medium-Range Weather Forecasts (ECMWF) is being requested from the tourist about a list of hottest places in Europe during the summer of 2019. This statistical information aims to help tourists planning their coming summer vacation.

In this example, to answer the user request, the analyst at the ECMWF may query information about all visited places in Europe and then searches for the relationships connecting these places to their meteorological stations. These relationships can be found by matching the visited places coordinates with the known location of ECMWF stations. We may pose the following spatial SPARQL query involving the *geo:sfWithin* spatial function for such a search. The namespaces can be found in Listing A of Appendix A.

Thanks to Linked Data principles that allows interlinking the sensor dataset to external information sources such as DBPedia, the analyst is able to query all the visited places in Europe via a

```
SELECT ?attractions ?sensorLocation
WHERE{
    ?entity skos:broader dbr:Category:Tourism_by_city .     (1)
    ?places skos:broader ?entity .                          (2)
    ?attractions dcterms:subject ?places .                  (3)
    ?attractions dbo:country ?country .                     (4)
    ?country dbo:location dbr:Europe.                       (5)
    ?attractions geo:lat ?lat.                              (6)
    ?attractions geo:long ?long.                            (7)
    ?attractions geo:hasGeometry ?attrLoc.                  (8)
    ?station a got:WeatherStation.                          (9)
    ?station geo:hasGeometry ?geoFeature.                   (10)
    ?geoFeature geo:sfWinthin(?attrLoc 50 "miles").         (11)
}
```

Listing 1.1: Spatial query for retrieving the list of visited place in Europe.

set of triple patterns (pattern 1 to pattern 8) that query across the DBPedia datasets. To search for the geographical relationship connecting the retrieved visited places and all meteorological stations nearby through their known locations (retrieved by executing query patterns 9 and 10), we are using the spatial function *geo:sfWithin* (in pattern 11). The spatial function also allows the analyst to additionally limit the area that the stations located in (i.e, within 50 miles).

After specifying the spatial relationship between the visited places and sensor stations, the analyst might be interested in a measurement property (i.e, temperature) and the time constraint of the user request (i.e, summer 2019). Therefore, we define the semantic relationship connecting the station and the sensor measurement property via patterns 12 to 16. Next, analytical temporal functions are introduced. The temporal functions will help the analyst to specify the type of statistics (i.e, *temporal:avg*) on the temporal properties (i.e, *observation value*) of a given semantic relationship (observation value and the visited place that the sensor observes). The complete corresponding query is shown below:

```
SELECT ?attractions ?sensorLocation
WHERE{
    ?entity skos:broader dbr:Category:Tourism_by_city .     (1)
    ?places skos:broader ?entity .                          (2)
    ?attractions dcterms:subject ?places .                  (3)
    ?attractions dbo:country ?country .                     (4)
    ?country dbo:location dbr:Europe.                       (5)
    ?attractions geo:lat ?lat.                              (6)
    ?attractions geo:long ?long.                            (7)
    ?attractions geo:hasGeometry ?attrLoc.                  (8)
    ?station a got:WeatherStation.                          (9)
    ?station geo:hasGeometry ?geoFeature.                   (10)
    ?geoFeature geo:sfWinthin(?attrLoc 50 "miles").         (11)
    ?sensor a sosa:Sensor;                                  (12)
            sosa:isHostedBy ?station;                       (13)
            sosa:observes got:AirTemperature.               (14)
    ?obs sosa:madeBySensor  ?sensor;                        (15)
         sosa:hasSimpleResult ?value.                       (16)
```

```
    ?value temporal:avg("01/01/2019" "31/03/2019")          (17)
}
GROUP BY (?attractions)
ORDER BY ?value limit 10
```

Listing 1.2: A corresponding spatio-temporal query of our illustrative example.

As illustrated in the query above, EAGLE engine allows multiple operators to be used in a single query. We can therefore execute a spatio-temporal-semantic queries that combine spatial and temporal functions. The query language and query execution process are discussed in Chapter 6.

1.5 PUBLICATIONS

Several parts of this thesis have been published as conference, workshop and journal articles as follows:

Published

- Quoc, Hoan Nguyen Mau, Martin Serrano, John G. Breslin, and Danh Le Phuoc. "A learning approach for query planning on spatio-temporal IoT data." In Proceedings of the 8th International Conference on the Internet of Things, p. 1. ACM, 2018. **Best Paper Award**.
- Quoc, Hoan Nguyen Mau, and Danh Le Phuoc. "An elastic and scalable spatio-temporal query processing for Linked Sensor Data." In Proceedings of the 11th International Conference on Semantic Systems, pp. 17-24. ACM, 2015.
- Quoc, Hoan Nguyen Mau, Martin Serrano, Danh Le-Phuoc, and Manfred Hauswirth. "Super stream collider–linked stream mashups for everyone." Proceedings of the Semantic Web Challenge at ISWC2012 (2012).
- Le-Phuoc, Danh, Hoan Nguyen Mau Quoc, Hung Ngo Quoc, Tuan Tran Nhat, and Manfred Hauswirth. "The Graph of Things: A step towards the Live Knowledge Graph of connected things." Journal of Web Semantics 37 (2016): 25-35.
- Le-Phuoc, Danh, Hoan Nguyen Mau Quoc, Quoc Hung Ngo, Tuan Tran Nhat, and Manfred Hauswirth. "Enabling live exploration on the graph of things." Proceedings of the Semantic Web Challenge (2014).
- Le-Phuoc, Danh, Hoan Nguyen Mau Quoc, Chan Le Van, and Manfred Hauswirth. "Elastic and scalable processing of linked stream data in the cloud." In International Semantic Web Conference, pp. 280-297. Springer, Berlin, Heidelberg, 2013.
- Le-Phuoc, Danh, Hoan Quoc Nguyen-Mau, Josiane Xavier Parreira, and Manfred Hauswirth. "A middleware framework for scalable management of linked streams." Web Semantics: Science, Services and Agents on the World Wide Web 16 (2012): 42-51.

- Le-Phuoc, Danh, Hoan Nguyen Mau Quoc, Josiane Xavier Parreira, and Manfred Hauswirth. "The linked sensor middleware–connecting the real world and the semantic web." Proceedings of the Semantic Web Challenge (2011): 22-23.

In addition, some parts of this work are relevant in project publications as follows:

- Jayaraman, Prem Prakash, Jean-Paul Calbimonte, and Hoan Nguyen Mau Quoc. "The schema editor of openiot for semantic sensor networks." arXiv preprint arXiv:1606.06434 (2016).

- Serrano, Martin, Hoan Nguyen Mau Quoc, Danh Le Phuoc, Manfred Hauswirth, John Soldatos, Nikos Kefalakis, Prem Prakash Jayaraman, and Arkady Zaslavsky. "Defining the stack for service delivery models and interoperability in the internet of things: a practical case with OpenIoT-VDK." IEEE Journal on Selected Areas in Communications 33, no. 4 (2015): 676-689.

- Saniat, Mahmudur Rahman, Hoan Nguyen Mau Quoc, Huy Le Van, Danh Le Phuoc, Martin Serrano, and Manfred Hauswirth. "Autonomic Frameworks Deployment Using Configuration and Service Delivery Models for the Internet of Things." In Interoperability and Open-Source Solutions for the Internet of Things, pp. 89-102. Springer, Cham, 2015.

- Serrano, Martin, Hoan Nguyen M. Quoc, Manfred Hauswirth, Wei Wang, Payam Barnaghi, and Philippe Cousin. "Open services for IoT cloud applications in the future internet." In 2013 IEEE 14th International Symposium on" A World of Wireless, Mobile and Multimedia Networks"(WoWMoM), pp. 1-6. IEEE, 2013.

- Le Phuoc, Danh, Martin Serrano, Hoan Nguyen Mau Quoc, and Manfred Hauswirth. "A complete stack for building Web-of-Things Applications." submitted to the IEEE Computer Systems Journal, Special issue on the Web-of-Things 2013 (2013).

In Submission

- A Performance Study of RDF Stores for Linked Sensor Data. Hoan Nguyen Mau Quoc, Martin Serrano, Han Nguyen Mau, John G. Breslin, Danh Le Phuoc. Submitted to Semantic Web Journal, July 2019.

- A Scalable Query Processing Engine for Linked Sensor Data. Hoan Nguyen Mau Quoc, Martin Serrano, Han Nguyen Mau, John G. Breslin, Danh Le Phuoc. Submitted to Sensors Journal, Special Issue on Semantics for Sensors, Networks and Things, Aug 2019.

1.6 THESIS OUTLINE

The remainder of the thesis is organized as follows:

- **Chapter 2** presents the preliminaries for this research, which involve SW, RDF engines, and spatio-temporal data representation. It defines the terminologies and notations used in the architectural design of the EAGLE engine.
- **Chapter 3** summarizes the related work regarding research challenges mentioned in Section 1.2. The first section presents the data modeling for Linked Sensor Data, followed by a discussion about the achievements and limitations of current approaches. Existing spatio-temporal query languages are also reviewed. In the second section, the works regarding spatio-temporal query processing engines are reported.
- **Chapter 4** describes our linked meteorological data publishing process. Through this, we also present a data modeling approach that is used to semantically describe Linked Sensor Data. The summary of the linked meteorological dataset and its application are given at the end. This work is under review in [157].
- **Chapter 5** reports our performance study of RDF stores for Linked Sensor Data. In this chapter, we present the fundamental requirements and analyze the general architectural design of RDF stores that can be used for managing Linked Sensor Data. Extensive performance comparison of the most popular RDF stores that support spatio-temporal query is also presented. A discussion and our main findings are given at the end of this chapter. This work is under review in [156].
- **Chapter 6** describes our prototype EAGLE's implementation with the goal of creating a scalable spatio-temporal query processing engine for Linked Sensor Data. Topics covered include system design and architecture, data partitioning strategy, query language, and interactivity components. In addition to that, a system performance evaluation of EAGLE is also given at the end of this chapter. This work is published in [154, 146].
- **Chapter 7** presents the query optimization techniques that we have developed in EAGLE to increase its performance. In this chapter, we first discuss the limitations of traditional approaches and then propose a query optimization method that applies machine learning algorithms. Additionally, we also discuss how to model spatio-temporal query features as feature vectors for machine learning algorithms. Finally, a detailed experiment is reported. This work is published in [155].
- **Chapter 8** summarizes this research, draws conclusions and lists further potential research topics.

2 BACKGROUND

This chapter provides the background concepts and terminologies that enable us to achieve the goals. Firstly, we will discuss the relevant notations and definitions of the Semantic Web, RDF data model, ontology and the SPARQL query language. We also give an overview of the RDF store. Next, we present some preliminary knowledge on the representation of geospatial and temporal information and focus on how this representation is adopted in the RDF model.

2.1 SEMANTIC WEB AND LINKED DATA

The Semantic Web (SW), which is considered as an extension of the current World Wide Web, has drawn a lot of attention recently. Its vision promises an extension of the current web in which data is accompanied with machine-understandable metadata allowing capabilities for a much higher degree of automation and more intelligent applications [22].

This web is founded on the concept of Linked Data (LD), a term used to refers to a set of best practices – proposed by Berners-Lee in his Web architecture [21] – for publishing and interlinking data on the Web [82]. It builds upon standard Web technologies such as HTTP, RDF, and URI, but rather than using them to serve web pages only for human readers, it extends them to share information in a way that can be read automatically by computers. Therefore, LD has quickly become the dominant solution for cross-database data integration. As shown in Figure 2.1, there has already been massive amounts of data from different domains interlinked with each other and compose a large data cloud. According to the recent LOD reports [10, 168], this cloud has consisted of more than 1,234 data sources with 16,136 linked covering many well-known areas, such as general knowledge (DBpedia [15]), bioinformatics (Uniprot [193]), GIS (Geoname [198], linkedgeodata [185]) and web-page annotations (e.g, RDFa [4], microformats [99]). In addition, LD is also a most recommended data model among governments and enterprises that see RDF as a more flexible way to represent their data. For example, this includes the US government (data.gov), the UK (data.gov.uk) as well as Google, Bing, etc.

In this thesis, we use various LD technologies that enable us to achieve our goals. In the following subsections, we briefly introduce the core concepts of LD, which are ontology and RDF data model, for representing data on the SW. After that, we will discuss the SPARQL query language to query data on the SW and the RDF engine for storing LD. Finally, the LD principles are elaborated. For a more detailed introduction to RDF and SPARQL, we refer the readers to the W3C specification documents [45, 53].

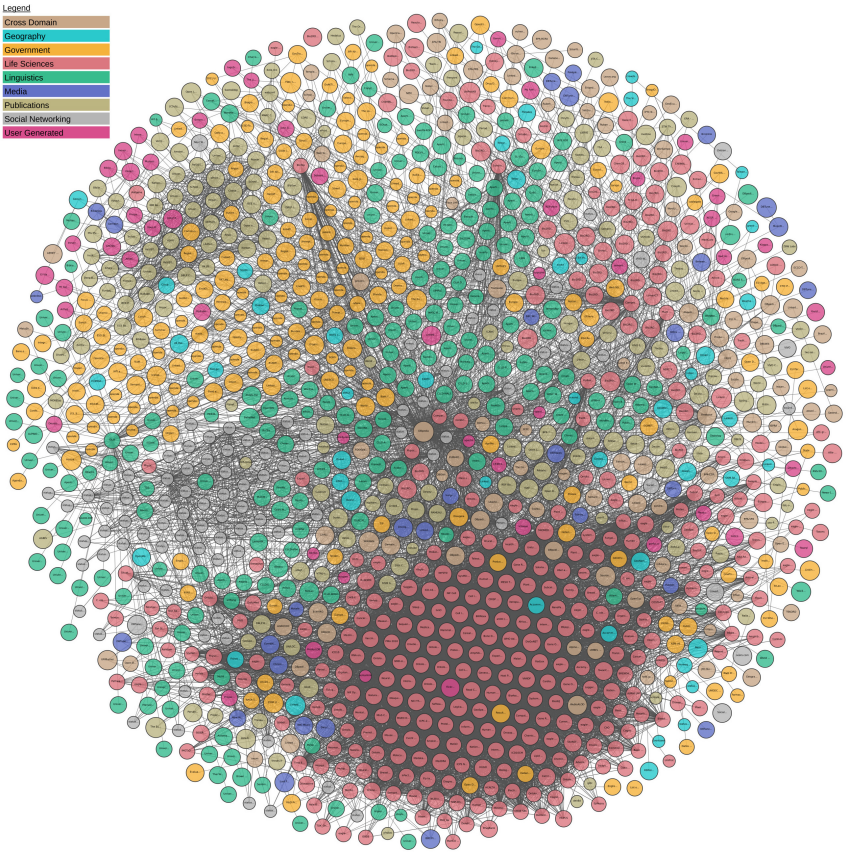


Figure 2.1: Linked Open Data diagram [191]

2.1.1 Ontology

Ontology is the key element of the SW. It formally defines the concepts, relationships and provides the schema to create the semantic metadata for objects [67]. In particular, an ontology describes the following aspects:

- **Individuals (or instances)**: Individuals are the basic and fundamental elements in an ontology. The individuals in an ontology may include concrete objects (people, animals, tables, automobiles, etc), as well as abstract individuals (numbers, words).
- **Classes**: classes are an abstraction of objects. Classes may classify individuals, other classes, or a combination of both.
- **Attributes**: objects in an ontology can be described by assigning attribute values to them. An attribute contains at least one name and one value, which store specific information of an object.
- **Relationships**: Relationships between objects in an ontology specify how an object is related to other objects. Usually, a relation is an attribute which has another object in the ontology as its value. In general, the set of relationships describes the whole semantic in a domain.
- **Events**: events are objects about time or instantiated object resources.

To build an ontology, the W3C[1] develops the Web Ontology Language (OWL) [122]. There are three subsets of OWL: Lite, DL and Full. OWL Lite is the least expressive one, a subset of OWL DL. It assures efficient reasoning by reducing axiom constraints in OWL DL. OWL DL (Description Logic) contains all elements of OWL, but is restrictively used. OWL DL provides reasoning functions in description logic, which is formalized on the basis of OWL. OWL Full contains all elements of OWL with no restriction. It extends the RDFS to a complete the ontology language, and is suitable for users who need no computational guarantees while the best expressive power with no limit.

2.1.2 Resource Description Framework

The Resource Description Framework (RDF) [104] has been adopted by the W3C as a standard for representing metadata on the Web. In the following, we introduce the definitions related to the RDF data model. Let I be the set of Information Resource Identifiers (IRI), L be the set of literals, and B be the set of blank nodes.

Definition 2.1 ***(RDF terms)*** *The set of RDF Terms RT is:* $RT = I \cup B \cup L$.

Definition 2.2 ***(RDF triple)*** *an RDF triple (s,p,o) is a member of the set* $(I \cup B) \times I \times (I \cup L \cup B)$. *For an RDF triple (s,p,o), the element s is called subject, the element p is called predicate, and the element o is called object.*

Definition 2.3 ***(RDF Data set)*** *An RDF dataset is a set DS:* $DS = \{(s_1, p_1, o_1), ..., (s_n, p_n, o_n)\}$.

An example of RDF triples is shown as Table 2.1. There, the statements convey the information that the *sensor A* is an air temperature sensor, while the two others present the sensor observation value and its observed time.

Table 2.1: An example of RDF triples

Triples	
(1)	<:_sensorA> <sosa:observes> <iot:AirTemperature>
(2)	<:_obs> <sosa:madeBySensor> <:_sensorA>
(3)	<:_obs> <sosa:hasSimpleResult> 27
(4)	<:_obs> <sosa:resultTime> "12-08-2018T00:00:00"

A set of RDF triples is referred as a directed, labeled RDF Graph [172], where a directed edge labeled with the property name connects a vertex labeled with the subject name and a vertex labeled with the object name. RDF Schema (RDFS) [121] provides a standard vocabulary for describing the classes and relationships used in RDF graphs and consequently provides the capability to define ontologies. An example of RDF graph is shown in Figure 2.2.

2.1.3 SPARQL Language

SPARQL is the standard RDF query language that provides facilities to extract information from RDF data. The detailed query syntax is defined by the W3C [195]. Similar to a RDF graph,

1 https://www.w3.org/

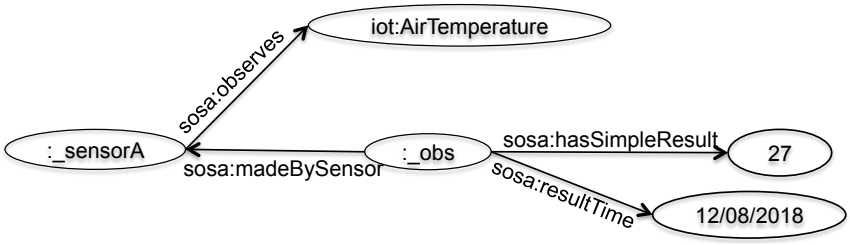


Figure 2.2: An example of RDF graph

the core component of SPARQL queries is a conjunctive set of triple patterns. The definition of a triple pattern is the same as a RDF triple, which is in the form of subject-predicate-object. However, the difference is that any component of the pattern could be a variable. A triple pattern could match a subset of the underlying RDF data, where the terms in the triple pattern respond to the ones in the RDF data [66]. A solution mapping is defined as the mapping from the variables to the responsible RDF terms. In the following, we present some important formal definitions from SPARQL query language which are used throughout the remainder of this thesis.

Definition 2.4 ***(SPARQL abstract query)*** *A SPARQL abstract query is a tuple (E, DS, R) where E is a SPARQL algebra expression, DS is an RDF dataset, R is a query form.*

In the Definition 2.4, the algebra expression E is a set of query operators which is evaluated against the RDF data set DS. The algebra expression E consists of graph patterns and operators such as *FILTER*, *JOIN*, and *ORDER BY*. The query form R uses the solutions from pattern matching to form result sets or RDF graphs. The query forms are SELECT, CONSTRUCT, ASK and DESCRIBE. The triple pattern, basic graph patterns and pattern matching are defined as follows:

Definition 2.5 ***(Triple Pattern)*** *A triple pattern is a member of the set:* $(T \cup V) \times (I \cup V) \times (T \cup V)$. *The set of RDF terms T is the set* $I \cup V \cup B$. *The set V is the set of query variables where V is infinite and disjoint from T.*

Definition 2.6 ***(Basic Graph Pattern)*** *A Basic Graph Pattern (BGP) expression is defined recursively as follows:*

1. *A tuple from* $(I \cup V \cup B) \times (I \cup V) \times (I \cup L \cup V \cup B)$ *is a graph pattern.*
2. *The expressions* $(P_1$ *AND* $P_2)$, $(P_1$ *OPTIONAL* $P_2)$ *and* $(P_1$ *UNION* $P_2)$ *are graph patterns if* P_1 *and* P_2 *are graph patterns.*
3. *The expression (P FILTER R) is a graph pattern if P is a graph pattern and R is a SPARQL built-in condition. The SPARQL built-in condition is defined in Definition 2.7.*

Definition 2.7 ***(SPARQL built-in condition)*** *A SPARQL built-in condition is constructed using elements of the set* $(I \cup L \cup V$ *and constants), logical connectives* $(\neg, \wedge, \vee)$, *inequality symbols* $(<, \leqslant, >, \geqslant)$, *the equality symbol* $(=)$, *and unary predicates like* ***bound***, ***isBlank***, *and* ***isIRI***, *etc. A built-in condition is a Boolean combination of terms constructed by using = and bound as follows:*

1. *If* $?X, ?Y \in V$ *and* $c \in I \cup L$, *then bound(?X),* $?X = c$ *and* $?X = ?Y$ *are built-in conditions.*
2. *If* R_1 *and* R_2 *are built-in conditions, then* $(\neg R_1)$, $(R_1 \vee R_2)$ *and* $(R_1 \wedge R_2)$ *are built-in conditions*

Definition 2.8 ***(Filter Expression)*** *Given a mapping μ and a build-in condition R, μ is said to be satisfied R, denoted by $\mu \models R$, if:*

1. *R is bound(X) and $X \in dom(\mu)$.*
2. *R is ?X =c, $?X \in dom(\mu)$ and $\mu(?X) = c$.*
3. *R is ?X = ?Y, $?X \in dom(\mu)$, $?Y \in dom(\mu)$ and $\mu(?X) = \mu(?Y)$.*
4. *R is $(\neg R_1)$, R_1 is a built-in condition, and it is not the case that $\mu \models R_1$.*
5. *R is $(R_1 \wedge R_2)$, R_1 and R_2 are built-in conditions, and $\mu \models R_1$ or $\mu \models R_2$.*
6. *R is $(R_1 \vee R_2)$, R_1 and R_2 are built-in conditions, and $\mu \models R_1$ or $\mu \models R_2$.*

The semantics of SPARQL is defined via mappings. A mapping μ is defined as:

Definition 2.9 ***(Solution Mapping)*** *A solution mapping μ from V to RT is a partial function $\mu : V \to RT$ where $RT = (I \cup B \cup L)$ and defined for a finite subset of variables V. RT is the RDF term also defined in Definition 2.1.*

Definition 2.10 ***(Triple Pattern Matching)*** *Let DS be a data set with set of RDF terms RT, and t a triple pattern of a SPARQL query. If P is a SPARQL graph pattern then var(P) is the set of variables occurring in P. $dom(\mu)$ is the domain of μ which is the subset of V where μ is defined. The evaluation of t over DS, denoted by $[[t]]_{DS}$ is defined as the set of mappings [164]:*

$$[[t]]_{DS} = \{\mu : V \to RT \mid dom(\mu) = var(P) \text{ and } \mu(P) \subseteq DS\}$$

If $\mu \in [[t]]_{DS}$, it can be said that μ is a solution for t in DS. If a data set DS has at least one solution for a triple pattern t, then one can say DS matches t.

For composing a query, a series of relational operators on sets of mappings are provided such as join($\bowtie$), union($\cup$), minus($\setminus$) and left outer join($\mathbin{⟕}$).

$$\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1 \wedge \mu_2 \in \Omega_2 \wedge \mu_1 \cong \mu_2\}$$
$$\Omega_1 \cup \Omega_2 = \{\mu \mid \mu_1 \in \Omega_1 \vee \mu_2 \in \Omega_2\}$$
$$\Omega_1 \setminus \Omega_2 = \{\mu \in \Omega_1 \mid \neg\exists \mu' \in \Omega_2, \mu' \cong \mu\}$$
$$\Omega_1 \mathbin{⟕} \Omega_2 = (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \setminus \Omega_2)$$

In the following, we will look in more detail at the structure of SPARQL. Figure 2.3 illustrates the SPARQL query language structure.

- PREFIX: this key word is used to declare the abbreviations for the namespaces used in a query.
- SELECT: the SELECT result clause returns a list of specific variables and values that satisfy the query pattern. The SELECT * is to select all the variables in a query.
- FROM: Defines the data set to be queried. If no dataset is specified, the default dataset is selected.
- CONSTRUCT: The CONSTRUCT query form returns a single RDF graph specified by a graph pattern.

Prologue (Optional)	BASE <uri> PREFIX prefix:<uri>
Query Form (Required)	SELECT (DISTINCT) ?var_1 ?var_2 … ?var_n SELECT (DISTINCT) * DESCRIBE ?var_1 ?var_2 … ?var_n or <uri> DESCRIBE * CONSTRUCT { Graph Pattern } ASK
RDF Dataset (Optional)	FROM <uri> FROM NAMED <uri>
Graph Pattern (Optional, Required for ASK)	WHERE { Graph pattern [Filter Expression]}
Solution Sequences Ordering (Optional)	ORDER BY
Solution Sequences Modifiers (Optional)	LIMIT n, OFFSET m

Figure 2.3: SPARQL query language structure

- DESCRIBE: It is used to return a RDF graph describing the resources that were found.
- ASK: It returns a Boolean value indicating whether or not a query pattern has a solution.
- WHERE: This clause provides the basic graph pattern to match against the data graph. The WHERE clause may include optional triples. If the triple to be matched is optional, it is evaluated when it is present, but the matching does not fail when it is not present. Also, it is possible to make UNION of multiple matching graphs - if any of the graphs matches, the match will be returned as a result.
- FILTER: In addition to specifying graphs to be matched in the WHERE clause, constraints can be added for values using the FILTER construct. By using this construct we can apply all kinds of value restrictions such as string value restrictions (like FILTER regex(?name, "Dublin") meaning that the ?name variable must contain the string "Dublin" as a substring) or number value restrictions (like FILTER (year(?time) < 2018) meaning the year of ?time must be less than 2018). Also, a few special operators are defined for the FILTER construct. These include the "isIRI" operator for testing whether a variable is an IRI/URI, the "isLiteral" operator for testing whether a variable is a literal and "bound" to test whether a variable is bound to other variables.
- DISTINCT: Used to distinguish the unique results.
- ORDER BY: The ORDER BY clause establishes the order of a solution sequence. Following the ORDER BY clause is a sequence of order comparators, composed of an expression and an optional order modifier (either *ASC()* or *DESC()*). Each ordering comparator is either ascending (indicated by the *ASC()* modifier or by no modifier) or descending (indicated by the *DESC()* modifier).
- LIMIT: The LIMIT clause puts an upper bound on the number of solutions returned. If the number of actual solutions is greater than the limit, then at most the limit number of solutions will be returned.

- OFFSET: OFFSET causes the solutions generated to start after the specified number of solutions. An OFFSET of zero has no effect. Using LIMIT and OFFSET to select different subsets of the query solutions will not be useful unless the order is made predictable by using ORDER BY.

An example of SPARQL query is shown in Listing 2.1:

```
prefix sosa: <http://www.w3.org/ns/sosa/>
prefix iot: <http://iotschema.org/>
SELECT ?sensor ?sensorLocation
WHERE{
?sensor a sosa:Sensor;
        sosa:isHostedBy ?sensorLocation;
        sosa:observes iot:AirTemperature.
}
```

Listing 2.1: An example of SPARQL query.

Although SPARQL is a RDF standard query language recommended by W3C, it still has some shortcomings. The major limitation is the lack of spatio-temporal support. To query spatio-temporal data, SPARQL requires the user to be familiar with the underlying ontology that was used to model the data. Basic spatio-temporal queries can be expressed by using SPARQL Date-time functions and query graph patterns. However, matching the spatio-temporal representation graph using RDF triples makes building the query more complicated. This limitation is inherited from SPARQL which was not originally designed for expressing spatio-temporal queries.

2.1.4 RDF Stores

RDF stores are the backbone of the Semantic Web, allowing storage and retrieval of semi-structured information [37]. The engineering of RDF stores is still an open area which attracts a lot of research and solutions have been proposed in respect of efficiently processing RDF data. Current RDF stores can be consequently categorized into three types, described below, depending on their querying processing and data storing methods [100].

- **Relational-based RDF stores**, making use of Relational Database systems to store RDF data permanently. A proper query translator will then be provided to translate the SPARQL query into equivalent relational algebraic expressions to execute [35]. The initial target of this type of triple store is to allow large and powerful solutions to be constructed with little programming effort. As commonly used in relational database systems, the table-based indexing mechanism is used to index and store RDF data.

- **Native RDF stores**, processing SPARQL queries using subgraph matching algorithms. In this case, the underlying store provides a way to store RDF closer to the data model and refrains from using mapping to a DBMS. It uses the triple nature of RDF as an asset.

- **NoSQL RDF stores**, providing a scalable solution for storage and retrieval of RDF data based on NoSQL technologies. There are various models of NoSQL databases such as

column-based (HBase[2], Cassandra[3]), document-oriented (MondoDB[4], MarkLogic[5]), graph (Neo4J[6], AllegroGraph[7]).

In the scope of this thesis, we only focus on the native RDF and NoSQL solutions. This is due to the fact that our proposed query engine is backed by a hybrid database management system which is a combination of native RDF framework and NoSQL technology.

2.2 THE LINKED DATA PRINCIPLES

The LD principles were introduced by Berners-Lee in his "Linked Data" note [21]. The idea behind these principles is to propose standards for the representation and the access to data on the Web. Moreover, the principles propagate to set hyperlinks between data from different sources. These hyperlinks connect all LD into a single global data graph, similar to the hyperlinks on the classic Web connect all HTML documents into single global information space. The Linked Open Data cloud diagram in Figure 2.1 gives an overview of the linked data sets that are available on the Web. The principles are described as follows:

1. *Use URIs as names for things*: The first principle recommends using URIs to identify individuals (e.g. thing, places, people), classes, and properties (e.g. relationships between objects, the color of an object). Note that, an URI is a single global identification system used for giving unique names to anything – from digital content available on the Web to real-world objects and abstract concepts. With the help of URIs, we can distinguish between different things or know that one thing from one dataset is the same as another in a different dataset.

2. *Use HTTP URIs, so that people can look up those names*: The second LD principle advocates combining the use of HTTP – the universal access mechanism of the Web – and URIs to enable the ID of every entity accessible via HTTP URI protocol.

3. *When someone looks up an URI, provide useful information, using the standards (RDF, SPARQL)*: The third LD principle advocates the use of a linked data standard model (RDF) for publishing data so different applications can process or query the data via SPARQL language.

4. *Include links to other URIs, so that they can discover more things*: The fourth principle advocates linking any type of things using their URIs. Similar to the hypertext web, linking to other URIs makes data interconnected and enables the user to find different things. By interlinking new information with existing resources, we maximize the reuse and interlinking among existing data and create a richly interconnected network of machine-processable meaning.

2 https://hbase.apache.org/
3 http://cassandra.apache.org/
4 https://www.mongodb.com/
5 https://www.marklogic.com/
6 https://neo4j.com/
7 https://franz.com/agraph/allegrograph/

2.3 PUBLISHING LINKED DATA

Publishing LD requires adopting the LD principles previously discussed in Section 2.2. In this section, we will present the important guidelines for publishing LD. This process is the fundamental instructions for our Linked Meteorological Dataset publishing process described later in Chapter 4.

2.3.1 Linked Data Design Considerations

Heath and Bizer [82] propose the primary design considerations that must be taken into account when preparing data to be published as LD on the Web, which are: using URIs as names for things, describing things with RDF, and making links with external data sources.

2.3.1.1 *Using URIs as Names for Things*

As presented in Section 2.2, the first LD principle is to use URIs as names for things. Things can be real-world entities such as a person, a place, a building, or more abstract concepts such as a scientific concept. Names for these things are considered as unique identifications so that they can be used to refer to a thing. In addition to the first principle, the second LD principle emphasizes using *http://* URI scheme to allow *names* to be looked up by any client that speaks the HTTP protocol. Using HTTP URIs as names imply that a data publisher should choose a part of a *http://* namespace that she controls. It can be either by owning the domain name, running a Web server for the domain name, and minting URIs in this namespace to identify the thing in the dataset. To promote linking to a data set, some guidelines have been proposed for having stable and persistent URIs: (1) To enable URI dereferencing, namespaces on which the data publisher does not have control should not be used; (2) URIs should not reflect implementation details that may need to change over time; (3) Keys that are used to creating URIs should be meaningful in the domain of a data set.

2.3.1.2 *Describing Things with RDF*

The third LD principle recommends providing useful information in response when a user client looks up an URI. This information is described by RDF which provides a generic, abstract data model for describing resources using RDF triples. However, RDF lacks providing domain-specific terms for describing real-world entities as well as their relationships. In this case, the taxonomies, vocabularies, and ontologies are used. SKOS (Simple Knowledge Organization System) [123], RDFS [27], and OWL [122] are the tools to express ontologies and taxonomies. SKOS can be used to express the conceptual hierarchies (or taxonomies). RDFS and OWL provide vocabularies to describe conceptual models, e.g, classes and properties. Existing vocabularies are also recommended to be reused.

2.3.1.3 *Making Links with External Datasets*

When a dataset is ready to be published as LD, it is important to ensure that all the related resources in the dataset are linked to each other and also are linked to other data sources. By doing so, the dataset becomes more discoverable for crawlers and LD application. In addition, those external datasets may include links to some resources in other external datasets, which leads to discovering even more data. An efficient strategy for making links with external datasets is to create the necessary RDF links and ask the owners of the external datasets to include these triples in their data. The opposite way is also equally important that a new dataset link to resources in the external dataset. DBpedia[8] is an example of this case, which allows third parties to include triples with links to their datasets.

2.3.2 Serving Linked Data

The minimal requirement for publishing LD is making URIs dereferenceable. In addition, the data publishers can also provide RDF dataset dumps or SPARQL endpoints for directly querying data. This section presents different ways to serve LD.

2.3.2.1 *Serving LD from a Relational Database*

There are many data publishers use a relational database to serve their linked dataset. An example of this is Big Lynx job vacancies database, which drives the publication of past and present job vacancy information on the Big Lynx Web site [9]. One well-known used tool designed for this purpose is D2R Server[10]. The use of D2R Serer is to build a mapping between the database schema and the target RDF terms, which is provided by the data publisher. Based on this mapping, the D2R serves will transform the relational data into RDF and response to the client. Figure 2.4 illustrates the architecture of D2R Server.

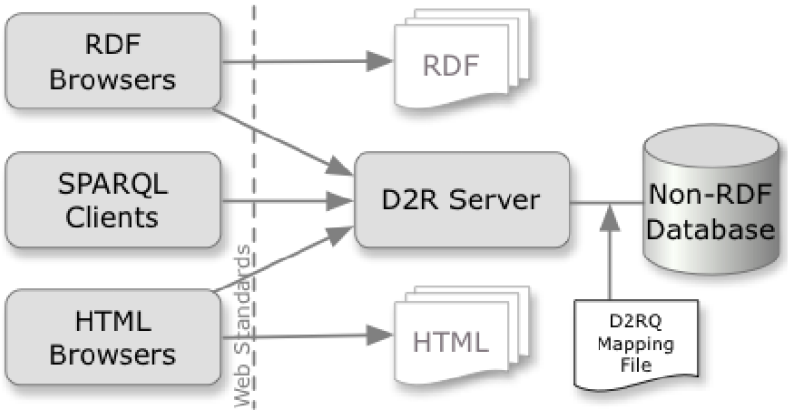


Figure 2.4: D2R Server Architecture

8 http://wiki.dbpedia.org/Downloads351#h120-1
9 http://biglynx.co.uk/
10 http://sites.wiwiss.fu-berlin.de/suhl/bizer/d2r-server/index.html

2.3.2.2 *Serving Linked Data from SPARQL Endpoints*

A SPARQL endpoint is a SPARQL query service on an HTTP network that is capable of receiving and processing SPARQL Protocol [53]. The SPARQL Protocol is an HTTP-based protocol for performing SPARQL query operations against RDF data via SPARQL Endpoints. Subject to the kind of operation being performed, HTTP payloads are dispatched using *GET*, *POST*, or *PATCH* methods. It also allows the user to specify the HTTP responses format for a SPARQL query and an update operation. However, an update operation is usually for private SPARQL endpoint. Public SPARQL endpoints limit the user to only query the data and do not support update operation. Nowadays, a large fragment of LD is served using SPARQL endpoints.

2.3.2.3 *Serving Linked Data by Wrapping Existing Application or Web APIs*

Recently, a lot of large companies such as Amazon, Facebook, Twitter allow the user to query their data via provided Web APIs. These APIs provide a set of varied query and retrieval interfaces and return results using a number of different formats such as plain text, XML, JSON, etc. In general, the content of the data responded by these APIs can be made available as LD by developing a LD wrapper around the APIs. The main functionalities of this wrapper are as follows:

1. To assign HTTP URIs to the resources about which the API provides data.
2. To rewrite the LD client's request into a request against the underlying API.
3. To transform the response of the API request to RDF and sent back to the client.

2.4 SEMANTIC SENSOR WEB AND LINKED SENSOR DATA

The sensor network has recently received more and more attention. This is evidenced by the increased deployments in all kinds of environments for different purposes for example, in environmental monitoring, urban traffic planning, flood prediction, health care, satellite sounding and other domains. Massive sensor data are generating continuously. However, these data are heterogeneous (different format, protocols, etc) and still lack semantic information, making more challenging for integrating and sharing sensor data. To address this problem, the Semantic Sensor Web (SSW) is introduced [174]. The main idea of the SSW is to use SW technologies for annotating sensor data. By using SW technologies, SSW is able to establish the common conceptual model, enhances the sensor data semantics, and enables the interaction and access of sensor data on the web.

The benefits of the SSW will not be fully realized if the sensor data are not published to LD. As emphasized in [17], by publishing sensor data to LD and by relating them to other data sources on the Web, users will be able to integrate the physical world data and the logical world data. This integration promises a wide range of innovative and valuable applications in smart cities, business intelligence, smart environments, etc. Moreover, because the Linked Sensor Data adopts all the LD principles and characteristics, it can be also queried, accessed and reasoned

```
<script type="application/LD+json">
{
  "@context" : {
    "@vocab" : "http://schema.org/"
  },
  "@type" : "Place",
  "@id" : "http://graphofthings.org/resource/Place/3329155",
  "url" : "http://graphofthings.org/resource/Place/3329155",
  "geo" : {
    "@type" : "GeoCoordinates",
     "longitude" : "-4.268",
     "latitude" : "50.184"
  }
}
```

Listing 2.2: Sensor location using longitude and latitude coordinates.

```
<script type="application/LD+json">
{
  "@context" : {
    "@vocab" : "http://schema.org/"
  },
  "@type" : "Place",
  "@id" : "http://graphofthings.org/resource/Place/5468",
  "url" : "http://graphofthings.org/resource/Place/5468",
  "name": "Anne Franks House",
  "description": "Museum house where Anne Frank and her family hid from the Nazis
      in a secret annex, during WWII.",
  "address" : {
    "type" : "PostalAddress",
    "streetAddress" : "Prinsengracht 267",
    "addressLocality" : "Amsterdam",
    "postalCode" : "1016GV"
  }
 }
```

Listing 2.3: Sensor location using complex description.

based on the same principles and technologies of LD. In comparison with LD, Linked Sensor Data has some additional attributes described as follows:

- **Spatial attributes:** The spatial attributes indicate the location-specific information for the sensor. It can be either a very specific geo-location such as longitude and latitude coordinate or high-level information likes postcodes, geohash string, country, etc. Listing 2.2 and Listing 2.3 illustrate different granularity level of sensor location in JSON-LD format.

- **Temporal attributes:** Temporal attributes in sensor observation data are used to indicate the observation result and measurement timestamp. In Linked Sensor Data context, using common ontologies for temporal specifications is an essential requirement to enable LD consumers to query and access temporal features of data using standard models and interfaces. Details of temporal representation of sensor data will be discussed more in the later section and also in Chapter 3.

- **Semantic attributes:** Semantic attributes, or sensor metadata, provide the bridge to connect sensor data and its domain knowledge. The metadata can be sensor type, tags, observed property, unit of measurement, features of interest and other more specific attributes such as operational and deployment attributes describe sensors with domain knowledge. As the sensor domain knowledge and its attributes can be various, relying on only general

sensor ontology will not be sufficient. Therefore, to describe more specific information, application/domain ontologies are suggested to describe detailed attributes more precisely.

2.5 REPRESENTATION OF SPACE

In recent years, a set of well-known standards have been developed by the Open Geospatial Consortium[11] OGC, such as GeoAPI, GeoSPARQL, GeoRSS, etc, which focuses on representing a geographic feature, data, and services, or geospatial data sharing. A geographic feature is a reflection of a real-world phenomenon and can have various attributes that describe its semantic and spatial aspects. A simple example of a geographic feature is a sensor station. In this example, the semantic information of a station can be the station id, the observed property, etc., while the spatial characteristic is its location on Earth. The spatial aspect of the sensor station can be represented by using geometries such as points, lines, and polygons. Each geometry is usually associated with a coordinate reference system which describes the coordinate space in which the geometry is defined.

In the following subsections, we will briefly give an overview of the coordinate reference system, followed by the definition of spatial relationships and the most widely-used space representation. The W3C Geo vocabulary and geohash system are also presented. Note that, having knowledge about the representation of space and its spatial-related concepts is crucial for implementing our spatio-temporal query processing engine.

2.5.1 Coordinate System

A Coordinate System (CS) defines the reflected relationship between the coordinates of a geometric object to the real locations on the Earth surface. Different kinds of CS exist, such as the geographic and projected coordinate systems. The geographic CS utilizes latitude, longitude, and optionally altitude, to capture geographic locations on Earth [116]. Specifically, this CS uses a series of horizontal (longitude lines or parallels) and vertical (latitude lines or meridians) reference lines to map the earth's three-dimensional ellipsoid. One of the most well-known geographic CS is the World Geodetic System (WGS). The latest revision of WGS is WGS84[12] used by the Global Positioning System (GPS).

Despite the commonality of WGS, there are still some applications that use the projected CS. In these cases, the projected CS is used to map the 3-dimensional ellipsoid approximation of the Earth into a 2-dimensional surface. For some individual countries or states, they developed their own projected CS that is more precise for their geographic area.

11 http://www.opengeospatial.org/
12 http://en.wikipedia.org/wiki/WGS84/

2.5.2 Spatial Relations

A spatial relation associates two spatial objects according to their relative location and extent in space. Spatial relationship can be *topological*, *directional* and *distance* [159].

Topological relations are used to represent the relative position of two spatial objects in the plane. The Region Connection Calculus (RCC) Formalism, introduced in [158], is the most widespread formalism for representing such relationships. There are 8 possible mutually exclusive relations between two regions, called RCC8 calculus[13]. The topological relations are shown in Figure 2.5. In this figure, the DC stands for DisConnected, EC for Externally Connected, TPP for Tangential Proper Part, NTPP for Non-Tangential Proper Part, and TPPi and NTPPi are the inverse relations from TPP and NTPP.

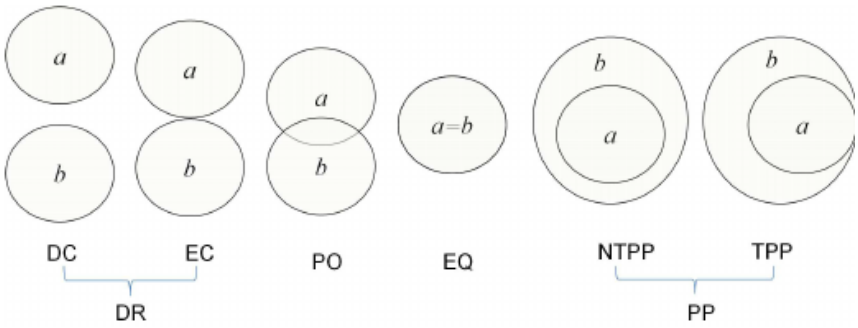


Figure 2.5: RCC8 spatial relations

Another similar formalization that provides a sound and complete set of topological relations between two spatial regions is the Dimensionally Extended nine-Intersection Model (DE-9IM) [39]. This model expresses important space relations which are invariant to rotation, translation and scaling transformations. For any two spatial objects a and b, that can be points, lines and/or polygonal areas, there are 9 relations derived from DE-9IM as shown in Table 2.2.

Table 2.2: DE-9IM topological relations

Equals	$a = b$ Topologically equal. Also $(a \cap b = a) \wedge (a \cap b = b)$
Disjoint	$a \cap b = \varnothing$ a and b are disjoint, have no point in common. They form a set of disconnected geometries.
Intersects	$a \cap b \neq \varnothing$
Touches	$(a \cap b = \varnothing) \wedge (a^0 \cap b^0 = \varnothing)$ a touches b, they have at least one boundary point in common, but no interior points.
Contains	$a \cap b = b$
Covers	$a^0 \cap b = b$ b lies in the interior of a (extends Contains).
CoveredBy	Covers(b,a)
Within	$a \cap b = a$
Crosses	a crosses b, they have some but not all interior points in common

Directional relations are defined based on cone-shaped areas [126]. As illustrated in Figure 2.6, there are eight directional relations, namely North (N), North East (NE), East (E), South East (SE), South (S), South West (SW), West (W) and North West (NW). For example, the assertion "object A is south of object B" indicates the directional relation of A to B.

13 https://en.wikipedia.org/wiki/Region_connection_calculus

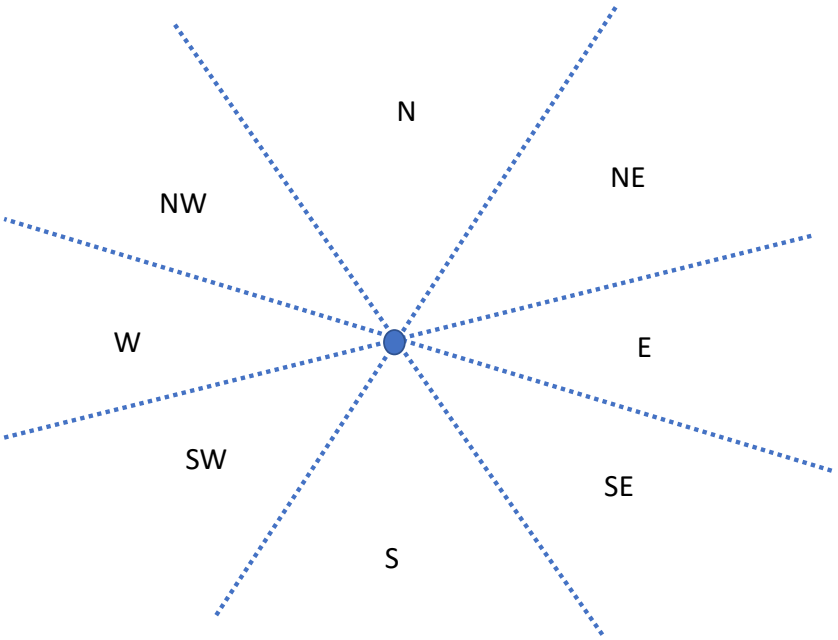


Figure 2.6: Cone-shaped directional spatial relations

Finally, the ***distance relations*** specify how far is the object away from the reference object, ie, "object B is 5 Km away from city A". Some qualitative distance relations are defined, such as "far", "near", "at", etc.

2.5.3 Well-Known Text

Well-Known Text (WKT) is a widely-used OGC text markup language for representing vector geometry objects on a map or spatial reference systems. It is also used for the data transformation between spatial reference systems. An example of WKT is POINT(44 55) that describes the longitude (44) and latitude (55) coordinates of a single point. More syntax details can be found in [84]. The interpretation of a geometry coordinate depends on the CRS that is associated with the geometry. Note that, according to the WKT standard, the CRS is never embedded in the object's representation, but is given separately by using appropriate notation.

2.5.4 W3C Basic Geo Vocabulary

The W3C Basic Geo Vocabulary is the RDF vocabulary for representing the latitude and longitude of spatially-located things. It is widely used within RDF documents, and also can be used as a namespace within non-RDF XML documents, such as RSS 2.0 and Atom. The W3C Geo Vocabulary uses WGS84 as a reference system. The simplicity of this representation lies in the fact that it does not require expensive pre-coordination or changes to a centrally maintained schema. The basic example of W3C Geo vocabulary is given in Listing 2.4.

```
_:1 RDF:type wgs84geo:Point .
_:1 wgs84geo:lat "10"^^xsd:double.
_:1 wgs84geo:long "20"^^xsd:double.
```

Listing 2.4: An example of W3C Geo vocabulary

2.5.5 Geohash

Geohash is a hierarchical spatial data structure which subdivides space into buckets of grid shape. It encodes the longitude and latitude of a geographic point coordinates into string identifiers. One of the benefits of geohash is that it supports multiple-precision just by removing or adding characters to the final hash value. The lower the number of characters used, the lower will be the precision. The advantage of using geohash is that nearby places usually share similar prefixes. The longer a shared prefix is, the closer the two places are [58, 180].

A geohash is constructed by a series of bits that repeatedly bisects a search space of latitudes and longitudes. The longitude dimension has a range [-180.0, 180.0], and the latitude dimension has a range [-90.0, 90.0]. When generating a geohash, a precision is specified. The length of 12 characters is the highest precision that 8-byte Long can hold. When removing the final characters from a geohash, its precision decreases and hence it represents a larger area of the map. The full precision is 12 characters which represent a point. Any geohash with less than 10 characters represents an area on the map, i.e. bounding box around an area. Figure 2.7 illustrates how the first character of a geohash string splits the projected earth into an 8 x 4 grid of horizontally aligned rectangles.

Figure 2.7: Space decomposition of the geohash algorithm on the first level

As previously mentioned, in a geohash system, nearby locations generally share similar geohash prefixes. However, in some cases, the nearest point can have wildly different geohash value, if the location is close to a grid-square boundary. Figure 2.8 illustrates an example of two closely

positioned points being located within different geohashes. The first point is located at the *"gc"* geohash cell and the second point positioned closely at the *"u1"* geohash cell.

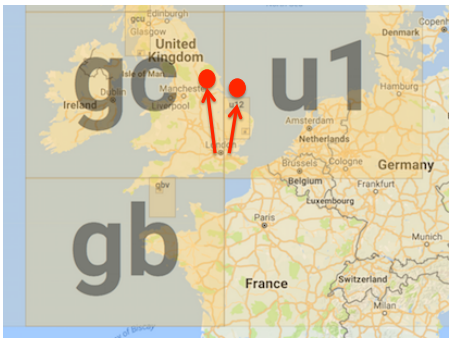


Figure 2.8: An example of two closely positioned points do not share the same Geohash prefix

Our query engine takes advantage of the geohash prefix for querying and partitioning data. More specifically, we assign a range of geohash prefix to each data partition. For this, all observation data of nearby sensors will be stored in the same partitions. This has extremely benefited the query performance and data loading throughput so that the query engine can quickly locate the partitions need to be processed for a query. Details of using geohash for data partitioning will be discussed in Chapter 6.

2.6 REPRESENTATION OF TIME

Along with space, time is also a fundamental aspect of the conceptualization of the physical world. Time can be presented as discrete or continuous, linear or cyclical, absolute or relative, qualitative or quantitative. In addition, time instances or intervals are also a different representation of time.

In the relational database field, the introduction of time in data models and query languages has attracted a lot of attention. The authors in [181, 46] introduced three distinct types of time:

- **User-defined time** refers to the time known only by the user, and is not interpreted by the database (e.g., February 1st, 2018 when the Weather Station A is setup)
- **Valid time** which is the time at which the event occurred in the real world, independent of the recording of that event in the database (e.g., the time 2018/07/15-10:49:33 when the observation data is observed). Within the scope of this thesis, we only consider the valid time when processing the temporal data.
- **Transaction time** which is the time when data is inserted in the database (e.g.,2018/07/15-10:50:18, the system time that the observation data is inserted in the database)

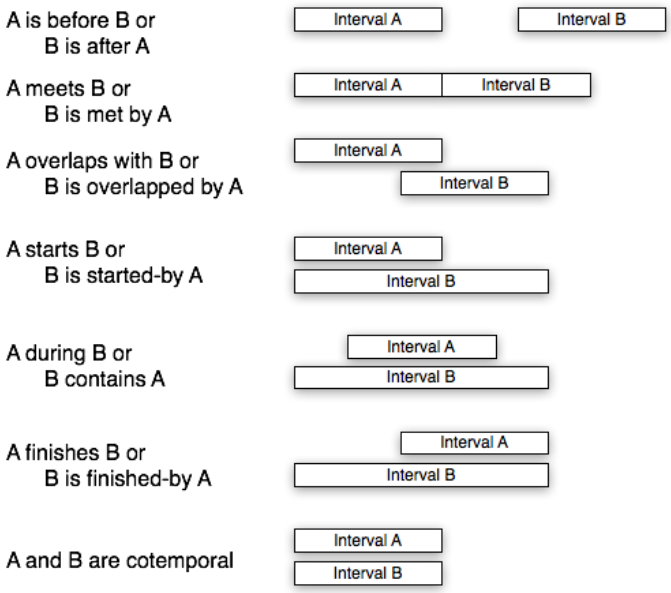


Figure 2.9: Allen's interval calculus

2.6.1 Temporal Relations

Temporal relations can be grouped into three major classes, namely ordering relations, duration relations, and complex relations. This thesis only focuses on ordering relations.

Ordering relations are binary relations that specify the order of occurrence of time points or intervals. Based on these ordering relations, tine intervals can be defined as ordered pairs of points s, e with $s < e$, often referred to start and end of an interval respectively. Given two such ordered pairs of points (i.e., intervals) the following relations can be defined between the two intervals t_1, t_2 in terms of their endpoints s_1, e_1 and s_2, e_2 respectively:

- $t_1 \text{ before } t_2 \equiv e_1 < e_2$
- $t_1 \text{ equals } t_2 \equiv s_1 = s_2 \wedge e_1 = e_2$
- $t_1 \text{ overlaps } t_2 \equiv s_1 < s_2 \wedge e_1 < e_2 \wedge e_1 > s_2$
- $t_1 \text{ meets } t_2 \equiv e_1 = s_2$
- $t_1 \text{ during } t_2 \equiv s_1 > s_2 \wedge e_1 < e_2$
- $t_1 \text{ starts } t_2 \equiv s_1 = s_2 \wedge e_1 < e_2$
- $t_1 \text{ finishes } t_2 \equiv s_1 > s_2 \wedge e_1 = e_2$

Similarly, the definitions of *after*, *overlappedBy*, *metBy*, *contains*, *startedBy* and *finishedBy* are defined accordingly by investing the definitions of *before*, *overlaps*, *meets*, *during*, *starts* and *finishes*. A temporal relation can be one of the 13 pairwise disjoint Allen's relations [8] in Figure 2.9.

2.7 SUMMARY

This chapter presents the relevant concepts, definitions, and techniques in the SW, LD, Linked Sensor Data, spatial and temporal representation. We first presented SW and LD, highlighted the foundation for data modeling, querying language and storage system. It was followed by the LD principles and the considerations for publishing LD. Besides, the Linked Sensor Data and its attributes are also introduced. Finally, we present the representation for space and time which are necessary for modeling Linked Sensor Data. All these techniques, concepts of these research areas are the leverages for developing our proposed query processing engine.

3 RELATED WORK

The research in this thesis lies at the crossing point of different areas of Computer Science, namely Sensor Networks, Semantic Web, Linked Sensor Data and Database Management Systems. Specifically, it is built on the research which involve data modeling, data indexing, and RDF query processing. Furthermore, our work is also heavily related to spatio-temporal data modeling as well as spatio-temporal database techniques for managing sensor data. In this chapter, we report the related work from these different areas and point out the limitations. We divide related work into five parts: (1) data modeling for Linked Sensor Data, (2) spatio-temporal query language; (3) publishing Linked Sensor Datasets on the LOD cloud; (4) RDF stores with spatio-temporal supports and (5) RDF store assessment.

3.1 DATA MODELING FOR LINKED SENSOR DATA

In this section, we first present some existing ontologies for modeling Linked Sensor Data. After that, we cover the spatio-temporal representation approaches that can be used for describing spatial and temporal aspects of Linked Sensor Data.

3.1.1 Sensor Ontologies

During the last decade, an extensive amount of ontologies has been proposed to not only address the challenge of modeling sensor network and its data, but also to tackle the heterogeneity problems associated with the hardware, software, and the data management aspect of sensors. These ontologies provide a means to semantically describe the sensor networks, the sensing devices, the sensor data, and enable sensor data fusion. In this thesis, the term *"sensor data"* is used to refer to the raw sensor data generated by sensing the phenomenon, and the *"sensor metadata"* is to describe the sensor capabilities.

The state-of-the-art approach in this area is the work from the OGC Sensor Web Enablement (SWE) working group [26]. They have specified a number of standards that define formats for sensor data and metadata as well as sensor service interfaces. These standards allow the integration of sensor and sensor networks into the Web, in what is called Sensor Web. In particular, they provide a set of standard models and XML schema for metadata description of sensors and sensor systems, namely SensorML [25], and Observations and Measurements (O&M) models for data observed or measured by sensors [41, 42]. Lack of semantic compatibility, however, is the primary barrier to realizing a progressive Sensor Web.

In [174], Amit et al. propose Semantic Sensor Web (SSW) that leverages current standardization efforts of the OGC SWE, in conjunction with the Semantic Web Activity of the W3C[1], to provide enhanced descriptions and meaning to sensor data. In comparison with Sensor Web, the SSW addresses the lack of semantic compatibility by adding semantic annotations to existing SWE's standard sensor languages. In fact, these improvements aim to provide more meaningful descriptions to sensor data than SWE alone. Moreover, the SSW acts as a linking mechanism to bridge the gap between the primarily syntactic XML-based metadata standards of the SWE and the RDF/OWL-based metadata standards of the Semantic Web.

The work in [163] describes a practical approach for building a sensor ontology, namely OntoSensor, that uses the SensorML specification and extends the Suggested Upper Merged Ontology (SUMO) [136]. The objective of OntoSensor is to build a prototype sensor knowledge repository with advanced semantic inference capabilities to enable the fusion processes of heterogeneous data. For that reason, in addition to reusing all SensorML's concepts[25], OntoSensor provides additional concepts to describe the observation data, i.e, the geolocation of the observations, the accuracy of the observed data or the process to obtain the data.

Similar to OntoSensor, the W3C Semantic Sensor Network Incubator group (SSN-XG) has defined the SSN ontology [40] in order to overcome the missing of semantic compatibility in OGC SWE standards, as well as the fragmentation of sensor ontologies into specific domains of application. The SSN ontology can be considered as a sort of standard for describing sensors and their resources in respect to the capabilities and properties of the sensors, measurement processes, observations and deployment processes. It is worth mentioning that, although the SSN ontology provides most of the necessary details about different aspects of sensors and measurements, it does not describe domain concepts, time, location, etc. Instead, it will be easily associated with other sources of knowledge concerning, i.e, units of measurement, domain ontologies (agriculture, commercial products, environment, etc.). This helps to pave the way for the construction of any domain-specific sensors ontology. Because of its flexibility and adaptivity, the ontology has become more general and has been used in many research projects and applied to several different domains in the last years. Some of the most recently published works which utilize the SSN ontology are the OpenIoT Project [182], the FIESTA-IoT[2][6], VITAL-IoT[3] and GeoSMA [91].

The broad success of the initial SSN led to a follow-up standardization process by the first joint working group of the OGC and the W3C. This collaboration aims to revise the SSN ontology based on the lessons learned over the past years and more specifically, to address changes in scope and audience, shortcomings of the initial SSN, as well as technical developments and trends in relevant communities. The resulting ontology, namely SOSA ontology [94], provides a more flexible coherent framework for representing the entities, relations, and activities involved in sensing, sampling, and actuation. The ontology is intended to be used as a lightweight, easy to use, and highly expandable vocabulary that appeals to a broad audience beyond the Semantic Web community but can be combined with other ontologies. In this thesis, to model the sensor data and metadata, several parts of SOSA and SSN ontologies are used along with spatio-temporal ontologies. Details of our data modeling for Linked Sensor Data will be presented in Chapter 4.

1 www.w3.org/2001/sw/
2 http://fiesta-iot.eu/
3 http://www.vital-iot.eu/

3.1.2 Temporal Representation for Linked Sensor Data

Introducing time into RDF has attracted a lot of attention from the Semantic Web community for years. Many attempts that semantically define temporal aspect have been proposed in the literature. In this section, we will analyze the advantages and disadvantages of common approaches for representing time and change that can be applied in Linked Sensor Data context.

OWL-TIME The most commonly used time ontology is the OWL-Time ontology proposed in [87] that focuses on describing date-time information specified in Gregorian calendar format. Initially, the OWL-Time was used to present the temporal information of Web pages and services. After several updates to the initial version, the ontology became widely-used and has been considered as a main reference time ontology. Based on its diversity concepts, the ontology is able to represent time, duration, clock, calendar, and temporal aggregates in many domains, i.e, information retrieval and question answering.

In its latest version, the OWL-Time ontology provides more concepts to describe time. These concepts can be categorized into five main core concepts, namely *TemporalEntity*, *Instant*, *Interval*, *ProperInterval*, and *DateTimeInterval*. The *TemporalEntity* is the root node of the ontology and can be refined in two sub classes: *Instant* and *Interval*. There are two properties, namely *hasBeginning* and *hasEnd*, that link to *TemporalEntity* to define the start and finish of its temporal duration. In addition, the ontology defines *CalendarClockDescription*, *DurationDescription*, and *TemporalUnit*. OWL-Time provides two alternative ways to represent time points, namely *DateTimeDescription* and *xsd:dateTime*. The advantage of the former is that it allows user to express more information (e.g., day of week, day of year, timezone, etc.), while the latter is based on a standard which has a wider acceptance and usage. Figure 3.1 illustrates an example of using OWL-Time to describe the sensing time of sensor observation. In here, we define an instant to represent the sensing time of an observation, called *ex:observedTime*. The sensing time (10:30am AEST on 25 July 2019) can be expressed using both *:inXSDDateTimeStamp* and *:inDateTime* in OWL. An example of using OWL-Time *DateTimeDescription* class is described in Listing 3.1.

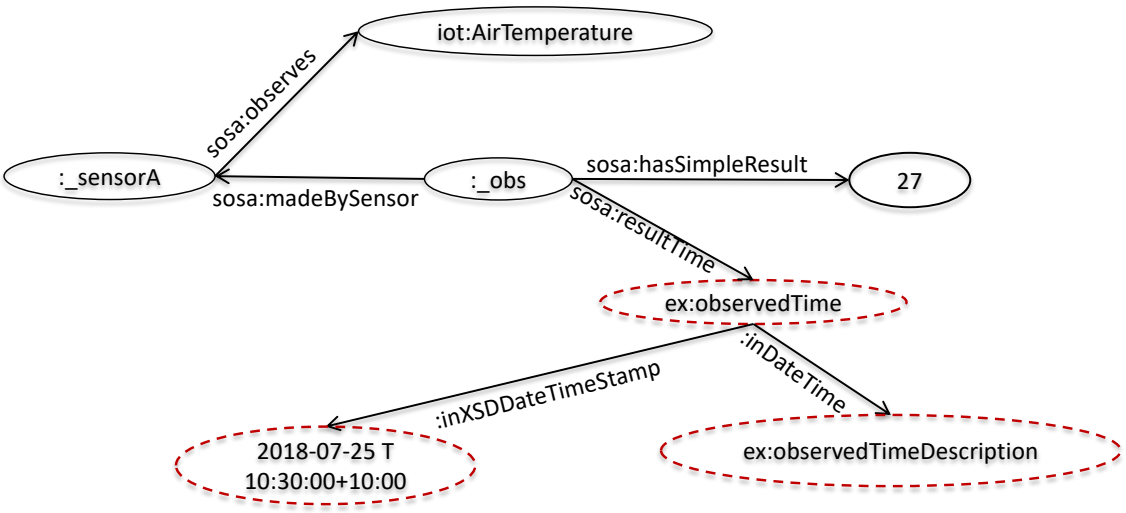


Figure 3.1: An example of using OWL-Time ontology to describe temporal information

```
ex:observedTime
  a                    :Instant ;
  :inDateTime          ex:observedTimeDescription ;
  :inXSDDateTimeStamp  2019-07-25T10:30:00+10:00 .

ex:observedTimeDescription
  a             :DateTimeDescription ;
  :unitType     :unitMinute ;
  :minute       30 ;
  :hour         10 ;
  :day          "---25"^^xsd:gDay ;
  :dayOfWeek    :Wednesday ;
  :dayOfYear    206 ;
  :week         30 ;
  :month        "--07"^^xsd:gMonth ;
  :monthOfYear  greg:July ;
  :timeZone     <https://www.timeanddate.com
                              /time/zones/aest> ;
  :year         "2019"^^xsd:gYear .
```

Listing 3.1: An example of OWL-Time DateTimeDescription

TEMPORAL RDF Temporal RDF [73] and its extension [152] provide a notion of incorporating temporal reasoning into RDF, yielding temporal RDF graphs. In this work, the authors define a new concept, namely *temporal label*, which is a temporal element t labeling a triple (a,b,c). The temporal label can be either a time interval or instant. The time interval represents the time period when the triple was valid and can be described by two defined properties (*initial* and *final*). For simplicity, the proposed approach only focuses on the valid time of the triple, while stating that transaction time can be addressed in an analogous way. An example of using Temporal RDF graph to describe an observation value with different valid timestamps is in Figure 3.2. In this example, we describe two observation values, *27* and *29*. The value *27* was valid within a time period *[0,3]* while the value 29 has been valid from time *3* to present.

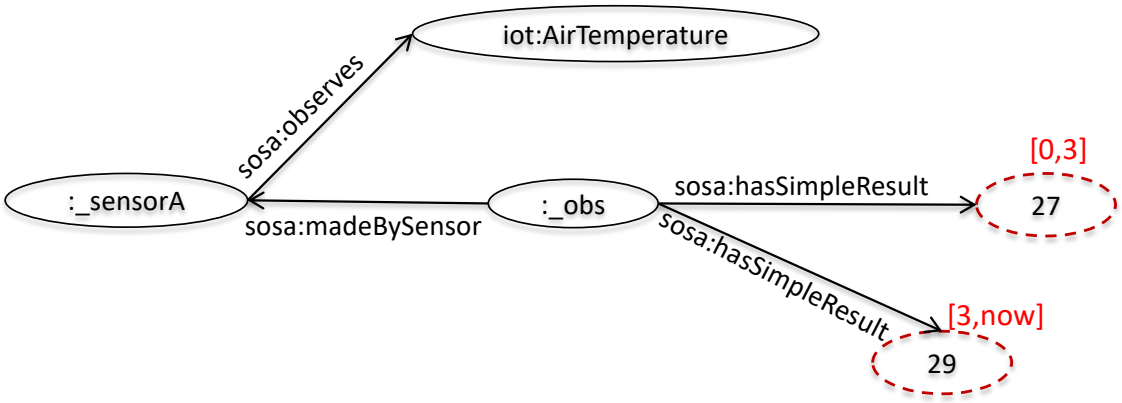


Figure 3.2: A Temporal RDF graph for observation data

A challenging problem of Temporal RDF is the requirement for extending the syntax and semantics of the standard RDF. For example, in order to handle an anonymous time of the triple, the authors introduce a *NOW* operator which stands for the current time. This new operator aims

to help the triples which do not know the exact valid time. For that, it acts as a place holder for the time at which the corresponding triple is evaluated. The authors also provide their own temporal query language, and it is claimed that the temporal labeling over triples does not introduce any complexity overhead. Nevertheless, from a practical perspective, because the query language is not similar with SPARQL, which is a well-known standard query language for RDF, it makes this approach difficult to use in practice. Moreover, details of the query processing engine correspond to the temporal query language are not mentioned.

REIFICATION In general, the RDF reification vocabularies aim to provide meta information about statements by turning them into subjects of other statements [4]. More precisely, it is used to represent *n-ary* relationships through binary relations. Based on this technique, several approaches [33, 173, 188] have been introduced to represent temporal information in RDF. In these approaches, if there is a relation R that holds between objects A and B at time t, this is expressed as $R(A,B,t)$. Figure 3.3 demonstrates the relation *Valid(Observation,Value,TimeInterval)* representing the fact that an observation has a sensing value that is only valid during a specific time interval. Using reification, the extra class *ReifiedRelation* is created having all the attributes of the relation as objects of properties.

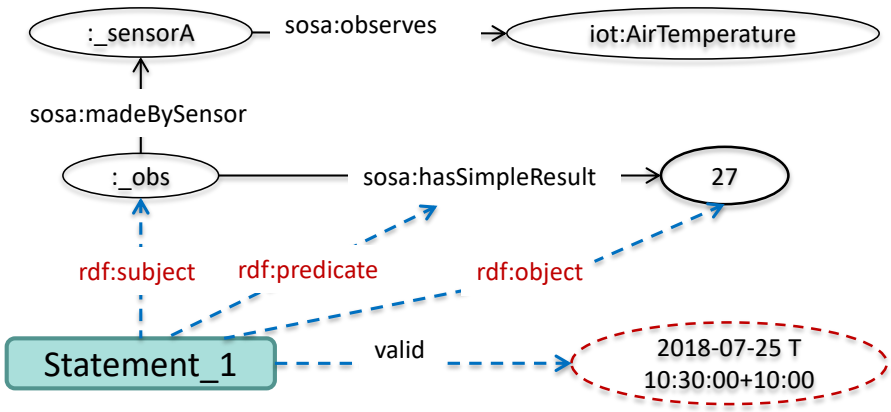


Figure 3.3: An example of temporal RDF reification

The advantage of using RDF reification approach is that the semantic of all the properties are retained. Nevertheless, using RDF reification is still not recommended due to the two following limitations [89]: (1) The number of statements is exploded which makes querying complicated. In addition, the reification model-theoretic implications are ambiguous; (2) It limits the OWL reasoning capabilities. This is because the relation R is represented as the object of a property, thus OWL reasoning over properties (e.g., inverse properties) cannot be applied (i.e., the properties of a relation are no longer associated directly with the relation itself).

NAMED GRAPHS In [189], the authors propose to use named graphs [32] to enable the temporal data representation by referring to a set of temporally related named graphs as a temporal graph. In this approach, each time interval is represented by exactly one named graph, where all triples belonging to this graph share the same validity period. The authors reuse the OWL-

4 http://www.w3.org/TR/rdf-mt/#Reif

Time ontology to describe the time interval of the named graph. This temporal information about the interval start and end time of all named graphs are then stored in the default graph. Because every interval is uniquely identified by the URI of its named graph, all the temporal-related information can be easily queried in the default graph. Figure 3.4 illustrates the temporal named graph of a sensor observation.

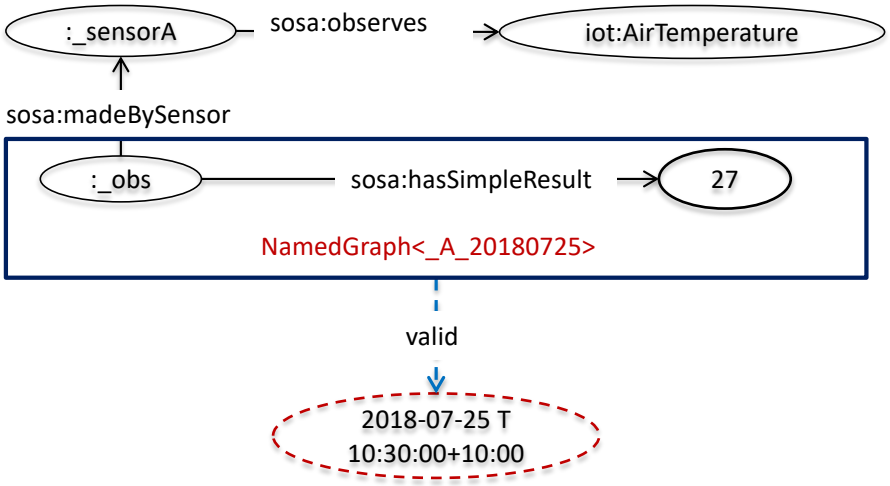


Figure 3.4: An example of RDF named graph for observation data

Similar to Temporal RDF, the authors of [189] also provide a temporal query language. The difference is that the language is extended from the standard SPARQL query language. This is actually an advantage of this approach in terms of practical usage. However, a major limitation needs to be addressed is the lack of a temporal reasoner. This is due to the fact that named graphs are not supported by OWL reasoners (named graphs are not part of the OWL specification). Moreover, this approach only performs best on data that has multiple data elements valid within the same interval. In the case of many distinct triple's validities, the performance is decreased due to the massive amount of named graph that needs to be defined.

4D-FLUENT The 4D-fluent approach [197] shows how temporal information and the evolution of temporal concepts can be effectively represented in OWL. According to the authors, the main reason for choosing OWL to represent the ontologies is its widely supporting and also to take advantages of the massive amount of existing OWL reasoners and editor tools. The main contribution of this approach is to provide a capability to describe the existence of an entity with multiple representations. Each representation is mapped to a defined time interval (time-slice) which is considered as the 4^{th} dimension. Time interval class is defined as a superclass while time instances and time intervals are represented of a time interval class which in turn is related to temporal concepts or temporal roles. An example of using this approach is shown in Figure 3.5.

In the literature, 4D-fluent is the efficient approach to handle dynamic properties in an ontology by providing a simple structure allowing to easily transform a static ontology into a dynamic one. However, the major limitation of this approach is the proliferation of objects. To describe a change of temporal entity, it is required two time-slices, each time-slice referring to one entity

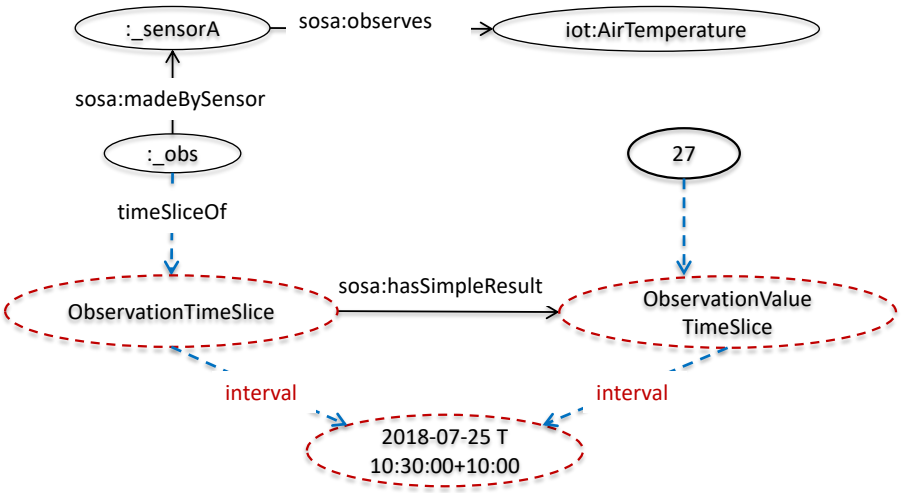


Figure 3.5: Representing temporal object with 4D-fluent

and one temporal interval. It means that there will be seven triples (including the triple in which the fluent is employed) need to be created in the dynamic domain for 1 triple in the static domain. As a result, it also causes an increase in the complexity of querying the temporal data and understanding the modeled knowledge.

OTHER APPROACHES In [86], the authors introduce the DAML-Time ontology that focuses on concepts to covers the basic topological temporal relations on instants and intervals, measures of duration, and the clock and calendar. Similar to the DAML-Time ontology, the KSL-Time ontology [203] provides a simple way to represent both the time points and time interval and considers them as primitive elements on a timeline. In general, the KSL-Time ontology defines a hierarchically structured class that describes the relationship between time interval and time points, axioms and time granularity. Different from the DAML-Time, the KSL-Time ontology distinguishes the closed and open intervals.

3.1.3 Spatial Representation for Linked Sensor Data

Similar to temporal representation, there has been significant community and research interest in representing and querying geospatial RDF data in the last few years. In [97], Katz et al. proposed an approach to translate the RCC8 calculus into OWL-DL, by adapting some of the know results on the translation of qualitative spatial formalism into Model Logics. However, the authors in [187] experienced that this approach lacks practicability and its implementation does not have a good performance. [36, 184] revisited the problem of defining a SW data model for spatial data by proposing an integrated spatio-temporal representation which includes qualitative relations. Unfortunately, these approaches lack of specialized spatio-temporal reasoning support.

In parallel with the aforementioned works, Amit Sheth's group has contributed a numerous interesting works to the representation of geospatial (and temporal) information into RDF [141, 142, 145, 143, 75, 31, 176]. Among these, Matthew Perry's approach [142] captures the geospatial

information by using an upper time and space ontology based on GeoSS. Particularly, in this ontology, the spatial information is presented through the class *geo:SpatialRegion* and its spatial subclasses (e.g., *geo:Polygon*). These classes are defined using the GeoRSS GML profile to model spatial geometries (e.g., polygons). The ontology also defines the property *stt:located_at* to link a geographic object (e.g., a town) to its spatial geometry (e.g., a polygon), which is an instance of the defined spatial classes. Depending on its type, various information of the geometry can be specified, i.e, the coordinates of the boundary as well as the coordinate reference system for the 2-dimensional polygonal area. A limitation of Perry's approach is in the SPARQL-St query language, which will be presented later in Section 3.2.

3.1.3.1 *GeoSPARQL*

A most widely-used approach in the Semantic Web community for representing geospatial data in RDF is GeoSPARQL [19], which provides a standard for geospatial RDF data insertion and query. It is worth mentioning that, part of our work is built upon the GeoSPARQL ontology and GeoSPARQL query language. The GeoSPARQL ontology is based on the OGC's Simple Features model [85], which is adapted for RDF. It defines a class *geo:SpatialObject*, which is a subclass of *owl:Thing*. As illustrated in Figure 3.6, the *geo:SpatialObject* class then has two primary subclasses, namely *geo:Feature* and *geo:Geometry*. These classes are meant to be connected to an ontology representing a domain of interest. Features can connect to their geometries via the *geo:hasGeometry* property.

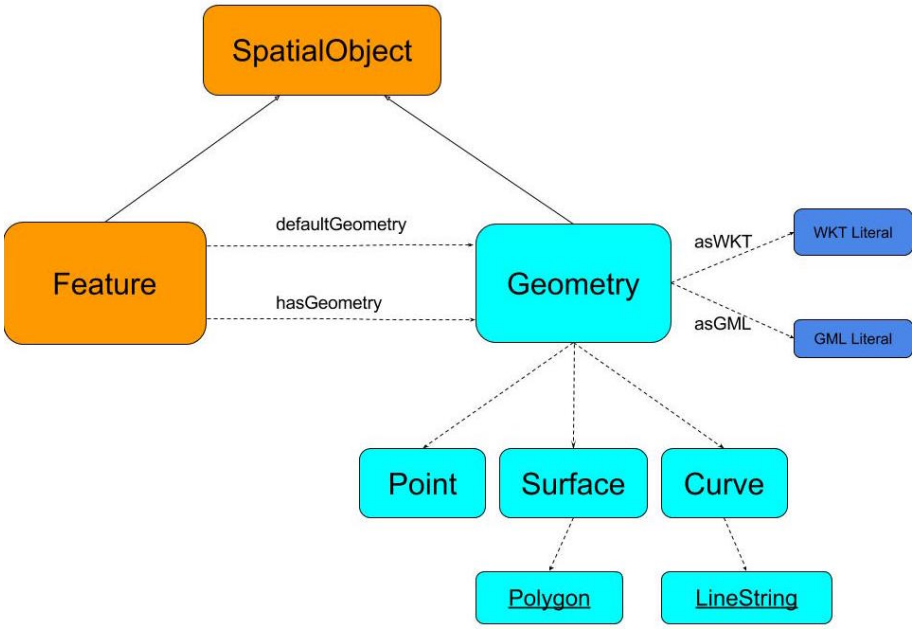


Figure 3.6: GeoSPARQL ontology [60]

GeoSPARQL provides various ways for representing geometry literals. It can be either WKT or GML using two additional RDF datatypes, the *geo-sf:wktLiteral*[5] and *geo-gml:gmlLiteral*[6] for these literals. Our query engine supports both of these representations. For geometry hierarchies description, GeoSPARQL provides different OWL classes for both WKT and GML, i.e, *geo-sf:Polygon*

5 http://www.opengis.net/def/dataType/OGC-SF/1.0/WKTLiteral
6 http://www.opengis.net/def /dataType/OGC-GML/3.0/GMLLiteral

(for WKT) and *geo-gml:Polygon* (for GML). Similarly, other geometry types such as point, curve, arc, and multi-curve are also supported. The *geo:asWKT*[7] and *geo:asGML*[8] properties link the geometry entities to the geometry literal representations.

An example of using GeoSPARQL ontology is modeling sensor location in our linked meteorological dataset presented in Chapter 4. In this dataset, we describe a sensor location as an instance of a *geo:Feature*. Note that, a sensor location can be either considered as a static location of a sensor station or a current location of a device that the sensor is attached. Either of them has to be measured and estimated in some way. A representation of the real-world location which has been measured becomes a *geo:Geometry*. For simplicity, in our dataset, the sensor location is measured as a single point via *geo:Point* class, a sub type of *geo:Geometry*. Details of a sensor location description in GeoSPARQL is given below:

```
prefix sosa: <http://www.w3.org/ns/sosa/>
prefix got: <http://graphofthings.org/ontology/>
prefix geo: <http://www.opengis.net/ont/OGC-GeoSPARQL/1.0/>
prefix geo-sf: <http://www.opengis.net/def/dataType/OGC-SF/1.0/>

:_station a got:WeatherStation;
          a geo:Feature;
          geo:hasGeometry :_pointA.
:_sensorA a sosa:Sensor;
        sosa:isHostedBy :_station
:_pointA a geo:Point;
    geo:asWKT "POINT(-77.022 38.2433)"^^geo-sf:wktLiteral.
```

Listing 3.2: Describing sensor location using GeoSPARQL ontology.

3.2 SPATIO-TEMPORAL QUERY LANGUAGE FOR LINKED DATA

Traditionally, SPARQL has been the query language for RDF data. It is a W3C recommendation that operates at the level of RDF graphs. However, SPARQL does not support spatio-temporal query. Using SPARQL for querying spatio-temporal information becomes relatively complex for building the query due to the complicated expressions. Additionally, it requires the user has to be familiar with the representation model and uses specific constructs triple patterns to the underlying spatio-temporal model. This section will review several spatio-temporal query languages for RDF that have been proposed recently to address this challenge.

In [107], the authors introduce the spatio-temporal model, namely stRDF, and its corresponding stSPARQL query language. The stSPARQL is an extension of SPARQL with additional temporal and spatial operators. Besides, it also defines two new variable types, namely spatial variables and temporal variables. The spatial variables can be used in basic graph patterns, refers to spatial literals denoting semi-linear point sets. The temporal variables can be used as the last term in a new defined temporal basic graph pattern, called quad pattern, refers to the valid time

7 http://www.opengis.net/ont/OGC-GeoSPARQL/1.0/asWK
8 http://www.opengis.net/ont/OGC-GeoSPARQL/1.0/asGM

of a triple. In the query translation phase, along with the temporal and spatial constants, these two type of variables can be taken as arguments of the spatial and temporal functions. Moreover, they can be used in the SELECT, FILTER, and HAVING clause of a SPARQL query. stSPARQL uses vectors of points for representing the different geometries and it does not support querying using qualitative operators as well as spatio-temporal reasoning.

Resuming the illustrative example query in Chapter 1, Listing 3.3 presents a translation of this query into stSPARQL. The stSPARQL extensions are marked in red colour. In this example, the stSPARQL query expressions involve model specific triples (i.e, *strdf:hasGeometry*), spatial function (i.e, *strdf:within*), temporal variable (i.e, *?T*) and its constraint (*t>="01/01/2019" and t<="31/03/2019"*). The example also implies that users have to be familiar with details of the underlying stRDF model for constructing a query.

```
SELECT ?attractions (avg(?value) as ?avg)
WHERE{
?entity skos:broader dbr:Category:Tourism_by_city .
?places skos:broader ?entity .
?attractions dcterms:subject ?places .
?attractions dbo:country ?country .
?country dbo:location dbr:Europe.
?attractions geo:lat ?lat; geo:long ?long.
?attractions strdf:hasGeometry ?attrLoc.
?sensor a sosa:Sensor;
        sosa:isHostedBy ?station;
        sosa:observes iot:AirTemperature.
?station a got:WeatherStation;
        geo:hasGeometry ?sLoc.
filter(strdf:within(?attrLoc,?sLoc)).
?obs sosa:madeBySensor  ?sensor;
         sosa:hasSimpleResult ?value ?T.
filter ?T (t>="01/01/2019" and t<="31/03/2019")
}
GROUP BY (?attractions)
ORDER BY avg(?value)
```

Listing 3.3: An example of stSPARQL query.

Similar to stSPARQL, the query language SPARQL-ST [142] also introduces spatial and temporal variables. In particular, the spatial variables (denoted by a % prefix) are used to represent the complex spatial features. In [142], the authors also extend the concept of SPARQL mapping in order to map a spatial variable to a set of triples that represents the required spatial information. Similarly, temporal variables (denoted by a # prefix) are mapped to time intervals and can appear in the fourth position of a temporal triple pattern in the style of [71]. Furthermore, in SPARQL-ST, two special filters are introduced, namely SPATIAL FILTER and TEMPORAL FILTER. These filters are used to filter the results with spatial and temporal constraints (OGC Simple Feature Access topological relations and distance for the spatial part, and Allen's interval calculus [8] for the temporal part). In order to enable the realization of this query language, the used spatial and temporal operators need to be implemented. Despite the fact that SPARQL-ST is an extension of SPARQL and is designed for querying RDF graphs, the ontology and data, however, are stored

```
SELECT ?attractions (avg(?value) as ?avg)
WHERE{
?entity skos:broader dbr:Category:Tourism_by_city .
?places skos:broader ?entity .
?attractions dcterms:subject ?places .
?attractions dbo:country ?country .
?country dbo:location dbr:Europe.
?attractions stt:located_at %g1.
?sensor a sosa:Sensor;
        sosa:isHostedBy ?station;
        sosa:observes iot:AirTemperature.
?station a got:WeatherStation;
?station stt:located_at %g2.
SPATIAL FILTER(within(%g1,%g2)).
?obs sosa:madeBySensor  ?sensor;
         sosa:hasSimpleResult ?value #t.
TEMPORAL FILTER(during(#t,interval(01/01/19,31/03/19,DD/MM/YY)))
}
GROUP BY (?attractions)
ORDER BY avg(?value)
```

Listing 3.4: An example of SPARQL-ST query.

```
#return the average air temperature value of 2019.
SELECT (avg(?value) as ?avg)
FROM SNAPSHOT 2019
WHERE{
?sensor a sosa:Sensor;
        sosa:observes iot:AirTemperature.
?obs sosa:madeBySensor  ?sensor;
         sosa:hasSimpleResult ?value.
}
```

Listing 3.5: An example of t-SPARQL time point query.

in a relational database. Therefore, to execute the SPARQL-ST query, a given query needs to be translated to the corresponding SQL query prior to execution. The first version of SPARQL-ST is built on top of Oracle's DBMS. This consequently leads to several drawbacks, i.e, reasoning and linked data modeling are difficult to apply (model semantic data in RDF is much easier than in SQL). Listing 3.4 illustrates our aforementioned motivation query in SPARQL-ST language.

t-SPARQL [189] is also another extension of SPARQL with additional operators for expressing temporal queries. In fact, this is a temporal language that is built upon the temporal named graph model [189], previously mentioned in Section 3.1.2. Similar to SPARQL-ST, the t-SPARQL queries are translated to SPARQL prior to execution. To retrieve the temporal information, the language supports two types of temporal queries: *time point query* and *temporal query*. The *time point query* supports retrieving information which is valid at a specified point in time, while *temporal query* is used for both wild-card intervals and time points. These wild-cards can be used to bind a variable to the validity period of a triple or to express temporal relationships between intervals. t-SPARQL allows one form of temporal wild-cards *[?s, ?e]* which binds the literal start time (?s) and end time (?e) values. Note that, t-SPARQL limits to retrieving temporal information and does not take into account the spatial aspect. Examples of time-point and wild-cards queries are shown in Listings 3.5, 3.6 respectively.

```
#return the average air temperature value for the time period 01/01/19 to 31/03/19.
SELECT (avg(?value) as ?avg)
WHERE{
?sensor a sosa:Sensor;
         sosa:observes iot:AirTemperature.
?obs sosa:madeBySensor  ?sensor.
[01/01/19,31/03/19]?obs sosa:hasSimpleResult ?value.
}
```

Listing 3.6: An example of t-SPARQL temporal query.

3.2.1 GeoSPARQL query language

The aforementioned GeoSPARQL data models [19] have provided the corresponding SPARQL-like query language. The major contribution of the GeoSPARQL language is providing a standard and flexible way that ask for relationships between spatial objects. These come in the form of topological binary properties which can be used in SPARQL query triple patterns as a normal predicate. These spatial predicates are then translated by the query rewrite component based on the predefined rules. Normally, these properties aim to ask the relationships between objects of the *geo:Geometry* class. However, with the support from the provided GeoSPARQL's query rewrite rules, they can also be used for the *geo:Features* objects, or between *geo:Features* and *geo:Geometries*. Several topological relations are supported: *equals*, *disjoint*, *intersects*, *touches*, *within*, *contains*, *overlaps*, and *crosses*. Listing 3.7 illustrates the GeoSPARQL query that uses topological relation (*geo:sfWithin*) in query triple pattern.

```
SELECT ?attractions
WHERE{
?entity skos:broader dbr:Category:Tourism_by_city .
?places skos:broader ?entity .
?attractions dcterms:subject ?places .
?attractions dbo:country ?country .
?country dbo:location dbr:Europe.
?attractions geo:lat ?lat.
?attractions geo:long ?long.
?attractions geo:hasGeometry ?attrLoc.
?station a got:WeatherStation;
        geo:hasGeometry ?stationLocation.
?stationLocation geo:sfWithin ?attrLoc.
}
```

Listing 3.7: An example of GeoSPARQL query example that uses topological binary properties as predicate.

Besides the topological binary properties, GeoSPARQL also provides a set of SPARQL extension functions for spatial computation. These functions can be used in FILTER clause for different purposes. For example, they can take multiple geometries as predicates and produce either a new geometry or another datatype as a result. Another functionality of these functions is to test the topological relation between the geometries, which returns a Boolean value. Listing 3.8 presents an example of the boolean topological functions, namely *ogcf:intersects* , which returns true if two geometries intersect.

```
SELECT ?sensorLocation
WHERE{
?sensor a sosa:Sensor;
        sosa:isHostedBy ?sensorLocation;
        sosa:observes iot:AirTemperature.
FILTER(ogcf:intersects(?sensorLocation,"POLYGON((-77.089005 38.913574,-77.029953 38.913574,
-77.029953 38.886321,-77.089005 38.886321,-77.089005 38.913574))"^^ogc:WKTLiteral))
}}
}
```

Listing 3.8: GeoSPARQL query example using spatial function.

The GeoSPARQL has become a standard SPARQL query language with spatial computation support and has been used in many European projects. Our proposed spatio-temporal query language also adopts the GeoSPARQL query language syntax, which will be presented in Chapter 6.

3.3 PUBLISHING LINKED SENSOR DATA ON LOD CLOUD

The lack of research in an area of managing Linked Sensor Data will draw more attention from the SW community if there are more valuable sensor datasets will be made available in the LOD cloud. In this section, we will review some existing works that publish their sensor data on LOD cloud.

Patni et al. [139] is a state of the art that publishes to LOD cloud a large dataset of sensor descriptions and measurement. The dataset includes observations within the entire United States during the time periods that several major storms were active, including Hurricane Katrina, Ike, Bill, Bertha, Wilma, Charley, Gustav, and a major blizzard in Nevada in 2002. To publish this sensor dataset, the authors firstly represent the raw data in Observation and Measurements standard [41] and then transform it to RDF. The transformation process results in a RDF dataset that consists of 1 billion triples. The limitation of this sensor dataset is a lack of interlinking with the external data source such as LinkedGeoData [16] or DBpedia [128].

In [17], Barnaghi and Presser describe a LD platform, called Sensor2Web, for publishing sensor data and linking them to existing resource on the semantic Web. The authors use stylesheets to transform the raw sensor data in XML format to different designated sensor description formats such as W3C SSN ontology[40] or SENSEI ontology [194] representations. The platform offers an interface for publishing Linked Sensor Data without requiring the users to have a semantic technology background. However, the user is requested to manually enter relevant keywords that describe the sensors for obtaining a list of suggested concepts from online repositories. No statistics of the dataset are given.

Similarly, the Australian Open Government Data[9] [112] has recently published a homogenized daily temperature dataset, namely ACORN-SAT, for the monitoring of climate variability and

9 http://data.gov.au

change in Australia. The publisher employs the latest analysis techniques and takes advantage of newly digitized observational data to provide a daily temperature record over the last 100 years. In ACORN-SAT, data are transformed to RDF based on the joint use of RDF Data Cube vocabulary [44] and SSN ontology. The dataset consists of 61 million triples and is enriched by linking with external datasets such as GeoNames.

Along the same lines, in Chapter 4, we describe our linked meteorological sensor dataset as a case study in modeling and publishing Linked Sensor Data on the LOD cloud. Our dataset is based on the ISH dataset, which is originally published by NCDC. In our work, we employ a publisher-subscriber mechanism in all processes, from the data collection to the data transformation. This technique enables us to make use of our physical cluster so that we can efficiently collect, transform and publish sensor data in parallel. In comparison with the aforementioned datasets, the advantages of our dataset are its scale and density. While the existing ones provide data at the national level, our dataset offers the meteorological observations in RDF format for all over the world. Additionally, the size of our dataset is also much larger (26 billion triples) in comparison with others. Details of our linked meteorological dataset will be reported in Chapter 4.

3.4 TRIPLE STORES AND SPATIO-TEMPORAL SUPPORTS

During the last decade, we have witnessed a rapid increase in the number of RDF store implementations that have been proposed [1, 50, 23, 131, 76, 178, 153]. Among them, RDF-3x [133], Hexastore [196] and gStore [204] are the state-of-the-art RDF stores. In these systems, the RDF data are well organized and indexed to efficiently and effectively answer the RDF queries. Unfortunately, since the stores are well-designed and none of them takes spatial or temporal features into consideration, all the systems are unsuitable for managing spatio-temporal sensor data without great modification. For example, in [29], the authors exploit RDF-3X [131] to build a spatial feature integrated query system. They employ the separated R-tree and RDF-3X indices for filtering the entities exploiting the spatial and the semantic features respectively. Similarly, YAGO2 [113] is extended to support spatial feature over statements and also provides an interface for querying SPARQL-like queries over YAGO2 data. Because such approaches are not intended to be a geographical or temporal database in the first place, they limit the user using very few hard-code spatial relationships, such as *north_Of*, *east_Of*, *south_Of*, etc, and do not support other complex ones (i.e, *within*, *intersects*). For temporal query, instead of building a dedicated temporal index, a special handling way is introduced. For that, temporal data are treated as standard RDF literals. To query these data, the temporal filter will be defined in the standard semantic SPARQL FILTER expression, using the basic comparison operators on the time values.

Toward supporting spatial queries, native RDF stores such as RDF4J[10] (former Sesame) and Jena[11] have recently enabled spatial querying by making use of Apache Lucence[12] spatial in-

10 http://rdf4j.org
11 https://jena.apache.org/
12 https://lucene.apache.org/core/

dex. Similarly, Virtuoso[13] and Stardog[14] implement their own spatial indices to support spatial query. However, all of these stores are mostly focus on querying the spatial data and do not fully support temporal analytical query on Linked Data. Specifically, temporal data with the timestamp are treated as standard RDF literal, hence, query on temporal data limits to basic SPARQL date-time functions. Taking the temporal query into consideration, Starbon[108] provides both spatial and temporal indices. However, its performance is quite slow when the data size reaches over 1 billion triples. The query performance comparison of these systems, as well as their query processing behaviors, will be reported and further analyzed in our performance study presented in Chapter 5.

Along the same lines of supporting spatial query, Brodt et al.[28] utilize RDF query engines and spatial index to manage spatial RDF data. [28] uses RDF-3x as the base index and adds a spatial index for filtering entities before or after RDF-3x join operations. Another example is OWLIM[101], which supports geospatial index in its Standard Edition. However, none of them systematically address the issue of elasticity and scalability to deal with the massive volume of sensor data. The technical detail and the index performance are also not mentioned in such systems.

In the light of dealing with performance and scalability of RDF stores, many centralized and distributed RDF repositories have been implemented to support storing, indexing and querying RDF data, such as Clustered TDB[137], Inkling[124], RDFStore [15], 4Store[16]. These RDF repositories are fast and able to scale up to many millions of triples or a few billions of triples. However, none of the systems takes the spatio-temporal feature into consideration.

Taking such short-comings into consideration, we propose EAGLE, a query engine that is able to support complicated spatio-temporal queries tailor of managing Linked Sensor Data while the engine is capable of dealing mentioned performance and scalability issues. The design of EAGLE is presented in Chapter 6.

3.5 RDF STORE ASSESSMENT

Along with the growth of RDF store implementations, there is a corresponding increase in studies interested in looking into their performance and data processing behaviors. For that reason, the number of performance assessment techniques specify to these RDF stores have been introduced. For example, in [115], the authors provide a performance comparison of seven selected RDF stores over the synthetic Lehigh University Benchmark (LUBM) dataset [69]. This evaluation aims to test the data loading and query performance of these stores with respect to a large data application. Similarly, Rohloff et al. [162] present a performance evaluation over the LUBM dataset of AllegroGraph, Jena, and Sesame with various storage backends (such as MySQL, DAML DB[138], BigIOWLIM, etc). In this work, a set of evaluation metrics is presented such as the data loading time, query execution performance, query completeness and soundness, and storage size requirements. Unlike [115, 162] that compares the performance of different RDF

13 https://github.com/openlink/virtuoso-opensource
14 http://stardog.com
15 http://rdfstore.sourceforge.net
16 http://4store.org

stores, the evaluation described in [169] focuses on the experimental comparison of a single native triple store (Sesame) and a vertically partitioned scheme for storing RDF data in a relational database on top of the SP2Bench SPARQL benchmark [170]. In [24], the authors introduce the Berlin SPARQL Benchmark (BSBM) for comparing the performance of native RDF stores, non-RDF relational databases and SPARQL-to-SQL rewriters. The benchmark provides valuable insights into system behavior by comparing the data loading time, the number of mixed queries executed per hour, etc.

Regarding a performance study of RDF stores over spatial RDF data, Kolas [105] presents a state-of-the-art assessment for querying geospatial data encoded in RDF. In this work, the author extends the LUBM dataset by adding spatial entities so that they were able to evaluate a spatial search on the geo-enabled RDF stores. Along the same lines, Garbis et al. [57] developed a benchmark, called Geographica, which uses both real-world and synthetic datasets to test the offered functionality and the performance of some prominent geospatial RDF stores. For evaluating temporal RDF data, the authors in [54] perform an empirical evaluation of various current archiving techniques and querying strategies on this data. The aim of this work is to serve as a baseline of future developments on querying temporal RDF data.

Despite the wide range of performance assessments between RDF stores, to the best of our knowledge, there is no existing approach that focuses on studying the readiness of RDF stores as regards Linked Sensor Data. In comparison with traditional RDF data, sensor data is usually associated with spatial and temporal contexts. This distinctive characteristic, therefore, poses challenges for the current RDF store implementations due to the requirements of geospatial supports, temporal filtering, and full-text search. In this regard, in Chapter 5, we present our performance study of RDF stores for Linked Sensor Data. Rather than providing another benchmarking effort, our study should be read as an empirical study to find the gaps and shortcomings with current RDF store technologies so that they can be properly applied for the management of Linked Sensor Data. Following this direction, our evaluation and analysis aim to help guide the development of RDF stores, rather than serving as a comparison of existing ones.

3.6 SUMMARY

In this chapter, we primarily present the related work to the research in this thesis, focusing on sensor ontologies, spatio-temporal representation for sensor data, spatio-temporal query languages, linked sensor datasets, RDF stores and performance evaluations. In the first two sections, we review the existing data model and query language that can be applied for modeling and querying Linked Sensor Data. For this, we analyze the use of existing sensor ontologies and then discussed the limitations for presenting spatial and temporal aspects of sensor data. Query languages associated with these models were also presented, supported by concrete query examples.

The third section discusses the existing work for publishing Linked Sensor Dataset on the LOD cloud. With regard to these works, in conjunction with the above analyses on sensor ontologies, support us to present the modeling and publishing process of our linked meteorological dataset in Chapter 4. The last two sections of this chapter are about the current generation of RDF

stores with spatio-temporal supports and their performance evaluation. In these sections, we generally discuss the restrictions of RDF stores applied in Linked Sensor Data context, hence, emphasize a need for an effective scalable solution for managing and querying Linked Sensor Data presented in Chapter 6. A more detailed discussion and performance evaluation of RDF stores with spatio-temporal supports will be given in Chapter 5.

4 A CASE STUDY IN MODELING AND PUBLISHING LINKED METEOROLOGICAL DATASET

The lack of research in the area of spatio-temporal Linked Data will draw more attentions when more datasets with such characteristics will be made available in the LOD Cloud. In this chapter, we describe the generation process of our linked meteorological dataset, which can be considered as a case study in modeling and publishing linked sensor dataset on the LOD clould. The dataset consists of a numerous amount of ISH meteorological observation data that were collected from more than 20,000 stations on all over the world during the time periods from 2008 to 2018. In conjunction with the data publishing processes, we also present the data modeling approach to semantically describe the ISH sensor data. In this regard, we follow an iterative approach based on the reuse of existing ontologies in sensor network area and the domain extension specified for meteorological data. Moreover, the URI design principles for our linked meteorological dataset are also elaborated. After that, we describe our data publishing processes including the processing workflows and the data enrichment phase. Next, we present the dataset exploitation use cases and give the summary at the end of this chapter. The dataset appears in [157, 146].

4.1 OVERVIEW OF ISH DATASET

The ISH dataset is a digital dataset DSI-3505 which has been developed since 1998 as a joint activity of NOAA's National Climate Data Center (NCDC), US Air Force Combat Climatology Center (AFCCC) and US Navy's Fleet Numerical Meteorological and Oceanographical Command Detachment (FNMOD). This data source is composed of worldwide surface meteorological observations which are collected from about 20,000 stations in numerous sources, including various data formats which were key-entered from paper forms, into a single common format and common data model. Consisting of over 1.7 billion surface observations since 1900s and has been used by numerous customers in many applications, the ISH dataset has become the most complete archive and well-known meteorological information.

In general, the ISH dataset has been developed through an iterative process [117], which can be split into two main phases, namely *database build* phase and *quality control* phase. The *database build* phase consists of the development of the integrated format, the data collection, the conversion of observation data from various sources and the development of extensive metadata to drive the processing and merging. The final phase, which is the quality control, aims to improve the overall quality of the dataset. This phase includes the research, development, and program-

ming of algorithms to correct the random and systematic errors in the data before publishing it to the real-world.

After the development process, the ISH observations are stored in one consistent format for the full period of record and are archived on NCDC's Hierarchical Data Storage System (HDSS, tape-robotic system). The dataset is operationally updated on a routine basis. In order to make the data source more accessible and convenient for end-user clients, the NCDC encodes the dataset in a special format so that the dataset size can be reduced significantly. After having the dataset downloaded, the users have to decode the data via a decoding algorithms provided by NCDC. Data are extracted into a readable simplified text form. Figure 4.1 presents an example of ISH data in readable format.

```
  USAF  WBAN YR--MODAHRMN DIR SPD GUS CLG SKC L M H  VSB MW MW MW MW AW AW AW AW W TEMP DEWP    SLP   ALT    STP
723415 03962 201601010053 050   3 *** 722 CLR * * * 10.0 ** ** ** ** ** ** ** ** *   40   32 1029.5 30.39 1008.7
723415 03962 201601010153 020   3 *** 722 CLR * * * 10.0 ** ** ** ** ** ** ** ** *   42   32 1030.1 30.41 1009.4
723415 03962 201601010253 030   3 *** 722 CLR * * * 10.0 ** ** ** ** ** ** ** ** *   41   32 1030.0 30.41 1009.4
723415 03962 201601010353 020   5 *** 722 CLR * * * 10.0 ** ** ** ** ** ** ** ** *   41   31 1030.0 30.41 1009.4
723415 03962 201601010453 990   6 *** 722 CLR * * * 10.0 ** ** ** ** ** ** ** ** *   42   31 1030.3 30.42 1009.7
723415 03962 201601010553 ***   0 *** 722 CLR * * * 10.0 ** ** ** ** ** ** ** ** *   40   32 1030.2 30.42 1009.7
723415 03962 201601010559 *** *** *** *** *** * * * **** ** ** ** ** ** ** ** ** * **** **** ****** ***** ******
```

Figure 4.1: An exmaple of ISH-formatted file

As illustrated in Figure 4.1, each line of the ISH file is an observation data record. Each record comprises 26 data fields and values. Table 4.1 presents the description of ISH observation data fields and example values. A more detailed description can be found in the original ISH format document [1]. In addition to the observation data, NCDC also provides the description of the sensor stations that generate the observation data[2]. The station description includes the station identifiers number, station name, geographical coordinates, country, state and the operational station history (start and end date). The station identifier is assigned by NCDC which currently appears as an 11-digit numerical field in positions 5 - 15 (USAF, WBAN) in ISH observation record.

Table 4.1: ISH data field description

Field	Position	Value	Description
USAF	5-10	723415	Fixed-weather-station USAF master station catalog identifier
WBAN	11-15	03962	Fixed-weather-station NCDC WBAN identifier
YR–MODA	16-23	20160101	The date of observation in the format YYYYMMDD
HRMN	24-27	0053	The time of observation in the format HHMM
DIR	61-63	050	Wind-observation direction angle
SPD	66-69	3	Wind-observation speed rate (meters per second).
VSB	79-84	10.0	Visibility-observation distance dimension (meters)
TEMP	88-92	40	Air temperature (Degrees Celsius)
...	...	...	...

4.2 A LINKED DATA MODEL FOR ISH DATASET

The development of our proposed LD model for semantically describing the ISH dataset has been performed by following an iterative approach based on the reuse of existing ontologies

1 http://www1.ncdc.noaa.gov/pub/data/ish/ish-format-document.pdf
2 ftp://ftp.ncdc.noaa.gov/pub/data/noaa/isd-history.txt

Table 4.2: Involved ontologies and namespaces in proposed linked data model

Prefix	Namespace URI	Description
got	http://graphofthings.org/ontology/	Our domain extension ontology to describe meteorological concepts
got-res	http://graphofthings.org/resource/	The GoT resource namespace
sosa	http://www.w3.org/ns/sosa/	Sensor, Observation, Sample, and Actuator (SOSA) ontology
ssn	http://www.w3.org/ns/ssn/	Semantic Sensor Network ontology
time	http://www.w3.org/2006/time#	Time Ontology in OWL
geo	http://www.opengis.net/ont/geosparql#	An RDF/OWL vocabulary for representing spatial information.
xsd	http://www.w3.org/2001/XMLSchema#	Schema namespace as defined by XSD.
rdf	http://www.w3.org/1999/02/22-rdf-syntaxns#	This is the RDF Schema for the RDF vocabulary terms in the RDF Namespace, defined in RDF 1.1 Concepts.
rdfs	http://www.w3.org/2000/01/rdf-schema#	RDF Schema provides a data modelling vocabulary for RDF data.
owl	http://www.w3.org/2002/07/owl#	This ontology partially describes the built-in classes and properties that together form the basis of the RDF/XML syntax of OWL 2.

in the sensor network area. More specifically, we construct our Linked Sensor Data model by combining the usage of the SSN ontology [40], SOSA ontology [93] in conjunction with the W3C Time ontology [87] and OGC GeoSPARQL [19]. Furthermore, we also extend the model with meteorological concepts in order to describe the ISH sensor data. List of involved ontologies and namespaces are presented in Table 4.2. Details of using these reference ontologies, the domain extensions, and URI design principles are discussed in the following subsequent sections.

4.2.1 Semantic Properties

The semantic properties of sensor data provide not only the general description of sensor but also the connections between sensor data and the domain knowledge. Several semantic properties describe the sensor data can be listed here, such as platform, sensor type, observable property, a feature of interest, etc. We propose using the W3C recommended sensor ontologies, called SOSA/SSN [40, 93] , to describe the semantic properties of sensor data. Figure 4.2 illustrates the modular structure consisting of SOSA/SSN as the central ontologies and connections with the domain ontologies. The description of the core classes and properties that are used to annotate the semantic properties of sensor data are described as follows.

- **sosa:Sensor** - Conceptually, a sensor is an abstract concept that implies a physical device, agent (including humans), or software (simulation) involved in, or implementing, a (sensing) procedure. A sensor can generate a sensing result and can be mounted on platforms, e.g., a modern smartphone hosts multiple sensors. In SOSA/SSN ontologies, the *sosa:Sensor* class lies at the central position from where any subdomain category of the sensor can be conceptualized.
- **sosa:Platform** - The *sosa:Platform* class is used to describe a complex system which carries at least one Sensor, Actuator, or sampling device to produce observations, actuations, or samples. A platform can also have geometric properties, i.e., placement, of sensors in relation to one another. A sensor attaches to the platform through *sosa:isHostedBy* property. In our meteorological dataset, a sensor station is defined as a platform.
- **sosa:Observation** - Observation is the outcome of the sensing activity of a sensor. Specifically, an observation involves a Sensor (*sosa:madeBySensor*) and yields a Result

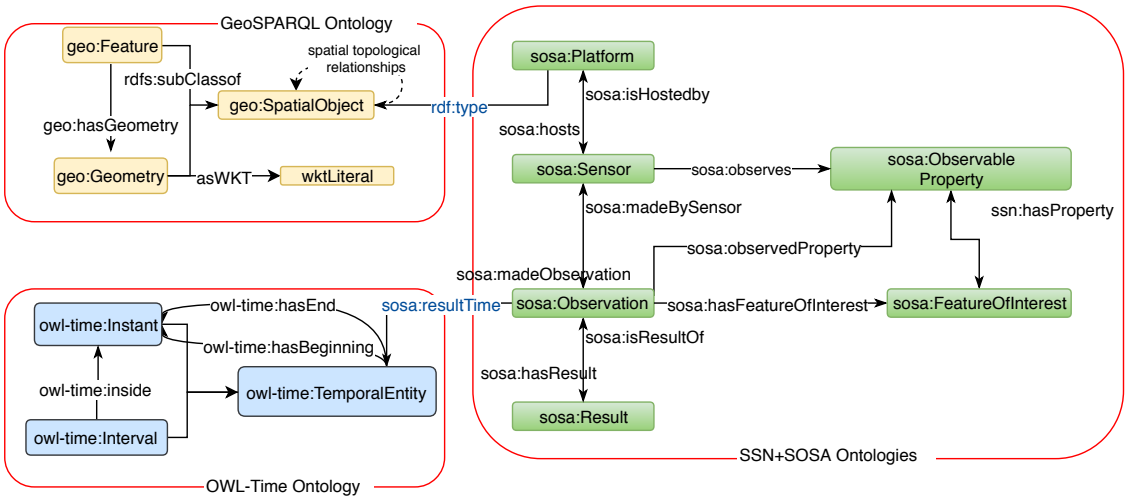


Figure 4.2: The core classes and properties of Linked Sensor Data model

```
got-res:Station/u4eu1epv0n_ish_1001499999 a got:WeatherStation;
              a sosa:Platform;
              sosa:hosts <sensorA>.
<sensorA> a sosa:Sensor;
        sosa:isHostedBy got-res:Station/u4eu1epv0n_ish_1001499999;
        sosa:observes got:AirTemperatureProperty.
<Obs123> a sosa:Observation;
        sosa:madeBySensor <sensorA>;
        sosa:observedProperty got:AirTemperatureProperty;
        sosa:hasSimpleResult "30";
        sosa:resultTime "2019-01-19T23:00:00Z"^^xsd:dateTime.
```

Listing 4.1: Semantic modeling of ISH sensor.

(*sosa:hasResult*). While SOSA relies on QUDT [88] or other vocabularies to describe observation results and their values, an additional datatype property is provided to handle the simple case that merely requires a typed literal, via the *sosa:hasSimpleResult* property.

- **sosa:ObservableProperty** - An observable property is defined as an observable quality of a thing, typically a feature of interest. Observable properties are similar to procedures or units of measure in the sense that they are singletons. One observable property will apply to many acts of observation, concerning different features of interest, at different times, or using different procedures or sensors, e.g.

- **sosa:Result** - This class describes a result of observation, actuation, or sampling, e.g., the value for an observed property of some feature of interest, such as 20 Celsius for the current temperature of the room.

- **sosa:FeatureOfInterest** - This is used to point to the observed feature of interest. A feature of interest can be any observed real-world phenomenon.

Listing 4.1 illustrates an example for describing the semantic properties of an ISH sensor and its observation data using SSN/SOSA ontologies. In this example, we also use some specific

domain concepts, such as *got:WeatherStation* and *got:AirTemperatureProperty*, which are defined in our meteorological domain extension ontology presented in the next section.

4.2.2 Meteorological Domain Extension for ISH Sensor Data

The SOSA/SSN ontologies represent sensor data at an abstract level, thus, in order to explicitly describe the ISH data, we need to extend the SOSA/SSN ontologies with the meteorological domain concepts. In this regard, we first collect all the meteorological-related terms in ISH dataset and define them as ontology classes and properties accordingly. The additional classes and properties are hosted in our separated domain extension ontology, namely GoT. Figure 4.3 illustrates these additional meteorological concepts and how they are connected to the SOSA/SSN ontologies.

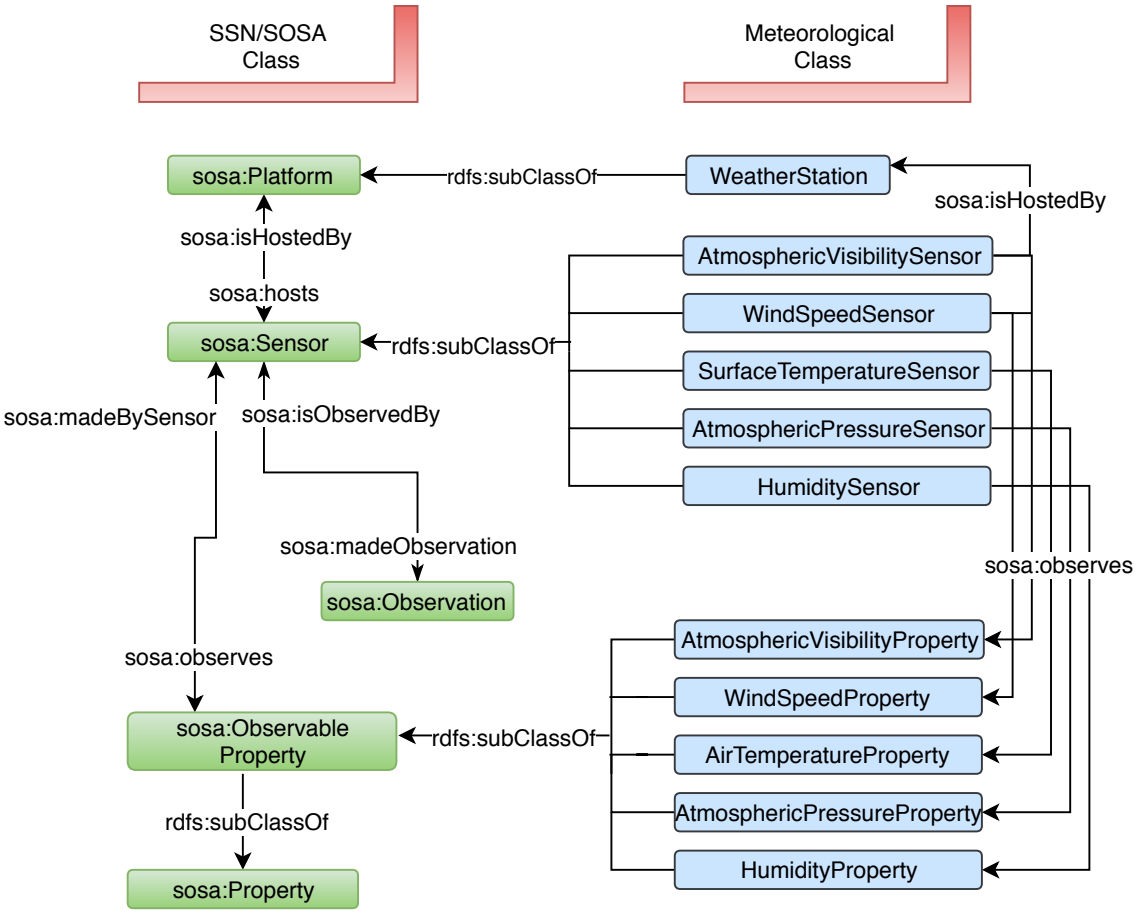


Figure 4.3: A SSN/SOSA meteorological domain extension model

As mentioned in Section 4.1, the ISH observations are generated by a set of sensors deployed on a weather station. In our proposed data model, a weather station is defined as a sub-concept of the *sosa:Platform* class in SOSA ontology. Therefore, to conceptualize this concept, we define the *got:WeatherStation* class linking to *sosa:Platform* as its subclass. After that, we extend the concept *sosa:Sensor* by means of a hierarchy of types of meteorological sensors used by ISH

dataset. Theses meteorological sensor classes are: *AtmosphericVisibilitySensor*, *WindSpeedSensor*, *SurfaceTemperatureSensor*, *AtmosphericPressureSensor* and *HumiditySensor*. Similarly, the measurement concept *sosa:ObservableProperty* has been also extended and populated accordingly to the observable properties gathered in ISH observations, i.e, *HumidityProperty*, *AirTemperatureProperty*, etc.

4.2.3 Spatial Properties

Spatial property of a sensor is its location which can be very specific geo-locations defined as latitude and longitude or high level information such as post-codes, address. To describe the sensor spatial properties, we reuse the OGC GeoSPARQL ontology [19] that defines vocabularies for representing geospatial data. As shown at the top left of Figure 4.2, the OGC GeoSPARQL core classes (*SpatialObject*, *Feature*, and *Geometry*) are imported to represent the ISH station location. We paraphrase the definitions of these spatial classes as follows:

- **geo:Feature** - A thing that associates with spatial information, i.e, a visited place, a weather station, etc.
- **geo:Geometry** - A representation of a spatial location, i.e, a set of coordinates.
- **geo:SpatialObject** - An abstract class of both *geo:Feature* and *geo:Geometry*.

Because a sensor station is always associated with a specific spatial location, we establish a relation between the *sosa:Platform* class and *geo:SpatialObject* class. For this purpose, we add a property *rdf:type* between these classes. This relation allows us to connect a semantic entity with its geospatial properties. Moreover, additional spatial properties of the semantic entity can consequently be added through the associate SpatialObject. For example, the object properties of GeoSPARQL such as covers, crosses, meets, and within, can be added to determine topological relations between the SpatialObject location and the request area. Listing 4.2 shows an example of using GeoSPARQL to describe the spatial properties of an ISH weather station.

```
<sensor123> a sosa:Sensor;
        sosa:isHostedBy got-res:Station/u4eu1epv0n_ish_1001499999;
        sosa:observes iot:AirTemperature.
got-res:Station/u4eu1epv0n_ish_1001499999 a geo:Feature;
        geo:hasGeometry <pointA>.
<pointA> a geo:Point;
    geo:asWKT "POINT(-77.022 38.2433)"^^geo-sf:wktLiteral.
```

Listing 4.2: Representing sensor location by using GeoSPARQL ontology.

4.2.4 Temporal Properties

To model the temporal data such as sensor observation value, a portion of OWL-Time ontology (mainly *Temporal Entity* and its subclasses, Time instant and Time Interval) is applied as the time

```
ex:observedTime
  a                    :Instant ;
  :inDateTime          ex:observedTimeDescription ;
  :inXSDDateTimeStamp  2018-07-25T10:30:00+10:00 .

ex:observedTimeDescription
  a             :DateTimeDescription ;
  :unitType     :unitMinute ;
  :minute       30 ;
  :hour         10 ;
  :day          "---25"^^xsd:gDay ;
  :dayOfWeek    :Wednesday ;
  :dayOfYear    206 ;
  :week         30 ;
  :month        "--07"^^xsd:gMonth ;
  :monthOfYear  greg:July ;
  :timeZone     <https://www.timeanddate.com
                              /time/zones/aest> ;
  :year         "2018"^^xsd:gYear .
```

Listing 4.3: Using OWL-Time for describing sensor observation result time

property of the semantic entity. In our data model, the OWL-Time ontology connects to the SOSA ontology via *sosa:resultTime* property of *sosa:Observation* class. The time interval or time instance associated with this relationship denotes the time at which the observation data is valid. An example of using OWL-Time is illustrated in Figure 4.4 and Listing 4.3.

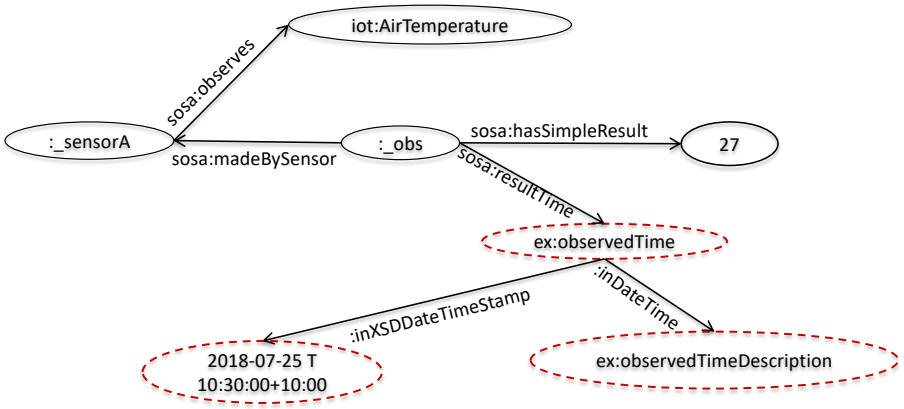


Figure 4.4: Representing temporal dimension of sensor data via OWL-Time ontology

4.2.5 URI Design Principles for GoT Ontology and Resources

The most common challenge issue for the Linked Data community when building an ontology and publishing its dataset is the URI design. To address this, there has been a lot of discussions and work on defining standard guidelines for the effective use of URIs [3] [4]. As a result, several principles have been derived from these discussions, namely *simplicity*, *stability* and *manageability*. In this section, where we draw upon the ideas in [13], we present our URI design decisions and

3 http://www.w3.org/Provider/Style/URI
4 http://www.w3.org/TR/chips/

the conventions followed by these principles for our meteorological domain extension ontology and its generated resources.

BASE URI STRUCTURE. The base URI, that is applied to all elements of the GoT ontology and its resources, is `http://graphofthings.org/`, prefixed as ***got***. After having the base URI defined, it is also necessary to specify a pattern that helps to separate the classes and resources of an ontology. Therefore, two separated URI schemes have been introduced, respectively: `http://graphofthings.org/ontology/` for our domain extension GoT ontology, and `http://graphofthings.org/resource/` for its generated resources.

DOMAIN EXTENSION URIS. As previously mentioned, the base URI scheme for our domain extension ontology is `http://graphofthings.org/ontology/`, to which the concept name is appended. In an ontology, concepts can be categorized into main types, namely *class* and *property*. Therefore, we design the ontology URIs pattern as `http://graphofthings.org/ontology/{class|property}`. Note that, the naming for each type of element is also differentiated. For example, the class name is started with an upper case (i.e, *got:WeatherStation*) while the property name is started with lower case (i.e, *got:hasGeometry*).

GENERATED RESOURCES URIS. Designing the URI pattern for generated resources requires special attention to ensure that every distinct resource is assigned by a unique URI. This aims to mitigate the well-known URI design issues such as *co-reference* [92] and *instance unification problem* [12]. In this regard, we have taken a set of URI design principles into consideration throughout the URI design development process:

- **Patterned URIs solution** [49]: This pattern solution is selected due to its human-readable format. Moreover, adding the class name to the base URI mitigates the co-reference problems of generating different distinct individuals with the same local identifier but different class.
- **Natural Keys patterns**[5]: This solution is used to model the URI identifiers by using the text property of resource such as name, label, etc. The selected text property will be converted by using the urify pattern[6] which applies an URL encoding and converts the spaces to underscores. An example of natural keys patterns is `http://graphofthings.org/resource/FeatureOfInterest/galway_eye_square`.
- **Composite Pattern**. This is an ad-hoc URI structure to uniquely identify the resources, where the local ID is formed by several interconnected pieces of information related to the resource being identified. For example, we append the *geohash prefix* to the URI used to identify the sensor, place or observation resources.

A summary of our URI design patterns can be found in Table 4.3. In this table, we present different URI design patterns for each type of resources, according to the aforementioned design decisions and patterns.

5 http://patterns.dataincubator.org/book/ natural-keys.html
6 http://d2rq.org/d2rq-language#dfn-uri-pattern

Table 4.3: URI Design Patterns

Resource class	URI pattern	Example
sosa:WeatherStation	got-res:<ClassName>/<geohash>_<source>_<stationLocalID>	got-res:WeatherStation/gu9gdby_ish_10010
sosa:AirTemperatureSensor	got-res:<ClassName>/<geohash>_<source>_<stationID> _<sensorID>	got-res:AirTemperatureSensor/gu9gdby_ish_10010
sosa:Observation	got-res:<ClassName>/<geohash>_<observableProperty> _<sensorID>_at_<timestamp>	got-res:Observation/gu9gdby_AirTemperature_10010_at _1528557797259
time:Instant	got-res:<ClassName>/timestamp	got-res:Instant/1528557797259

4.3 RDF DATASET GENERATION

Our data generation workflow comprises different steps, as shown in Figure 4.5. Details of these steps are described as follows:

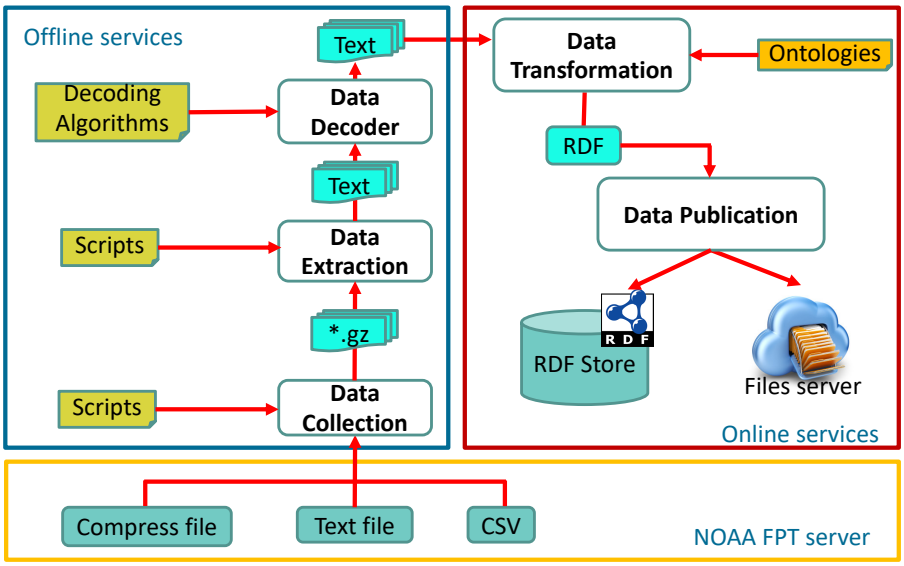
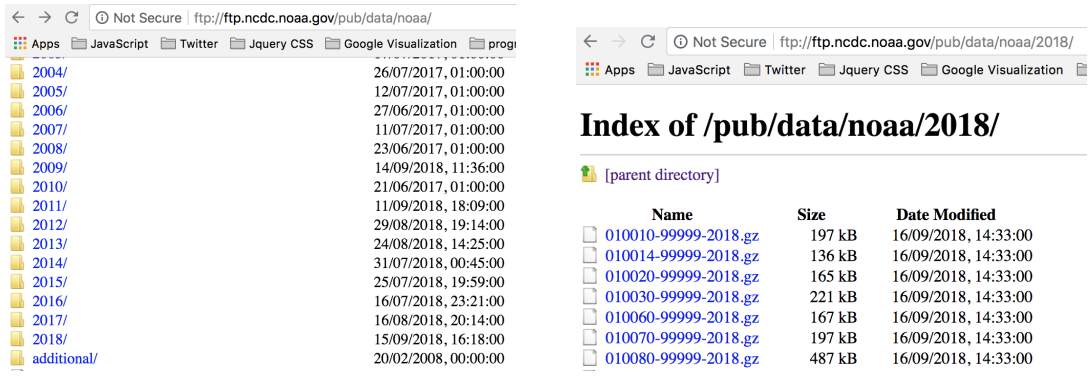


Figure 4.5: Linked meteorological data generation workflow

Data Collection - The first step of the RDF Data generation process is to identify and access the ISH meteorological data sources through the NCDC FTP service. NCDC provides a public FTP service so that the users can access and download the historical ISH dataset and the latest hourly updated data. Due to our restricted infrastructure, we only collect the data from 2008 to 2018. Figure 4.6 presents the screenshots of NCDC server and example ISH data in year 2018. As illustrated in the figure, the ISH observation data are categorized by year and stored in separated folders. The latest updated data are kept in the *Additional* folder. We collect the ISH dataset via a dedicated Java script which is set to automatically execute for every 3 hours.

Data Extraction - Once the ISH observation data have been collected, we extract them into a textual ASCII encoded format. The ASCII encoded text files are then stored in a file server as a security back-up. The accessed URLs of these files are sent later to the **Data Decoder** processes which run on a Publish-Subscribe system.

Data Decoder - This step involves using the decoding algorithm provided by NCDC FTP service to decode the ISH ASCII encoded data to readable text form. Figure 4.7 describes the decoding process which is established on a Publish-Subscribe system. As mentioned above, the input of the publisher side is the encoded file access URL produced by the Data Extraction step. At the



(a) NCDC FPT Server Snapshot

(b) Example of ISH Data in 2018

Figure 4.6: NCDC FPT Server

subscriber side, we deploy multiple workers that can work in parallel to speed up the decoding process. The output of this process is the list of decoded observation data, as illustrated in Figure 4.1.

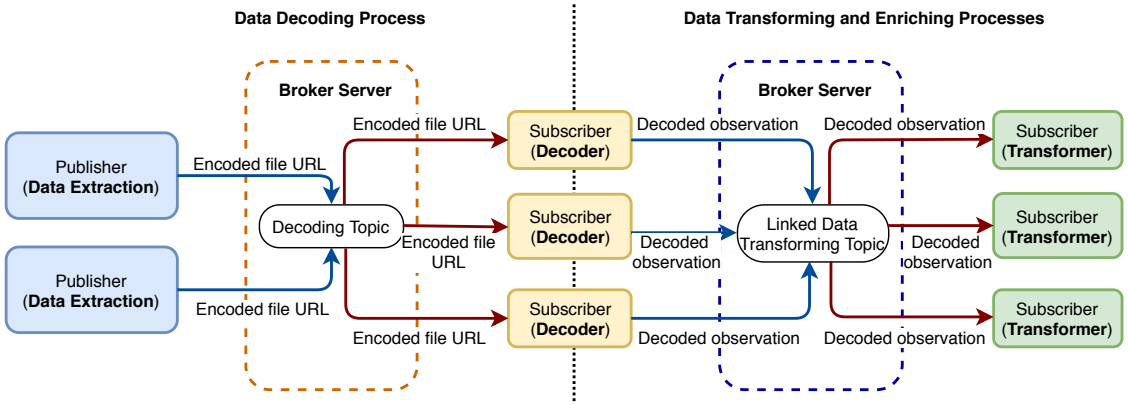


Figure 4.7: Publish-Subscribe architecture of ISH data decoding and transforming processes

Data transformation - This step is responsible to generate RDF data from the ISH decoded observation records produced in the previous step. Data are transformed to RDF upon our Linked Sensor Data model presented in Section 4.2. As shown at the right-hand side of Figure 4.7, we also apply the Publish-Subscribe mechanism for transforming the data. In this regard, a set of workers (transformers) is dedicated for receiving and transforming new ISH decoded observations. By using the Publish-Subscribe system, we address the potential bottleneck performance when transforming a large amount of observation data.

In addition to the RDF transformation of ISH observation data, the ISH station metadata, such as sensor description, observed properties, platform, location, etc, are also transformed via a separated single process. Because the station metadata are not frequently updated, their RDF transformation process is executed one time only. However, it can be updated if there is any change detected or informed (i.e, add a new station, station shutdown, etc). Along with the provided general sensor description, we also want to enrich it by adding more geographical information associated with the place where the station located. The enrichment process will be discussed in Section 4.4.

```
<got-res:Station/u4eu1epv0n_ish_1001499999> a  sosa:Platform;
        a got:WeatherStation;
        <geonames:nearbyFeatures> <http://sws.geonames.org/6954612/nearby.rdf> ;
        <geonames:parentADM1>   "Jan Mayen" ;
        <geonames:parentADM2>    <http://sws.geonames.org/7535695/> ;
        <geonames:parentCountry> "Svalbard and Jan Mayen" ;
        <geonames:parentCountry>  <http://sws.geonames.org/607072/> ;
        <wg284:lat> "70.933"^^xsd:double ;
        <wgs84:long> "-8.667"^^xsd:double .
```

Listing 4.4: Enriching spatial data with GeoNames

4.4 ENRICHING DATA WITH EXTERNAL DATASETS

In the Linked Data context, the data enrichment process is defined as linking the authoritative third-party data with an existing dataset. This aims to enhance the value of the dataset and make it more discoverable in the LOD cloud. For this purpose, we choose to enrich the spatial information of our meteorological RDF data by performing several interlinking processes to discover the connections between the ISH station location and their associated GeoNames[7] entities and features using the GeoNames APIs. Given a brief overview, GeoNames is a geographical database that contains over 25,000,000 geographical names corresponding to over 11,800,000 unique features[8]. For example, we want to discover the GeoNames connection of an ISH sensor station which locates at coordinate (70.933 -8.667). The discovery result is shown in Listing 4.4. As shown in this Listing, the output of this process is the set of associate GeoNames features that link to the station location, such as *geonames:parentCountry*, *geonames:parentADM1*, *geonames:parentADM2*.

4.5 PUBLISHING LINKED METEOROLOGICAL DATASET

In Section 4.4, we have described how to enrich the dataset by linking it with external data sources. There are more than 13,000 connections between the ISH sensor locations and GeoNames dataset that have been discovered. The dataset's main characteristics is summarized in Table 4.5.

In addition to the RDF data generation and enrichment, the final step is how to publish the dataset so that it can be accessible for end-user applications. In this regard, for the users who want to use the dataset for benchmarking purpose, we host the dataset in a dedicated file server. To simplify the download process for the user, the dataset is compressed and split by year, as described in Table 4.5.

Beside making the dataset accessible through a file server, we also want to store it in a RDF store so that it can be available to query via the store's SPARQL endpoint as a part of LOD cloud. However, selecting a proper RDF store for hosting this dataset is not an easy task due to the following reasons: (1) In order to preserve the spatio-temporal characteristic of the dataset, the

7 http://www.geonames.org
8 https://en.wikipedia.org/wiki/GeoNames

Table 4.4: Dataset Statistics

Category	Resources
Total Num. of Triples	~26 billions
Observation	3,706,719,471
Weather Station	26,628
Place	24,037
Time Series	Jan 2008 - Nov 2018
AirTemperature	1,023,960,866
Wind Direction	744,834,141
Wind Chill	864,179,251
Atmosphere Visibility	675,869,935
Atmosphere Pressure	397,875,278
Linked to GeoNames	13,359

Table 4.5: Dataset Statistics By Year

Year	File name	Num. Temperature	Num. Pressure	Num. Visibility	Num. Wind direction	Num. Wind speed	Total Observation	Num. Triples (billion)
2008	ish_2008_2_rdf.tar.gz	96,574,580	41,893,793	66,293,519	71,148,913	83,339,972	359,250,777	2.51B
2009	ish_2009_2_rdf.tar.gz	102,818,703	43,429,227	70,716,310	74,921,648	88,016,399	379,902,287	2.66B
2010	ish_2010_2_rdf.tar.gz	109,780,539	45,058,575	73,614,461	77,630,482	91,401,277	397,485,334	2.78B
2011	ish_2011_2_rdf.tar.gz	116,501,144	45,881,291	76,472,323	81,322,916	95,100,835	415,278,509	2.91B
2012	ish_2012_2_rdf.tar.gz	121,312,189	47,052,669	79,342,713	84,228,864	98,687,810	430,624,245	3.01B
2013	ish_2013_2_rdf.tar.gz	121,678,842	42,453,331	76,912,311	85,744,593	99,275,984	426,065,061	2.98B
2014	ish_2014_2_rdf.tar.gz	58,508,660	20,707,934	36,514,345	42,523,160	48,238,096	206,492,195	1.45B
2015	ish_2015_2_rdf.tar.gz	125,499,097	45,598,369	83,861,189	95,231,843	110,138,675	460,329,173	3.22B
2016	ish_2016_2_rdf.tar.gz	21,397,526	7,961,972	14,455,619	16,589,591	18,901,030	79,305,738	0.56B
2017	ish_2017_2_rdf.tar.gz	52,442,098	20,056,866	34,538,614	41,247,696	46,463,422	194,748,696	1.36B
2018	ish_2018_2_rdf.tar.gz	97,447,488	37,781,251	63,148,531	74,244,435	84,615,751	357,237,456	2.50B

selected RDF store has to support the spatial and temporal searches. This is a crucial requirement for processing sensor data. Although there are many triple stores that can host the dataset as the traditional RDF dataset without supporting spatial and temporal searches, however, this consequently reduces the main value and applicability of the dataset. (2) Due to the massive amount of sensor data, the selected RDF store should also have a capability to deal with the "big data" processing challenge. Specifically, the store should not only be able to store millions or years of sensor data but also can answer regular aggregate spatio-temporal queries on sensor data in a timely manner.

As already discussed in Section 3.4 and 3.5 in Chapter 3, although there has been a rapid increase in the number of RDF store that have been proposed recently, in conjunction with a number of performance assessment techniques specify to these stores, there is still a lack of study on analyzing the requirements and constraints to adopt a RDF store as a fundamental back-end solution for semantic sensor-based applications and services. This encourages us to conduct a first performance study of RDF stores for Linked Sensor Data. Details of our study will be presented in Chapter 5.

4.6 DATASET EXPLOITATION USE CASES

During the time of writing this thesis, the usages of the dataset have been studied and analyzed how it could be useful for different kinds of users within the LD community. This section exploits the real and potential use cases that can make use of the dataset.

```
SELECT ?month (avg(?tempValue) as ?TempAVG)
WHERE
{
    ?sensor         sosa:isHostedBy   $stationURI$.
    ?sensor         a           got:AirTemperatureSensor.
    ?temperature    sosa:madeBySensor     ?sensor.
    ?temperature    sosa:resultTime       ?time.
    ?temperature    sosa:hasSimpleResult  ?tempValue.
    FILTER(year(?time)=$year$).
}
GROUP BY (month(?time) as ?month)
```

Listing 4.5: Return the monthly average temperature in 2018 of a given station location

```
    ?temperature    sosa:resultTime       ?time.
    ?temperature    sosa:hasSimpleResult  ?tempValue.
    FILTER(?time >= xsd:dateTime($startTime$) &&
            ?time <= xsd:dateTime($endTime$)).
```

Listing 4.6: Time interval filter

4.6.1 Retrieving Analysis and Statistics from Sensors and Observations

Since the dataset is expressed in RDF with respect to the data model presented in Section 4.2, it is possible to retrieve analysis and statistics by running SPARQL and GeoSPARQL queries via provided SPARQL endpoint. For example, given the location of a weather station and year, it is possible to calculate the monthly average temperature based on the observations that the station generated. The corresponding query is illustrated in Listing 4.5. For that, the query first retrieves all the temperature observations produced by the sensor and then calculates the monthly average value. The <*stationURI*> is the placeholder which correspond to the given station URI, and <*year*> would be the year at which the user is interested to retrieve observations. Additional filters can be added, e.g. if the observations to retrieve from a weather station must be during a time interval, the final part of the query can be replaced with Listing 4.6.

Another example is shown in Listing 4.7. The query searches for the hottest temperature that has been observed for a given place within a time period. This query is more complicated in comparison with the previous example due to the required heavy aggregate computation on a massive amount of observation data.

4.6.2 Sensor Discovery And Data Mashup Application

We have developed a GraphOfThings[9] application that provides an easy-to-use map-based GUI in order to discover all kind of sensors at a given location [149]. The application is built on top of our linked meteorological dataset in a combination with our other datasets such as transportation, cameras, etc. The application provides a set of geographical functions allowing users to choose the area that they are interested in by providing a location name or street address. From the input location information, corresponding SPARQL query will be generated and executed.

9 http://graphofthings.org

```
SELECT ?maxValue ?weatherStation ?lat ?long
WHERE
{
  {
    SELECT (MAX(?value) as ?maxValue) ?sensor
    {
      ?sensor      a        got:TemperatureSensor.
      ?observation sosa:madeBySensor      ?sensor.
      ?observation sosa:hasSimpleResult   ?value.
      ?observation    sosa:resultTime       ?time.
      FILTER(?time >= xsd:dateTime($startTime$) &&
            ?time <= xsd:dateTime($endTime$)).
    }
  }
    ?sensor         sosa:isHostedBy   ?weatherStation.
    ?sensor         a           got:AirTemperatureSensor.
    ?weatherStation geo:hasGeometry ?geoFeature.
    ?geoFeature       geo:lat ?lat.
    ?geoFeature       geo:long ?long.
}
```

Listing 4.7: Search for the hottest place during a given time period

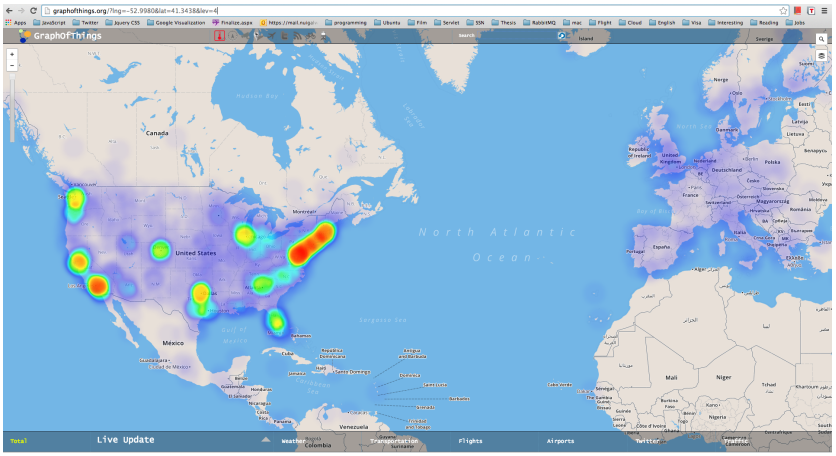


Figure 4.8: Screen shot of GraphOfThings application

Finally, all sensors located in the result area will be displayed on a map. Through the map UI, users can also view the historical data of each sensor as well as the live data which are being streamed from our sensor data management system [154]. A screen-shot of the application is shown in Figure 4.8.

4.7 SUMMARY

The chapter presents the process of generating and publishing our linked meteorological sensor dataset which is originally made publicly available by NOAA's NCDC. We have reused ontologies like SOSA/SSN, GeoSPARQL and OWL-Time for semantically describing sensor data. Furthermore, by employing the LD publishing principles presented in Chapter 2, our linked me-

teorological dataset is made available to query via a SPARQL endpoint as a part of LOD Cloud. The dataset consists of nearly 26 billion RDF triples generated by transforming 3.7 billion meteorological observation records during the time from 2008 to 2018.

In addition to the large scale of the dataset, another feature that makes the dataset more valuable is its meaningful spatio-temporal information. We have demonstrated the potential usages of our dataset via a set of example analytical spatio-temporal queries and via our real-world sensor application built on top of the dataset. In addition to that, through a public endpoint and a dedicated hosted file server, other systems can consume our data and offer visualizations, aggregations over it or use it for benchmarking evaluation purposes. In this thesis, this dataset has proved its vital role by being used in different types of experiments.

It is worth mentioning that the publishing process of our dataset has also raised a challenge for selecting an appropriate RDF store for hosting the dataset. Unlike the traditional RDF stores, the selected one should support spatio-temporal queries and has a capability for dealing with ”big data” challenge for processing sensor data. While there has been no such studies existed, this motivates us to conduct a first performance study of RDF stores for Linked Sensor Data, that will be presented in the next chapter.

5 A PERFORMANCE STUDY OF RDF STORES FOR LINKED SENSOR DATA

The rapid growth of the Linked Sensor Data leads to the urgent requirement for assessing the readiness of current RDF stores in terms of data loading and query performance. Current RDF benchmarks and experiments mostly focus on evaluating the response time and query throughput of individual stores on semantic data to show the general weaknesses and strengths of RDF implementations, but have not yet evaluated their performances over spatial and temporal data. Moreover, in their experimental conclusions, there is also a lack of sufficient insights that will help RDF store developments. In this chapter, following the challenge raised in the previous chapter, the requirements and constraints to adopt a RDF store as a fundamental back-end solution for semantic sensor-based applications and services are analyzed. Following that, we will briefly introduce some selected qualified RDF stores. An extensive performance comparison of these stores is also presented. Unlike the traditional approach, our study also focuses on evaluating the system performance on spatial and temporal sensor data. In the conclusion, we will highlight some findings and existing issues that need to be addressed and thus, to clarify the contribution of our approach that will be described in later chapters.

This work appears in [156].

5.1 FUNDAMENTAL REQUIREMENTS OF A PROCESSING ENGINE FOR LINKED SENSOR DATA

Sensor data has distinctive characteristics that make traditional RDF engines an obsolete solution. This is due to the limited capability of such engines to process the massive amount of Linked Sensor Data available, as well as the lack of spatio-temporal index support. In general, when providing an RDF processing engine for Linked Sensor Data, there are some important features that should be provided. These features are as follows:

- **Geospatial search**: This mandatory feature plays an important role for processing sensor data. Having spatial search support will help to not only provide the spatial information of a spatial object, but will also help to compute the spatial relationships between the two geometries of objects (i.e, within, intersecting, etc). For example, this can be used for finding all the weather stations that are operating within a given area.
- **Temporal filter**: Most of the queries from end-user or sensor-based applications require filtering on the temporal aspect of sensor observation data. This task is very expensive

due to the high-frequency updates of sensor observation data. Furthermore, this is also the main reason behind a dramatic increase in the required storage size. Therefore, several enhancements should be made to the temporal filter by the query processing engine so that observation data can be easily retrieved.

- **Full-text search**: This is an advanced feature that aims to improve the performance of full-text search queries. Essentially, these kinds of searches are mostly based on finding given keywords embedded in literal values such as descriptions, street names, location names, etc.
- **Scalability**: This is also another advanced feature that allows the engine to deal with the "big data" processing challenge of sensor data. Also, this can help the engine to address a performance issue when executing a high volume of user queries. Thus, either a scalable solution or an efficient index mechanism is needed [20]. One possible solution that can be considered is one that will duplicate the same service so as to allow many concurrent queries while also providing suitable fault tolerance capabilities. Alternatively, another solution is to split the data across different servers and to provide a middleware component that can coordinate data transactions amongst these underlying servers.
- **Spatio-temporal query language**: It is important to note that the standard SPARQL query language does not support spatio-temporal queries (nor full-text queries). For this reason, to enable this feature, the RDF database creators must provide spatio-temporal querying by either defining a new language in their own specific syntax or by extending the SPARQL query language.

5.2 ARCHITECTURAL DESIGN

The general architectural design of RDF stores that support spatio-temporal query processing over Linked Sensor Data can be classified into two categories, namely *native* architectures and *hybrid* architectures. In this section, we will discuss the details for each architecture and analyze the features that might affect the query performance of each.

5.2.1 Native Architectural Design

The native architectural design of RDF stores for Linked Sensor Data is illustrated in Figure 5.1. This architecture needs to implement the data indexing system, the query engine, and physical storage.

5.2.1.1 *Data Index*

The data index has a vital role and can significantly affect the performance of data insertion. This is one of the primary design elements for storage systems. Additionally, an efficient data indexing system also helps to improve query performance. In the Linked Sensor Data context,

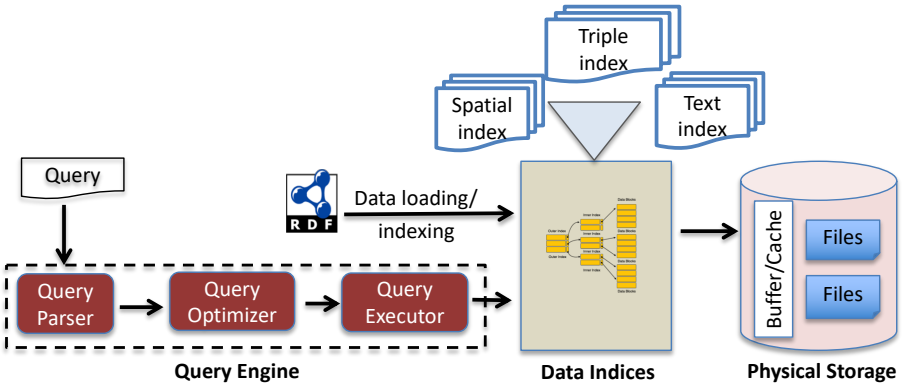


Figure 5.1: The native architectural design

along with the triple index, defining spatio-temporal data indices is an essential requirement for engines that follow native architectural design principles. This multidimensional index feature not only helps with triple-based retrieval requirements but also with supporting spatio-temporal queries. For example, Virtuoso uses RTrees for spatial indexing and bitmap indices for RDF triples. Meanwhile, Jena uses a Lucene spatial index along with B/B+ and three advanced triple indices on *spo*, *pos* and *osp* to accommodate different triple patterns.

Another important feature of the data index component is to define the triple patterns that are used to extract the spatial, temporal or text values from the input data. The extracted values will be indexed properly based on their characteristics and implied context. For example, spatial data are indexed by an R/R+ trees algorithm while the text literals are analyzed by a string index algorithm.

5.2.1.2 *Query Engine*

The query engine is used for data retrieval which is generally comprised of three sub-modules, namely a query parser, query optimizer, and query executor.

QUERY PARSER The query parser is responsible for parsing the input query from a user or an application program to an algebraic expression. Unlike the standard SPARQL query parser, the one for spatio-temporal queries over Linked Sensor Data has to adapt to its associated spatio-temporal query language syntax. As SPARQL does not cover spatio-temporal queries, so the query language that supports spatio-temporal or full-text search can be defined either by the RDF engines in their own syntax or by extending the SPARQL language. For example, Virtuoso and Jena define their own spatial query language by adding spatial built-in functions to SPARQL. These additional functions are assigned along with dedicated prefixes, such as <*bif:*> and <*spatial:*>, for Virtuoso and Jena, respectively. In the meantime, instead of defining a new query syntax, Strabon [108] and Stardog[1] adopt the WGS84 and OGC's GeoSPARQL vocabularies, which have been widely used and have become the standard W3C recommendation.

1 https://www.stardog.com/

QUERY OPTIMIZER This module is used to determine the most efficient way to execute a given query by considering all the possible query execution plans. The proper execution plan will be represented as an execution-order tree that consists of algebraic operators. A good execution plan will help to prune all the redundant intermediate results, and thus will improve the query performance by reducing the memory consumption as well as result materialization.

Obviously, each query plan leads to different query performance. Finding the proper execution plan that can balance the resource-consuming cost and the query response time is always an ultimate goal of the query optimizer. In the Linked Sensor Data context, this process becomes more complex and expensive due to the difficulties in building a comprehensive cost model that can reflect all the spatial and temporal aspects of Linked Sensor Data. In addition, the lack of statistical information on spatio-temporal data and access patterns also makes the join order optimization challenging and resource consuming. Currently, most RDF query optimizers can only collect limited statistics, such as Jena and Strabon, which collects the number of times a predicate appears. Furthermore, other systems (i.e., Virtuoso and Stardog) cache query plans for later use.

QUERY EXECUTOR The query executor is responsible for taking the query execution plan handed back by the query optimizer, recursively processing it by accessing the physical storage, and returning the query results.

5.2.1.3 *Physical Storage*

The final component is the physical storage which includes the database files, a buffer or its own file cache manager that manages the buffering of data to reduce the number of disk accesses. The physical storage can be classified into two types, namely *native* and *non-native* storage [43].

NATIVE STORAGE The *native storage* treats RDF triples as first-class citizens and stores them in a way that is close to the RDF data model. There are two approaches to implement a *native storage*: persistent disk-based and main memory-based. The *persistent disk-based* systems store the RDF data permanently in files [78, 120, 131]. The advantage of these implementations is that the data is safe during a system restart. However, it should be considered that the data writing and reading operations can be slow due to a performance bottleneck.

In contrast to the *disk-based* approaches, the *main memory-based* ones keep data in the system memory. All the data reading and writing operations occur there, thereby improving the overall system performance. However, due to the limited size of the memory, this solution requires a memory-efficient data representation as well as composite index-based techniques so that there will be enough space to not only store the data but also for the other operations associated with query processing and data management [52]. Some implementations falling into this category are Jena, Hexastore[196], Bitmap[14], etc.

NON-NATIVE STORAGE The *non-native storage* relies on the relational database system to store RDF data permanently [38, 77, 76, 199, 50]. In this type of storage, data is either stored in a single table (schema-free approach) or in a set of relational tables (schema-based approach) such

as a subject table, predicate table, object table, etc. The advantage of the *non-native storage* is the less demanding efforts in terms of design and implementation. Nevertheless, in order to achieve good system performance, an efficient mapping solution of SPARQL-to-SQL and graph-to-relational schema have to be taken into consideration [43].

5.2.2 Hybrid Architectural Design

The hybrid architecture uses existing systems as sub-components for the processing needed. The common architecture of a hybrid solution is illustrated in Figure 5.2. In this architecture, the chosen sub-components are accessed with different query languages and different input data formats. Hence, the hybrid approach needs a Query Rewriter, a Query Delegator and a Data Transformer. The Query Rewriter rewrites a SPARQL-like query to sub-queries that the underlying systems can understand. The Query Delegator is used to delegate the execution process by externalizing the processing to sub-systems with the rewritten sub-queries. In some cases, the Query Delegator also includes some components for correlating and aggregating partial results returned from the hybrid databases if they support this. The Data Transformer is responsible for converting input data to the compatible formats of the access methods used in the hybrid databases. It also has to transform the query results from these systems to the format that the Query Delegator requires.

By delegating the processing to available systems, building a system following the hybrid architecture takes less effort than using the native approach. However, the disadvantages of this approach include the limited control over the sub-components as well as the challenges involved to efficiently coordinate them.

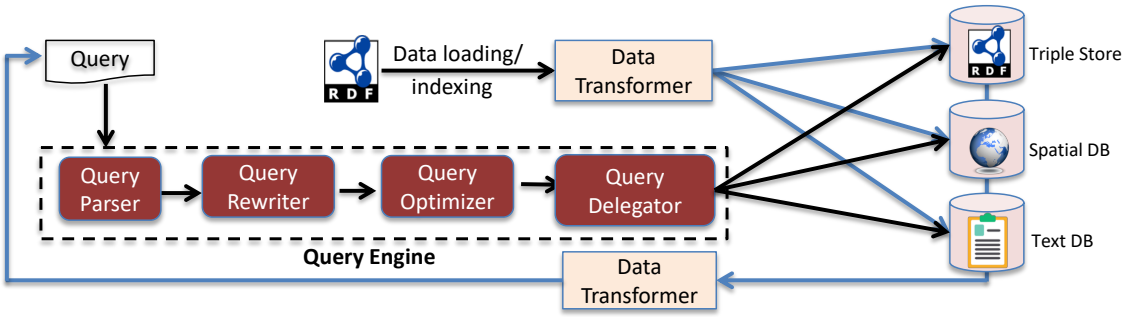


Figure 5.2: The hybrid architectural design

5.3 ANALYZING EXISTING RDF STORES FOR LINKED SENSOR DATA

In this section, we will briefly analyze the most popular and mature existing RDF stores that qualify for the required features which have been identified and discussed in Section 5.1. Table 5.1 summarizes the features supported by these selected stores. Please be aware that the selection

process has been carried out with the target of identifying suitable RDF stores that can be used for efficiently processing Linked Sensor Data with respect to the provided features. A detailed systematic-level evaluation of these stores will be presented later in Section 5.5.

5.3.1 Virtuoso Open Source

Virtuoso Open Source (v7.2.4) [50] is an example of the native architecture type and is also a widely-used RDF store in the Semantic Web community. It is the storage system that is hosting the DBPedia database [15]. Virtuoso is a general-purpose RDBMS with extensive RDF adaptations that are comprised of RDF-oriented data types and a SPARQL-to-SQL front-end compiler. In Virtuoso, RDF data can be stored as RDF quads which are indexed properly in SQL tables. The engine supports the SPARQL 1.1 language. A SPARQL query will be translated properly to SQL via the SPARQL-to-SQL compiler. Virtuoso's user community is quite active and the software is updated regularly.

The current version of Virtuoso Open Source does not support clustering features. However, it supports advanced spatial indexing (using RTrees) and full-text search. The engine provides its own spatial and text search query language via SPARQL built-in functions. Virtuoso does not have a dedicated temporal index. Instead, the value with a timestamp is indexed as an RDF literal value. Therefore, a temporary query is processed via the provided built-in SPARQL date-time functions. For RDF data, Virtuoso's index scheme consists of five indices (*psog*, *pogs*, *sp*, *op*, *gs*) that have two full indices over RDF quads plus three partial indices.

Virtuoso serves SPARQL queries via a pre-assigned public SPARQL endpoint. The query modules will then translate an input SPARQL query to the corresponding SQL query referring to the five triple store tables mentioned above. Virtuoso is backed by a RDBMS and can adapt to all SQL optimization techniques. For example, it provides a cost-based SQL optimizer which performs several types of query transformation, such as join order, index selection, selection of join algorithms, etc. The Virtuoso cost model is based on table row counts, defined indices and uniqueness constraints, and column cardinalities, i.e. count of distinct values in columns. Additionally, histograms can be made for the value distribution of individual columns.

5.3.2 Stardog

Stardog[2] is a knowledge graph platform that supports RDF storage. Stardog follows the native architecture design solution. Originally, it was implemented by the developers of a well-known OWL reasoner (Pellet) [179]. At the time of writing, the latest version of Stardog is v6.0.1 which supports full-text search, spatial and basic temporal filters. As a graph database, Stardog supports ACID transactions and SPARQL 1.1 [43]. Unlike Virtuoso, this engine does not define its own spatio-temporal query language. Instead, it adopts the WGS84 and OGC's GeoSPARQL vocabularies [144].

2 http://stardog.com

There are two indexing modes supported in Stardog, one based only on triples and another one for quads. Indexing data are stored on-disk for both cases. However, "in-memory" mode is also available. No other details in terms of index mechanisms are provided in any publications.

Stardog follows the bottom-up approach to evaluate the query execution plans generated from a given SPARQL query [103]. In the query plan tree, leaf nodes without input are evaluated first, and their results are then sent to their parent nodes up to the plan. Typical examples of leaf nodes include scans, i.e. evaluations of triple patterns, evaluations of full-text search predicates, and VALUES operators. Parent nodes, such as joins, unions, or filters, take the results produced by the leaf nodes as inputs, process them and send their results further towards the root of the tree. The root node of the plan tree, which is typically one of the solution modifiers[3], produces the final results of the query which are then encoded and sent to the client.

Another important feature of Stardog is clustering support which allows horizontal scaling. However, this feature is only available in the commercial version. According to a recent platform report, Stardog can process 10 billion triples stored on a single server.

5.3.3 Apache Jena

Apache Jena[4] is an open-source SPARQL 1.1 framework for Java. It provides an API to extract data from and write to RDF graphs. Jena implements the native architecture design which includes the optional quads RDF storage layers, namely TDB (file system), SDB (SQL DBMS), and in memory. Jena also provides inference support (supporting RDFS, OWL-Lite or using custom rules), but it works only on triple stores and not on quadruples stores. From version 3, Jena supports full-text search and basic spatial index through the use of Lucene or Solr. No clustering solution has been mentioned.

Regarding the indexing architecture, the one in Jena is built around three concepts, namely a node table, triple/quad indices, and a prefixes table. Among them, the node table is responsible for storing the dictionary which provides two mappings for the RDF terms, namely *string-to-id* and *id-to-string*. The default storage of the former is implemented using B+ trees, and the latter is based on a sequential access file. Triples and quads are stored in specialized structures. Triples are held as three identifiers in the node table while quads are assigned as four. Again, B+ trees are used to persist these indices.

Query execution in Jena involves both static and dynamic optimizations [190]. Static optimizations aim to transform the algebras of the execution tree into new, equivalent and optimized algebra forms. This transformation process is performed in advance of the query execution. Meanwhile, dynamic optimizations involve deciding on the best execution approach during the execution phase and can take into account the actual data so far retrieved.

A number of optimization strategies are provided: a statistics-based strategy, a fixed strategy and a strategy of no reordering. For the statistics-based strategy, the Jena optimizer uses information captured in a per-database statistics file to specify the join order of the query triple patterns. For that, the statistics file takes the form of a number of rules for approximate matching counts

3 https://www.w3.org/TR/sparql11-query/#solutionModifiers
4 https://jena.apache.org/

for triple patterns. The file can be automatically generated when the engine starts. Users can update it manually by adding and modifying rules to tune the database based on higher-level knowledge, such as inverse function properties.

5.3.4 RDF4J

RDF4J[5] (formerly known as Sesame [30]) is an open-source Java framework for processing RDF data. It is a native RDF processing engine which includes parsing, storing, inferencing and querying over RDF. Similar to Apache Jena, RDF4J also supports two out-of-the-box RDF databases (in-memory and native store) along with third-party storage solutions. It also fully supports SPARQL 1.1, full-text search and spatial index in conjunction with the GeoSPARQL language. RDF4J has no clustering feature.

The key component of RDF4J is a "Storage And Inference Layer" (SAIL). In short, SAIL is an application programming interface that offers RDF-specific methods to its client and translates these methods to calls to its specific underlying DBMS. The benefit of the introduction of SAIL is the flexible implementation of RDF4J on top of a wide variety of repositories without changing other RDF4J's components. Consequently, in addition to its own *in-memory* and *disk-based* repositories, RDF4J can be easily attached to other DBMS such as MySQL, PostgreSQL, etc. The repository used in this work is the default native *disk-based* repository, which originally provided by RDF4J. Regarding the query optimization used in RDF4J, no detailed information is publicly provided.

5.3.5 Strabon

Strabon[108] is an open-source semantic DBMS that can be used to store linked geospatial data expressed in stRDF format and query it using the stSPARQL language [106]. It follows the hybrid architecture design and is backed by a PostGIS extension of Postgres RDBMS plus RDF4J, allowing it to manage both semantic and spatial RDF data. Strabon supports SPARQL 1.1. It provides spatial, temporal search and basic text search. In contrast to other engines, Strabon provides an advanced temporal search which allows users to query Allen's temporal relationship between two temporal objects. In terms of indexing capability, the developers of Strabon reported that it can scale up to 500 million triples. No clustering solution is available for Strabon.

When data is imported to Strabon, the data is firstly decomposed into stRDF triples and each triple is processed separately. Data is stored using dictionary encoding that follows a "per-predicate" scheme. The "per-predicate" scheme uses vertical partitioning to store triples in different tables based on their predicate, and one table per predicate is maintained. The B-Tree algorithm is then applied on these predicate tables. For spatial literals found during the data loading, Strabon stores them in a dedicated *geo_values* table. A spatial index is then created by applying an R-tree-over-GiST algorithm [83].

5 http://rdf4j.org

Query processing in Strabon is implemented by modifying the RDF4J components. The query engine consists of a parser, an optimizer, an evaluator and a transaction manager. Moreover, besides the standard formats offered by RDF4J, Strabon offers the KML and GeoJSON encodings, which are widely used for representing the spatial data. Strabon applies exactly the same as RDF4J's query optimization techniques for the standard SPARQL part of a stSPARQL query. For the spatial extension functions, an additional optimization step is introduced. In this step, the cost of the spatial functions presented in the query will be evaluated by PostGIS. Based on the cost estimated, the query tree is then optimized and executed.

Table 5.1: RDF stores comparison

	Technical characteristics						
	Query language	**Spatial filter**	**Full-text search**	**Temporal filter**	**License**	**Clustering**	**Latest release Date**
Virtuoso 7	SPARQL, SQL	Y	Y	Y (basic)	Cm,OS	N	Oct 2018
Stardog 6	SPARQL, SQL	Y	Y	Y (basic)	Cm,OS	Y	Jan 2019
Apache Jena	SPARQL	Y	Y	Y (basic)	OS	N	March 2019
RDF4J	SPARQL	Y	Y	Y (basic)	OS	N	April 2019
Strabon	SPARQL	Y	N	Y (advanced)	OS	N	Jan 2018

5.4 EVALUATION METHODOLOGY AND METRICS

The previous sections give insight to the architecture and available features of selected RDF stores that can be applied for Linked Sensor Data. This section will describe in detail the methodology and the metrics used to evaluate the performance of these systems. We have triggered the performance of Jena, RDF4J, and Strabon by modifying their source code. Virtuoso and Stardog already provide some metrics themselves, which we retrieve by using the Virtuoso JDBC/CLI and Stardog APIs, respectively.

Our evaluation methodology is carried out within two phases. In the first phase, we will evaluate the data loading performance of the RDF stores over the Linked Sensor Dataset. In this experiment, we evaluate separately the loading performance of semantic, spatial and text data.

The second phase is to evaluate the query execution performance. We firstly define a benchmark queries set which is performed over our linked meteorological dataset. These queries and dataset are described later in Section 5.5. After having these benchmark queries defined, we evaluate the query performance by executing these queries. In addition to the overall query execution time, we also measure the execution performance of query parsing, query optimization, and query execution. The query parsing time is calculated from the time the system retrieves the query string to the time the query algebra tree is generated. Similarly, the query optimization process is considered starting from the time the query tree and it is finished when an execution plan is delivered. For simplicity, any other run-time decisions are considered as part of the query execution rather than part of the query optimization. The query execution is finished when the last result has been received.

5.5 EXPERIMENTAL SETTINGS

5.5.1 Benchmark Dataset

Our experiments are conducted over the linked meteorological dataset previously described in Chapter 4. The meteorological dataset consists of 10 years of meteorological data for over the world. The dataset is categorized into two parts, namely static and dynamic dataset. The static one describes the station and sensor descriptions, such as spatial information, text data, etc, and is not frequently updated. The dynamic dataset contains the observation data generated by the sensor station described in the static dataset.

Due to the large size of the meteorological dataset and also because of our limited infrastructure resources, only a subset of this dataset is used in our experiments. As described in Table 5.2, this subset contains the static dataset and the observation data from the year 2016 to 2018, results in a set of 2.5 billion triples.

The spatial and text data loading performance is evaluated over the static dataset. To provide a more comprehensive evaluation, we enlarge this dataset by generating more spatial and text objects on the basis of the SOSA/SSN data model [94, 40], the same data model used for our linked meteorological data. The static dataset is described in Table 5.3.

Table 5.2: Benchmark Linked Meteorological Dataset

Year	Num. Observation	Num. Triples (billion)	Size (Gb)
2016	79,305,738	0.555	112
2017	194,748,696	1.363	274
2018	357,237,456	2.5	503
Sum	**631,291,890**	**4.419**	**889**

Table 5.3: Static datasets description

Dataset	Static dataset
Num. spatial/text object (million)	20
Num. triples (million)	120
Raw Data size	20G

The dynamic dataset presenting the observation data is used to evaluate the query performance in the second phase of our experiment. The observation data are extracted within the time period from Jan 2016 to May 2016 (250M triples), as described in Table 5.4, which we believe could indicate a general performance measure for this test.

Table 5.4: Datasets description for query performance evaluation (from 01/2016 to 05/2016)

Dataset	Num. observation (million)	Num. Triples (million)	Size
2 months	17.5	122.4	2.8G
3 months	22.2	155.6	4.3G
4 months	25.8	180.9	6.5G
5months	30.5	213.3	7.1G
Static	0	0.75	135Mb

5.5.2 Benchmark Queries

We have selected a set of 11 queries from the query logs of our real-world sensor application GoT [110] based on their characteristics (frequency, query operators, temporal range, observed properties, etc.). In general, these benchmark queries aim to test the engine processing capability with respect to their provided features for querying Linked Sensor Data. It is also important to mention that the selected queries are historical queries that perform over our linked meteorological dataset. Moreover, as previously mentioned, because the standard SPARQL 1.1 language does not support spatio-temporal queries nor the full-text queries, some RDF stores have to extend the SPARQL language with their own specific syntax. Therefore, some of these queries need to be rewritten so that they are compatible with the engine under test.

We summarized some highlight features of the benchmark queries as follows: (i) if the query has input parameter; (ii) if it requires geospatial search; (iii) if it uses temporal filter; (iv) if it uses full-text search on string literal; (v) if it has Group By feature; (vi) if the results need to be ordered via ORDER By operator; (vii) if the results are using the LIMIT operator; (viii) the number of variables in the query; and (ix) the number of triples patterns in the query. The group-by, order-by, and limit operators imply the effectiveness of the query optimization techniques used by the engine (e.g., parallel unions, ordering or grouping using indices, etc.), and the number variables and triples patterns give a measure of query complexity. The summary of highlight features of these queries is presented Table 5.5 and their SPARQL representations can be found in Appendix A.

Table 5.5: Benchmark queries characteristics on linked meteorological data

Query	Parametric	Spatial filter	Temporal filter	Text search	Group By	Order By	LIMIT	Num. variables	Num. triple patterns
1	✓	✓						3	3
2	✓							3	4
3	✓		✓					7	8
4	✓	✓	✓		✓	✓	✓	7	8
5	✓		✓		✓	✓		4	5
6	✓		✓					5	8
7	✓	✓	✓			✓	✓	7	8
8				✓		✓		3	4
9				✓	✓			6	7
10	✓		✓	✓	✓			7	9
11					✓			2	1

5.5.3 Platform

We conducted our experiments on a physical server which has following configuration: 2x E5-2609 V2 Intel Quad-Core Xeon 2.5GHz 10MB Cache, Hard Drive 3x 2TB Enterprise Class SAS2 6Gb/s 7200RPM - 3.5" on RAID 0, Memory 128GB 1600MHz DDR3 ECC Reg w/Parity DIMM Dual Rank.

5.5.4 Setup

We have experimented on Jena v3.9.0, RDF4J v2.6.5, Stardog v6.0.1, and Virtuoso Open Source v7.2.5. For Virtuoso, the system parameters NumberOfBuffers and MaxDirtyBuffers were set to 40GB. Java heap size for Jena and RDF4J is also set to 40GB. Besides, for Jena, we have enabled

its optimization strategy option to statistic strategy. Regarding RDF4J configuration, we have configured the index mechanism to *spoc*, *posc* and *opsc*. For Stardog, as we will not evaluate the inference capability, we disable this option when importing data. The spatial and textual index options were enabled. Similar to other engines, Stardog memory size was also set to 40 GB. The rest of the parameters were left to the default values.

5.6 RESULTS AND DISCUSSION

In this section, we present and discuss the experimental results following the evaluation methodology and metrics described in Section 5.4.

5.6.1 Data Loading Throughput

5.6.1.1 *Triple loading performance*

The triple loading performance of the test stores is depicted in Figure 5.3. We can see how the performance is affected when the size of the RDF dataset increases. According to the results, Virtuoso is the fastest, followed by Stardog and Jena with about 2 times and 6 times slower than Virtuoso, respectively. RDF4J is the slowest being about 10 times slower than Virtuoso. According to our observation, there are two possible explanations for the poor loading performance of RDF4J: (1) The high frequent index updates due to the relatively small page size used by RDF4J; (2) The HTTP communication latency between the RDF4J client and server components. Note that, RDF4J does not support bulk data import. Instead, it provides a set of REST API functions for the client to access the RDF database either for reading, writing or querying data. This consequently leads to the dramatical increase in the network communication latency time when the number of request access to the database goes up. This reason is also explained for the poor data loading performance of Strabon.

Regarding the required repository size, it can be observed in Table 5.6 that this metric has a linear increase with respect to the dataset size. Unfortunately, due to the complex hybrid architecture of Strabon, its repository size metric can not be measured. For Virtuoso, its index compression methods pay off, resulting in the smallest index size. Stardog uses 80% more space than Virtuoso. Jena and RDF4J generate much larger indices about 4 times larger than those of Virtuoso, though they have fewer indices.

Table 5.6: Required repository capacity (in GB) with respect to dataset size

RDF Engine/Data size	500 (mil)	750 (mil)	1000 (mil)	1250 (mil)	1500 (mil)	1750 (mil)	2000 (mil)
RDF4j	60	79	99	121	142	NA	NA
Virtuoso 7	14	19	23	32	39	47	56
Stardog	24	43	64	82	102	125	146
Apache Jena	62	89	117	135	164	NA	NA
Strabon	NA	NA	NA	NA	NA	NA	NA

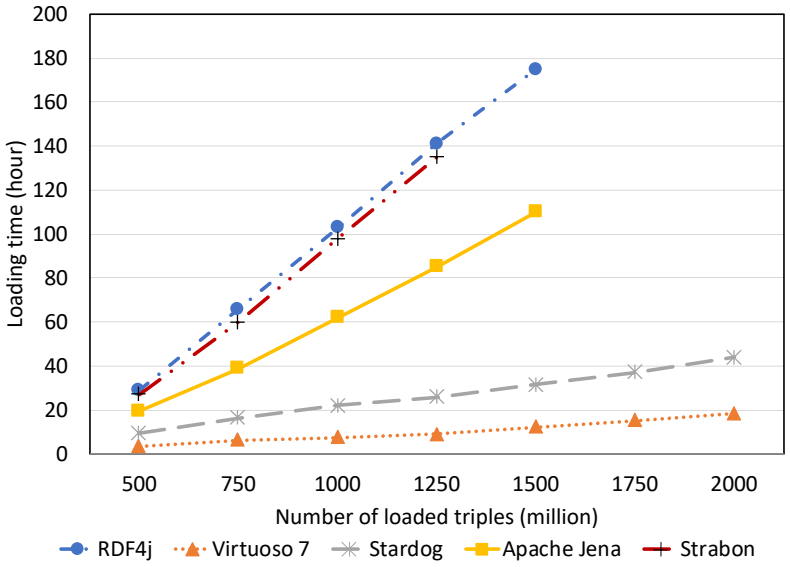


Figure 5.3: RDF stores performance of RDF data loading

5.6.1.2 *Spatial and text data loading performance*

Unlike the existing performance studies that only focus on general data loading performance, our evaluation also measures the loading performance on spatial and text data. In this regard, we measure the loading speed via the number of objects can be indexed per second, instead of the number of triples. An object consists of spatial and text description. This evaluation helps us to have a better understanding of the indexing behavior of the test engines for specific types of data such as geospatial and text.

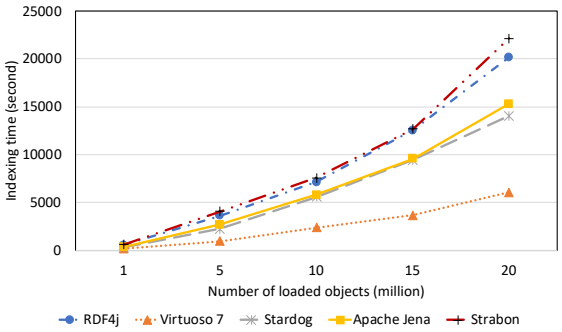


Figure 5.4: Spatial/text data loading time

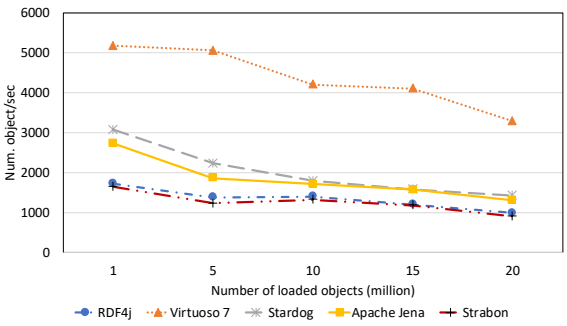


Figure 5.5: Average spatial/text data loading speed

The spatial and text data loading time with respect to dataset size is shown in Figure 5.4. Figure 5.5 depicts the average loading speed. As described, RDF4J and Strabon perform poorly - their average loading speed is less than 1000 obj/sec when loading 20 million objects. Apache Jena has a better performance and its loading speed can reach to 1306 obj/sec. It loads 20 (mil) dataset in about 5.6 hours. Stardog performs better than Jena, results in 4 hours for loading the same dataset and the average speed is almost 14k obj/sec. In comparison, Virtuoso achieves much faster loading speed than others, which can reach to 3287 obj/sec. However, a drawback of the impressive loading speed is the increase of number triples stored in Virtuoso with respect to the original data provided. This is because the additional *geo:geometry* spatial triples which are

generated during the transformation of the *geo:lat* and *geo:long* triples to enable the geo-spatial indexing.

5.6.2 Query Performance

The set of figures 5.6 to 5.16 presents the average query response time concerning Jena, Virtuoso, RDF4J, Stardog and Strabon with respect to the different time horizons of two, three, four and five months observation data of the year 2016, respectively. The datasets used to evaluate the query performance are described in Table 5.4. The benchmark queries have been executed by performing a pseudo-random sequence of these queries repeated ten times in order to minimize the caching effect. Besides, we also empty the system cache before running the query of each test execution.

Looking at the evaluation results, unsurprisingly, the query performance of all systems decreases with the growth of dataset. However, in the cases of no spatio-temporal nor full-text searches are involved (Q2, Q11), the query performances remain stable and are weakly affected by the increase of dataset size. This is understandable because these queries are only performed over the static dataset, which is unchanged and not influenced by the dynamic data. Virtuoso and Stardog perform closely and require less execution time, in the order of *ms*. For example, in the case of 5 months dataset, Virtuoso takes 83 (ms) to execute the Q2 while it is 137 (ms) in Stardog. Regarding the others, the performances of RDF4j and Strabon are comparable, followed by Jena.

For the query that has only the spatial filter and basic triple matching such as Q1, Virtuoso executes this query in about 80 (ms) for 5 months dataset. Following this is Stardog with 112 (ms). Jena and RDF4J appear to have the worst performance with average response times are 50128 (ms) and 1761 (ms), respectively. We attribute this to the effectiveness of the query optimizer in each system. Wrong execution plan might lead to a catastrophic performance. We will discuss this further in the following subsection that is about the cost breakdown of query performance.

Another aspect to be considered is the mixing of spatial filter, time filter and full-text search queries (Q3, Q4, Q5, Q9, Q10). According to the evaluation results, Virtuoso is still best ranked with less than 1(s) of execution time for 5 months dataset, followed by Stardog. With Jena, RDF4J and Strabon, we obtained an extremely high query execution time, greater than 50(s). Analyzing the cost breakdown of these queries, we derive two possible explanations: (1) The growth of the observation dataset size, which certainly requires more data processing operations (lookup, read, etc) and resource-consuming, results in the considerable increase of the query execution time. (2) The inefficient query execution strategies that have been applied. Inappropriate execution plan also consequently effects to the query performance.

5.6.3 Cost Breakdown

Table 5.7 shows the cost breakdown between query parsing, query optimization, and execution, across all stores and queries for 5 months dataset. It is easy to realize that, except Virtuoso, for most stores, the query run-time cost is mostly dominated by the execution time. For example,

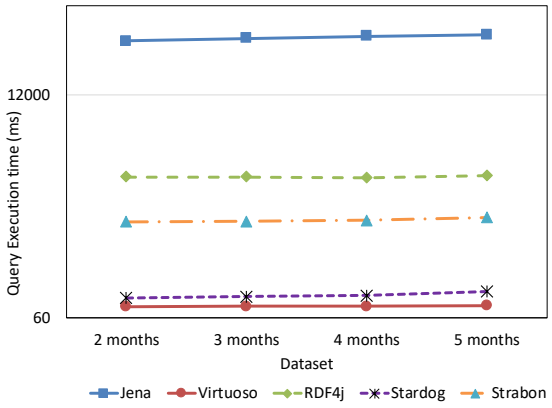


Figure 5.6: Q1 execution time

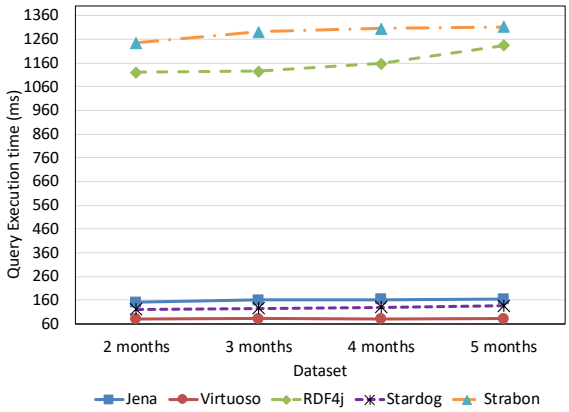


Figure 5.7: Q2 execution time

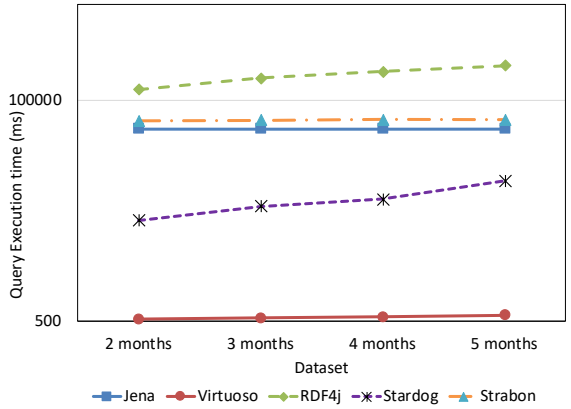


Figure 5.8: Q3 execution time

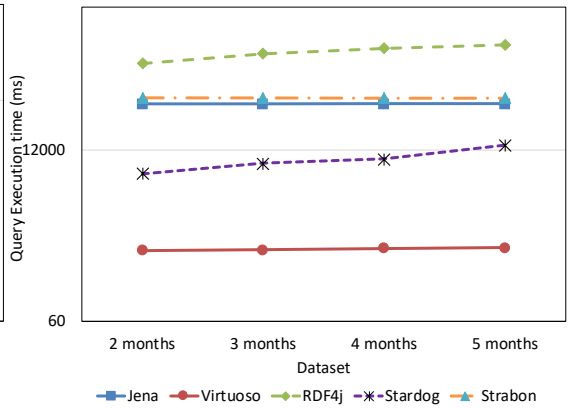


Figure 5.9: Q4 execution time

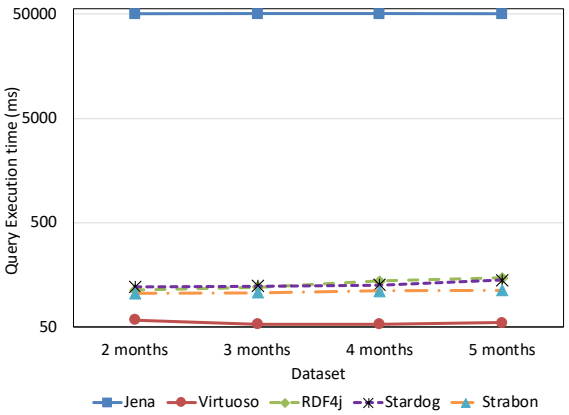


Figure 5.10: Q5 execution time

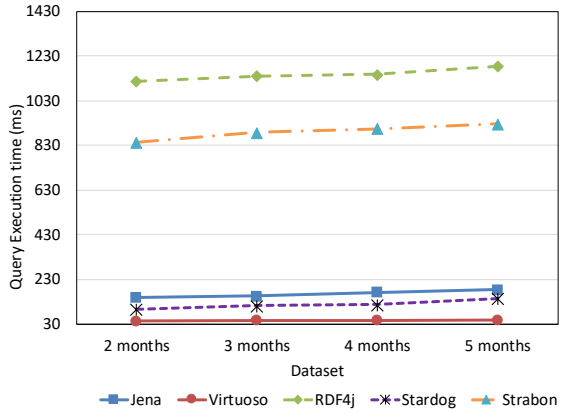


Figure 5.11: Q6 execution time

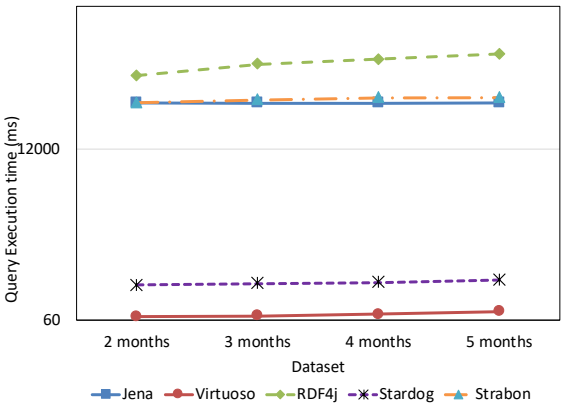


Figure 5.12: Q7 execution time

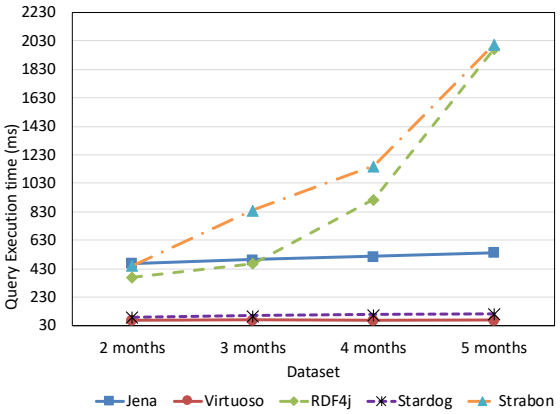


Figure 5.13: Q8 execution time

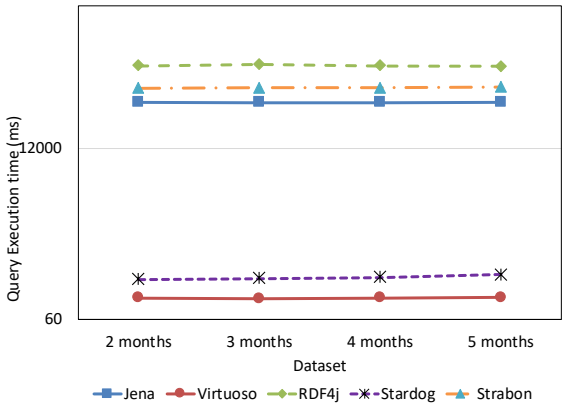


Figure 5.14: Q9 execution time

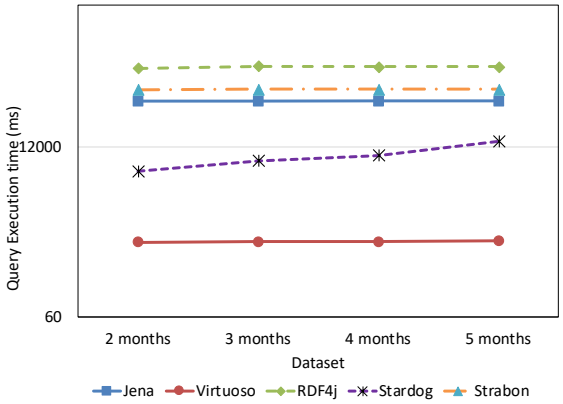


Figure 5.15: Q10 execution time

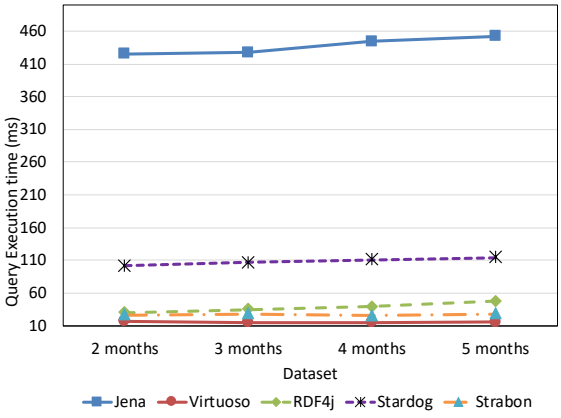


Figure 5.16: Q11 execution time

looking at the Q3 cost breakdown, the execution costs of Jena and Stardog are 48,833 (ms) and 11410 (ms), taking over 97% and 78% of query response time, respectively. Meanwhile, Virtuoso dominates only 74 (ms) which takes about 12% of the total cost.

The query parsing time only takes a small fraction of the total cost. This is applied to all stores. Regarding the cost of query optimization, we observe that this cost is different across the systems. Specifically, Virtuoso spends significantly more time with respect to others. An example of this is the Q6, Virtuoso spends 48 (ms) over the total time 49 (ms) for the query optimization, taking almost 100%. In contrast, the optimization cost of Q6 is 80 (ms) in Stardog, about 55% of the total cost.

In the following, for further analyzing the cost breakdown, we choose the Q10 as the representative for all queries. The reason for this is due to its high complexity, that covers almost all query characteristics and also demands a heavy computation. Analyzing the results in Figure 5.17, we observe that: (1) Optimization costs of this query in Virtuoso, Stardog, and Jena are fairly consistent and are not significantly affected by the dataset size. We attribute this to the effectiveness of query execution plan caching and the statistical approach taken in these systems. (2) On the contrary, RDF4J and Strabon clearly present growth in optimization costs with respect to the dataset size. This indicates that these two systems have applied different optimization techniques in the presence of different workloads.

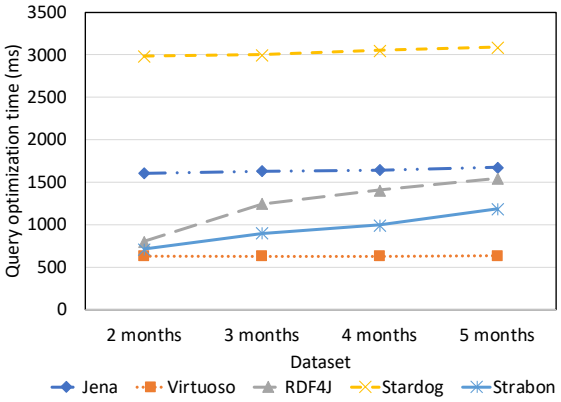


Figure 5.17: The optimization time of Q10 by varying increasing dataset

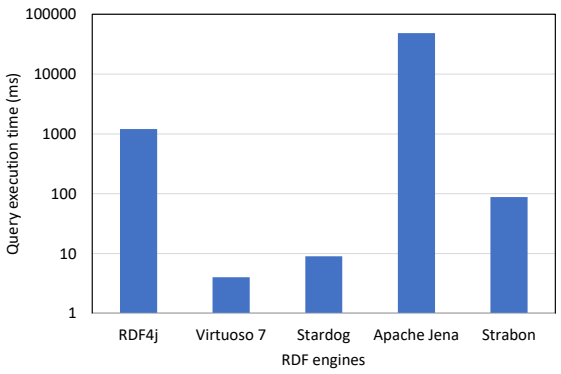


Figure 5.18: Q1 execution costs for 5 months dataset (in logscale)

Following these two observations mentioned above, it still lacks evidence to conclude which strategy performs better. It is evidenced by the unpredictable query processing behavior that draws our attention to Q1. Note that, the complexity of this query is low as it only requires simple spatial computation in conjunction with semantic triple matching. Among the four stores, Virtuoso spends 98% query run-time for query optimization. For Q1, the total query run-time is 80 (ms) and Virtuoso takes 75 (ms) for query optimization. However, the significant effort for optimization pays off. Figure 5.18 illustrates Q1 execution costs across different stores, where Virtuoso significantly outperforms other stores. In the meanwhile, RDF4J spends only 30% time for query optimization on Q1, much less than Virtuoso. However, the execution time is extremely high, which is 1210 (ms), far worse than Virtuoso.

In the same case of Q1, Apache Jena spends more time than RDF4J for query optimization, 1814 (ms) in comparison with 542 (ms). However, it results in catastrophic performance. Analyzing

Table 5.7: Cost breakdown of evaluated queries for 5 months dataset

Query	Jena			Virtuoso			RDF4j			Stardog			Strabon		
	Pars. (ms)	Opt. (ms)	Exe. (ms)	Pars. (ms)	Opt. (ms)	Exe. (ms)	Pars. (ms)	Opt. (ms)	Exe. (ms)	Pars. (ms)	Opt. (ms)	Exe. (ms)	Pars. (ms)	Opt. (ms)	Exe. (ms)
Q1	10	1814	48304	1	75	4	9	542	1210	2	101	9	11	551	88
Q2	7	105	53	1	70	12	6	1100	127	1	94	42	8	100	1203
Q3	15	1320	48833	1	500	74	16	2500	226414	1	3085	11410	20	3140	59830
Q4	13	1840	48305	2	560	27	15	1432	309528	1	3115	10883	19	4859	55427
Q5	14	1410	48739	1	50	4	15	90	14	2	73	66	21	85	6
Q6	13	163	10	1	48	0	16	813	356	1	80	64	23	90	815
Q7	20	1731	48415	2	72	4	21	1430	227404	1	204	3	30	2147	56817
Q8	10	159	373	1	65	4	8	1100	868	1	71	41	15	1425	568
Q9	16	1774	48388	1	110	8	12	2113	151653	1	105	136	17	3549	76977
Q10	22	1677	48470	2	635	8	21	1546	144141	2	3093	11201	33	1187	71636
Q11	5	37	410	1	15	0	4	20	4	1	46	67	6	19	3

```
(join
  (quadpattern
   (quad <urn:x-arq:DefaultGraphNode>
    ?station rdf:type got:ontology/WeatherStation)
     (quad <urn:x-arq:DefaultGraphNode>
        ?station geo:hasGeometry ?geoFeature)
   (propfunc spatial:withinCircle
     ?geoFeature (59.783 5.35 20 ''miles'')
    (table unit)
))
```

Listing 5.1: Jena optimization plan for Q1

the Jena query processing log, we attribute this to the inappropriate query execution plan generated for Q1. According to the data statistic, the most efficient way to execute Q1 is that the spatial filter with low selectivity should be executed first. After that, the results will be joined with the semantic triple matching with higher selectivity. This execution plan helps to eliminate all unnecessary intermediate results so that it will reduce the number of join operations. Unfortunately, Jena executes this query in the opposite way, as shown in Listing 5.1, that requires a lot of join operations. This results in the dramatical increase in query processing time. We attribute this to a lack of spatio-temporal statistical information about the dataset. Although Jena already provided a function to generate the statistic of a specified dataset, however, it only works for standard RDF dataset, not for spatio-temporal dataset.

5.7 DISCUSSIONS

Our performance study addresses a number of well-known RDF stores such as Virtuoso, RDF4J, Apache Jena, Stardog and Strabon in terms of data loading performance and query execution behaviors on processing over Linked Sensor Data. In order to analyze the system's strengths and weakness of these stores, we first carefully assess their data loading performance on sensor data. In this regard, along with the general RDF data loading evaluation, we also study the spatial and text data loading speed. Generally, Virtuoso and Stardog perform better than others. Moreover, we also learn that they have better supports on bulk and near real-time data import. In our opinion, this is an advanced feature that should be considered when selecting an RDF engine for sensor data.

Along with the different data aspects loading assessment, the corresponding query performance evaluations are also discussed. Unlike the existing approaches that only observe the overall query performance, we also analyze the dynamics and query processing behaviors of the test stores at a deeper detailed level. We split the query execution process into a series of sub-processes, which are query parsing, query optimization, and execution. Each sub-process is then evaluated separately. Through the evaluation results, with the general assessment, Virtuoso outperforms all other stores. It is followed by Stardog and Strabon. Moreover, we also learn that the time cost for each process is different across these stores. For example, Virtuoso dedicates significantly most time for query optimization process with respect to the others. On the contrary, Jena and RDF4J mostly focus on the execution process.

Beside the judgments about the overall engine's performance, several further findings related to data index and query optimization have been derived: (1) Due to the poor performance of the queries that require temporal analytical computing on sensor observation data, such as aggregation, a proper index approach for observation data is needed. Among these evaluated stores presented in this chapter, Strabon has a dedicated temporal index for data with timestamps, but it just focuses on querying temporal relations of two temporal objects rather than on supporting efficient analytical functions. According to our series of research on sensor data area [148, 147, 135, 154, 146, 155], sensor-based application users might be more interested in analytical queries rather than in temporal relation queries. (2) Selecting a wrong query execution plan is potentially catastrophic. In our experiments, for Jena, failure in query planning results in the worse performance. This is due to the lack of statistic on spatial and temporal data in sensor datasets. In particular, while most of RDF stores use statistics-based query planning models for query cost estimation, missing statistics on spatial and temporal data makes the query planning inaccurate.

5.8 SUMMARY

In this chapter, we present our comprehensive performance study of RDF stores for Linked Sensor Data. One of the main contributions of this chapter is to summarize the list of fundamental requirements of RDF stores so that they can be used for managing and querying sensor data. We have also described the abstract architecture design of current RDF database technologies that support spatio-temporal queries. Another main contribution of our work is to provide a comparative analysis of data loading and query performance of a representative set of RDF stores, namely Apache Jena, Virtuoso, RDF4J, Stardog and Strabon. In our evaluation, particular attentions have been given on evaluating the performance of geo-spatial search, temporal filter and full-text search over sensor data.

The investigations and findings demonstrate detailed aspects of RDF stores for managing Linked Sensor Data. Furthermore, these studies also provide a deeper understanding of the store's behaviors during data loading and query execution on spatio-temporal sensor data. This motivates us to propose our query processing engine as presented in the following chapter. Additionally, the results presented in this chapter show that a standalone RDF store could run into serious performance bottlenecks in the presence of a massive amount of Linked Sensor Data. Therefore, we will also investigate how to efficiently process large datasets using distributed systems in our approach. Furthermore, to address the query planning issue, we propose a learning approach for query planning that applies machine learning techniques. Our query planning approach is presented in Chapter 7.

6 EAGLE – SCALABLE QUERY PROCESSING ENGINE FOR LINKED SENSOR DATA

Motivated by the investigations and findings presented in the previous chapter, in this chapter, we presents EAGLE, a scalable query processing engine for Linked Sensor Data. EAGLE provides a flexible architecture for implementing efficient spatio-temporal query processing engines over historical Linked Sensor Data. It is worth mentioning that the processing scope of the engine is limited to answer the historical spatio-temporal queries. Streaming queries, hence are not taken into consideration. In the first section of this chapter, we describe the EAGLE framework and its components. Next, we elaborate on the spatio-temporal storage model for efficiently querying sensor observation data. In this section, we also present our proposed data partitioning strategy as well as the row-key design scheme for storing temporal data. After that, a detailed implementation will be given which consists of our indexing approaches and the query delegation model. Following the EAGLE's implementation is our SPARQL query language extensions to support querying Linked Sensor Data. An extensive experimental evaluation at the end of this chapter highlights the advantages of EAGLE over the existing approaches for querying Linked Sensor Data. Furthermore, this section also identifies what needs to be done to achieve further better performance of EAGLE.

This work appears in [154, 146].

6.1 EAGLE ARCHITECTURE

The architecture of EAGLE is illustrated in Figure 6.1. The engine accepts sensor data in RDF format as input and returns an output in SPARQL Result form[1]. The general processing works as follows. When the Linked Sensor Data is fed to the system, it is first analyzed by the Data Analyzer component. The Data Analyzer is responsible for analyzing and partitioning the input data based on the RDF patterns that imply the spatial and temporal context. The output sub-graphs of the Data Analyzer will be converted by the Data Transformer to the compatible formats of the underlying databases. The Index Router module then receives the transformed data and forwards them to the corresponding sub-database components in the Data Manager. In the Data Manager, we choose Apache Jena TDB[2], OpenTSDB[3] [177] and ElasticSearch[4] as underlying stores for such partitioned sub-graphs.

1 https://www.w3.org/TR/rdf-sparql-XMLres/
2 https://jena.apache.org/
3 http://opentsdb.net/
4 https://www.elastic.co/

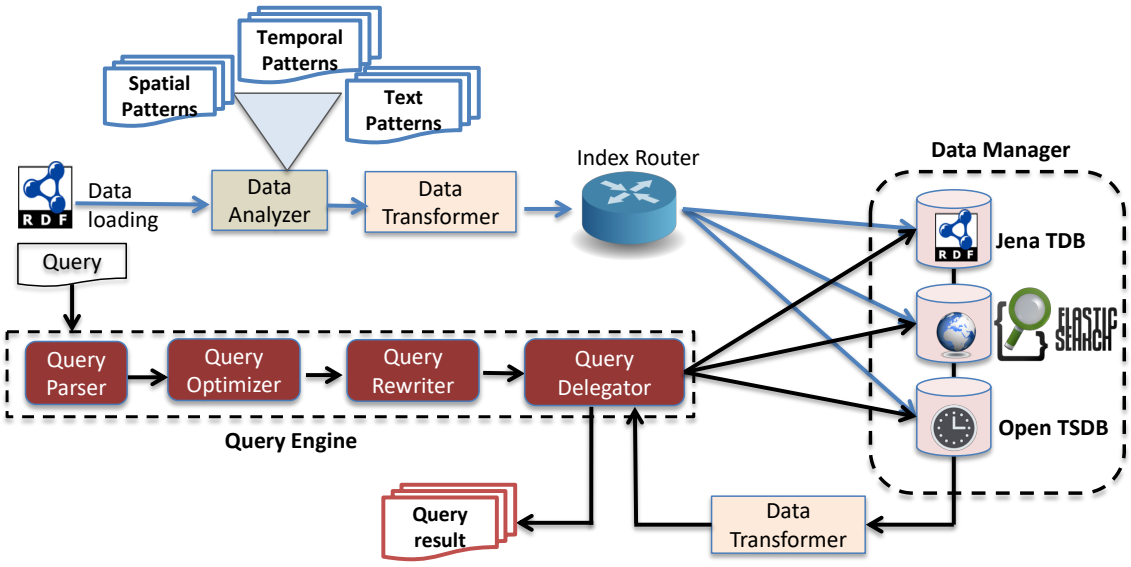


Figure 6.1: EAGLE's architecture

To execute the spatio-temporal queries, a Query Engine module is introduced. The Query Engine consists of several sub-components that are responsible for parsing the query, generating the query execution plan, rewriting the query into sub-queries and delegating sub-queries execution processes to the underlying databases. The Data Manager executes these sub-queries and returns the query results. After that, the Data Transformer transforms the query results accordingly to the format that the Query Delegator requires. Details of EAGLE's components are described in the following subsections.

6.1.1 Data Analyzer

As mentioned above, for the input sensor data in RDF format, the Data Analyzer evaluates and partitions them to the corresponding sub-graphs based on their (spatial, temporal or text). Data characteristics are specified via a set of defined RDF triple patterns. In EAGLE, these RDF triple patterns are categorized into three types: spatial patterns, temporal patterns, and text patterns. The spatial patterns are used to extract the spatial data that need to be indexed. Similarly, temporal patterns extract the sensor observation value along with its timestamp. The text patterns extract the string literals. An example of the partitioning process is illustrated in Figure 6.2. In this example, we define *(?s wgs84:lat ?lat. ?s wgs84:long ?long)* and *(?s rdfs:label ?label)* as the triple patterns used for extracting spatial and text data, respectively. For instance, assume that the system receives a set of input triples as follows:

```
:dublinAirport a geo:Feature;
        wgs84:lat "53.1324"^^xsd:float;
        wgs84:long "18.2323"^^xsd:float;
        rdfs:label "Dublin Airport".
```

As demonstrated in Figure 6.2, the two triples *(:dublinAirpot wgs84:lat "53.1324"^^xsd:float; wgs84:long "18.2323"^^xsd:float)* are found to be matched the defined spatial patterns *(?s wgs84:lat ?lat; wgs84:long ?long)*, thus, are extracted as a spatial graph. Similarly, we have the text sub-graph *(:dublinAirport rdfs:label "Dublin Ariport")* extracted. These sub-graphs will be transformed into compatible formats to be used by the indexing process in the Data Manager. The data transformation process will be presented in the following section.

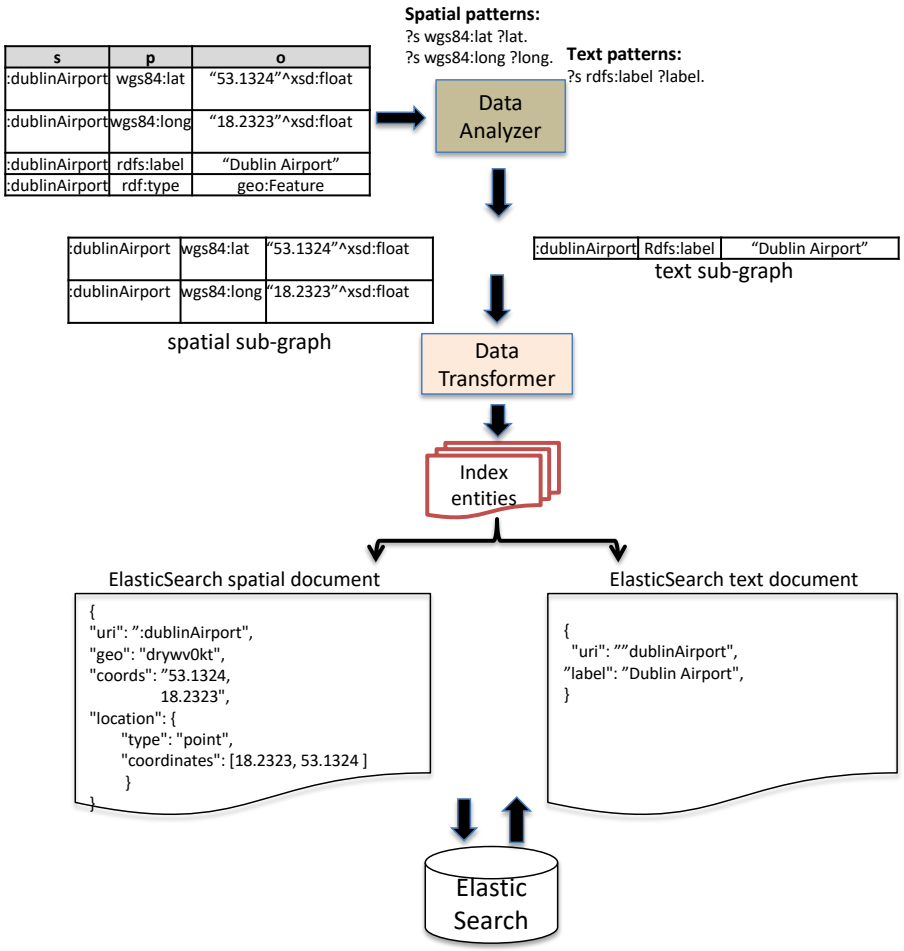


Figure 6.2: Transform spatial and text sub-graphs to ElasticSearch documents

6.1.2 Data Transformer

The Data Transformer is responsible for converting the input sub-graphs received from the Data Analyzer to the index entities. The index entities are the data records (or documents) constructed to a compatible data structure so that they can be indexed and stored in the Data Manager. Returning to the example in Figure 6.2, the Data Transformer transforms the spatial sub-graph and text sub-graph into ElasticSearch documents. In addition to transforming the sub-graphs into the index entities, the Data Transformer also has to transform the query outputs generated by the Data Manager to the format that the Query Delegator requires.

6.1.3 Index Router

The Index Router receives the index entities generated by the Data Transformer and forwards them to the corresponding database in the Data Manager. For example, the spatial and text index entities will be routed to ElasticSearch to index and ones that have temporal values will be transferred to the OpenTSDB cluster. For the index entities that do not match any spatial or temporal patterns, they will be stored in the normal triple store. Due to the fact that the access methods can vary across different databases, the Index Router, therefore, has to support multiple access protocols such as Rest APIs, JDBC, MQTT, etc.

6.1.4 Data Manager

Rather than rebuilding the spatio-temporal indices and functions into one specific system, our Data Manager module adopts a loosely coupled hybrid architecture that consists of different databases for managing different partitioned sub-graphs. More precisely, we used ElasticSearch to index the spatial objects and text values that occurr in sensor metadata. Similarly, we used a time-series database, namely OpenTSDB, for storing temporal observation values. The reasons for choosing ElasticSearch and OpenTSDB can be explained as follows: (1) ElasticSearch and OpenTSDB both provide flexible data structures which enable us to store sub-graphs which share similar characteristics but have different graph shapes. For example, *stationA* and *stationB* are both spatial objects but they have different spatial attributes (i.e., point vs. polygon, names vs. label, etc). Moreover, such structures also allow us to dynamically add a flexible number of attributes in a table without using list, set, or bag attributes or redefining the data schema. (2) ElasticSearch supports spatial and full-text search queries. Meanwhile, OpenTSDB provides a set of efficient temporal analytical functions on time-series data. All of these features are the key-point requirements for managing sensor data. (3) Finally, these databases offer the clustering features so that we are able to address the "big-data" issue, which is problematic for the traditional solution when dealing with sensor data.

For the non-spatio-temporal information that does not need to be indexed in the above databases, this will be stored in the native triple store. We currently use Apache Jena TDB to store such generic data. In the case of a small size dataset, it can be easily loaded into the RAM of a standalone workstation for the sake of boosting performance.

6.1.4.1 *Spatial-driven Indexing*

To enable querying of spatial data, we transform the sub-graph that contains spatial objects as a geo document and store it in ElasticSearch. Figure 6.2 demonstrates a process that transforms a semantic spatial sub-graph to an ElasticSearch geo document. Please be aware that, along with spatial attributes, ElasticSearch also allows the user to add additional attributes such as date-time, text description, etc. This advanced feature allows us to develop a more complex filter that can combine spatial filters and full-text search in a query.

The ElasticSearch geo document structure is shown in Listing 6.1. In this data structure, *location* is an ElasticSearch spatial entities to describe geo spatial information. Its has two properties: *type*

```
{
     <field_1>: <value_1>,
     . . .
     <field_n>: <value_n>,
     "location": {
         "type": <geo shape type>
         "coordinates": <points>
      }
 }
```

Listing 6.1: ElasticSearch geo document structure

and *coordinates*. *Type* can be point, line, polygon, envelope while *coordinates* can be one or more arrays of longitude/latitude pair. Details of the spatial index implementation will be discussed in Section 6.3.1.

6.1.4.2 *Temporal-driven Indexing*

A large amount of sensor observation data is fed as a time-series of numeric values such as temperature, humidity and wind speed. For these time-series data, we choose OpenTSDB (Open time-series Database) as the underlining scalable temporal database. OpenTSDB is built on top of HBase [59] so that it can ingest millions of time-series data points per second. As shown in Figure 6.1, input triples which are comprised of numeric values and time-stamps are analyzed and extracted based on the predefined temporal patterns. Based on this extracted data, an OpenTSDB record is constructed and then stored in OpenTSDB tables.

In addition to the numeric values and timestamps, additional information can be added to each data record of OpenTSDB. Such information also can be used to filter the temporal data. Additional information is selected by their regular use for filtering data in SPARQL queries. For example, a user might want to filter data by type of sensor, type of reading, etc. The data organization and schema design in OpenTSDB will be discussed in Section 6.2.

6.1.5 Query Engine

As shown in the EAGLE architecture in Figure 6.1, the query processing of EAGLE is performed by the Query Engine that consists of a Query Parser, a Query Optimizer, a Query Rewriter and a Query Delegator. It is important to mention that our query engine is developed on top of Apache Jena ARQ [96]. Therefore, the Query Parser is identical to the one in Jena. The Query Optimizer, Query Rewriter and Query Delegator have been implemented by modifying the corresponding components of Jena. For the Query Optimizer, in addition to Apache Jena's optimization techniques, we also propose a learning optimization approach that is able to efficiently predict a query execution plan for an unforeseen given spatio-temporal query. Details of our approach will be presented in Chapter 7.

The query engine works as follows. First, for a given query, the Query Parser translates it and generates an abstract syntax tree. Note that, we have modified the Query Parser so that it can adapt to our spatio-temporal query language. Next, the syntax tree is then mapped to the

```
(propfunc temporal:avg
  (?value ?obs ?sensor ?time) ("10/01/2017"^^xsd:dateTime "10/02/2017"^^xsd:dateTime)
  (join
    (graph <urn:x-arq:DefaultGraphNode>
      (propfunc geo:sfWithin
        ?stationGeo geo:sfWithin (40.417287 -82.907123 40    miles   )
        (table unit)))
    (quadpattern
      (quad <urn:x-arq:DefaultGraphNode> ?sensor sosa:isHostedBy ?weatherStation.)
      (quad <urn:x-arq:DefaultGraphNode> ?sensor ?sensor sosa:observes got:WindSpeedProperty.)
      (quad <urn:x-arq:DefaultGraphNode> ?obs sosa:madebySensor  ?sensor.)
    )))
```

Listing 6.2: Textual representation of spatio-temporal query tree in Example 5

SPARQL algebra expression, resulting in a query tree. In the query tree, there are two types of nodes, namely non-leaf nodes and leaf nodes. The non-leaf nodes are algebraic operators such as joins, and leaf nodes are the variables present in the triple patterns of the given query. Following the SPARQL Syntax Expressions[5], Listing 6.2 presents a textual representation of the query tree corresponding to the spatio-temporal query in Example 5 of Section 6.4.

Please be aware that the query tree generated by the Query Parser is just a plain translation of the initial query to the SPARQL algebra. At this stage, there is no optimization technique being applied yet. After that, the query tree is processed by the Query Optimizer. This component is responsible for determining the most efficient execution plan with regard to the query execution time and resource consumption. After having a proper execution plan, it is passed to the Query Rewriter for any further processing needed. Basically, the Query Rewriter rewrites the query operators to the compatible query language of the underlying databases. In the next step, the Query Delegator delegates these rewritten sub-queries to the corresponding database in the Data Manager. For example, the sub-query that contains the spatial operator or full-text search will be evaluated by ElasticSearch, while the temporal operator is executed by OpenTSDB. For the non-spatio-temporal queries, they are processed by Jena. After having the sub-queries executed, the query results need to be transformed to the format that the Query Delegator requires. The Query Delegator then performs any post-processing actions needed. The final step involves formatting the results to be returned to the user.

6.2 A SPATIO-TEMPORAL STORAGE MODEL FOR EFFICIENTLY QUERYING ON SENSOR OBSERVATION DATA

As mentioned in Section 6.1, we chose OpenTSDB as an underlying temporal database for managing sensor observation data. In this section, we present a preliminary design of the OpenTSDB data schema used for storing these data sources. Due to the data-centric nature of wide column key-value stores of OpenTSDB, there are two most important decisions on storage model design that can affect to the system performance, which are: the form of the row keys and the partition of data. This section will present in details our decisions for rowkey design and the data

5 https://jena.apache.org/documentation/notes/sse.html

partitioning strategy that aim to enhance the data loading and query performance on sensor observation data.

6.2.1 OpenTSDB Storage Model Overview

OpenTSDB is a distributed, scalable, time-series database built on top of Apache HBase [59], which is modeled after Google's BigTable [34]. It consists of a time-series daemon (TSD) along with a set of command-line utilities. Data reading and writing operations in OpenTSDB are primarily achieved by running one or more of the TSDs. Each TSD is independent. There is no master, no shared state so that many TSDs can be deployed at the same time, depending on the loading throughput requirement. The OpenTSDB architecture is illustrated in Figure 6.3.

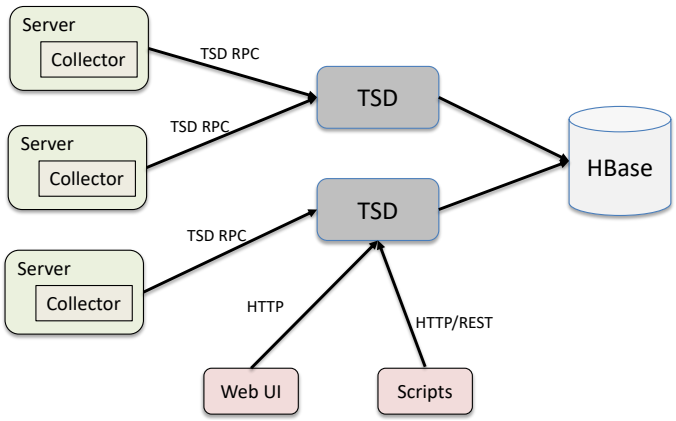


Figure 6.3: OpenTSDB architecture

Data in OpenTSDB are stored in a HBase table. A table contains rows and columns, much like a traditional database. A cell in table is a basic storage unit, which is defined as <*RowKey,ColumnFamily:ColumnName,TimeStamp*>. There are two tables in OpenTSDB, namely *tsdb* and *tsdb-uid*. The *tsdb-uid* table is used to maintains an index of globally unique identifiers (UID) and values of all metrics and tags for data points collected by OpenTSDB. In this table, two columns exist, one called *name* that maps an UID to a string, and another table, denoted as *id*, mapping strings to UIDs. Each row in the column family will have at least one of following three columns with mapping values: *metrics* for mapping metric names to UIDs, *tagk* for mapping tag names to UIDs, *tagv* for mapping tag values to UIDs. Figure 6.4 illustrates the logical view of *tsdb-uid* table.

A central component of OpenTSDB architecture is the *tsdb* table that stores our time-series observation data. This table is originally designed to not only support time-based queries but also to allow additional filtering on metadata, represented by *tag* and *tag value*. This is accomplished through careful design of the rowkey. As described in Table 6.1, an OpenTSDB rowkey consists of 3 bytes for the metric id, 4 bytes for the base timestamp, and 3 bytes each for the tag name ID and tag value ID, repeated. Notice that the OpenTSDB is optimized for metric-centric queries, thus, the metric UID comes first. Moreover, data stored in OpenTSDB are ordered by rowkey, so that the entire history for a single metric is stored as contiguous rows. Within the run of rows for a metric, they are ordered by base timestamp. The base timestamp in the rowkey is rounded

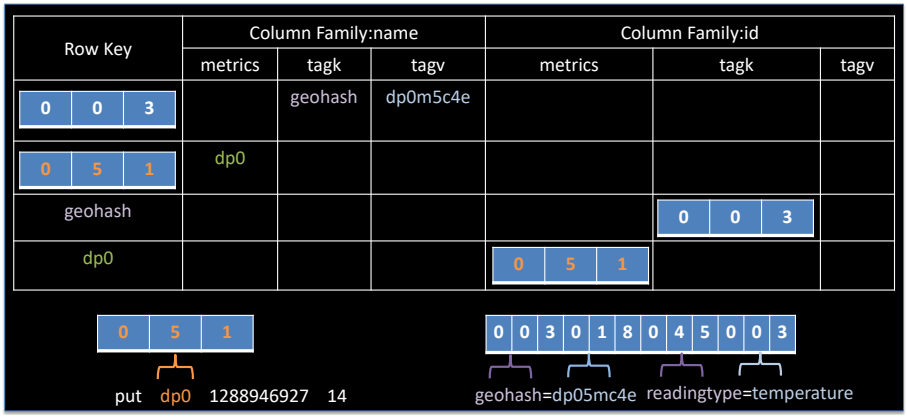


Figure 6.4: OpenTSDB *tsdb-uid* table

down to the nearest 60 minutes so that a single row stores a bucket of measurements for an hour. The tag name and value UIDs come last in the rowkey. As mentioned earlier, storing all these attributes in the rowkey allows them to be used for filtering the search results.

Table 6.1: OpenTSDB row key format

Element name	Metric UID	Base-timestamp	Tag names	Tag values	...
Size	3 bytes	4 bytes	3 bytes	3 bytes	...

With each additional tag, 6 bytes are added to the row key size. The first 2 bytes are reserved for the column qualifier. In these 2 bytes, 12 bits are used to store an integer value, which is delta from the base timestamp in the row key. The last 4 bits describe the data stored, in which the first bit indicates whether the value is an integer or a floating-point. The remaining 3 bits indicate the length of the data. The measure/value of a data point is stored as the cell value and is reserved to 8 bytes.

Figure 6.5 presents an example of *tsdb* rowkey. As shown in this figure, the schema contains only a single column family, namely *t*. This is due to the requirement of HBase that a table has to contain at least one column family [48]. In OpenTSDB, the column family is not so important as it does not affect the organization of data. The column family *t* might consist of one or many column qualifiers representing delta elapse from the base timestamp. In this example, *16* is the value, *1288946927* is the base timestamp and column qualifier *+300* is the delta elapse from base timestamp.

6.2.2 Designing a Spatio-temporal Rowkey

In order to make well-informed choices of the rowkey design, we first identified common data access patterns required by the user application when querying the sensor observation data. In the following, we enumerated a few common queries that can be expected by the realistic sensor-based applications presented in [150, 182, 139, 111]:

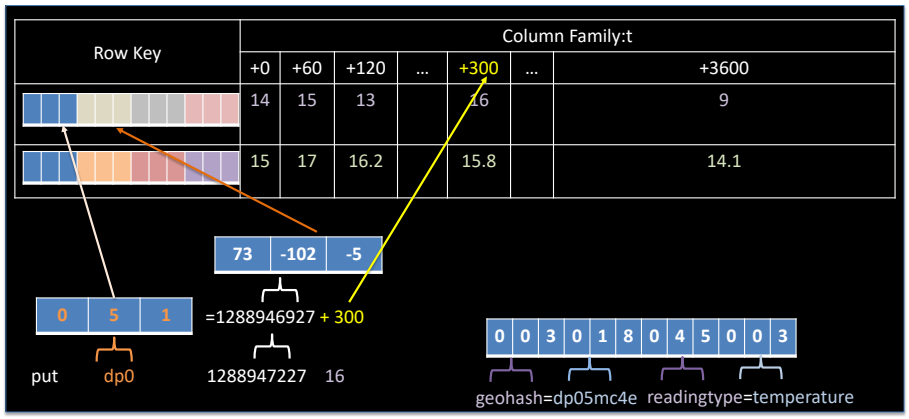


Figure 6.5: OpenTSDB *tsdb* table

- A user may request meteorological information of an area over a specific time interval. The query may include more than one measurement values, i.e. humidity, wind speed along with the temperature.
- A user may request the average observation value over a specific time interval using variable temporal granularity i.e. hourly, daily, monthly, etc.
- A user may request statistical information about the observation data that are generated by a specific sensor station.
- A user may ask for statistical information, such as the hottest month over the last year for a specific place of residence. Such queries can become more complex if the residence address is not determined by city name or postal code but by its coordinate.

There can be different rowkey design approaches for answering the aforementioned queries. Nevertheless, to have fast access to a relevant data based on the rowkey, there are two points needed to be taken into consideration when designing a rowkey schema for storing sensor observation data in OpenTSDB table: (1) data should be evenly distributed across all RegionServers to avoid the region hot-spotting performance [48]. Note that, a bad key design will lead to sub-optimal load distribution. The solution to address this issue will be presented in Section 6.2.3. (2) The spatio-temporal locality of data should be preserved. In other words, data of all the sensors that locate within the same area should be stored in the same partitions on the disk. The latter is essential in order to accelerate range scans since users will probably request data of a specific area over a time interval instead of just a single point in time.

Starting with the row key schema, we have to decide what information and in which order will be stored in the row key. Since spatial information is usually the most important aspect of user queries, encoding the sensor location in rowkey is prioritized. In this regard, a *geohash* algorithm is selected. Recall that, a geohash is a function that turns the latitude and longitude into a hash string. A special feature of geohash is that, for a given geohash prefix, all the points within the same space match the common prefix. To make use of this feature, we encode the first three characters of geohash prefix as the *metric uid* of our rowkey schema. The length of the geohash prefix that is used to encode the *metric uid* can be various, depending on the data density. Data stored in the *tsdb* table are sorted on rowkey, thus, encoding geohash as *metric uid*, which is the

first element of rowkey, ensures the data of sensor stations close to each other in space are close to each other on disk. Next, we append the measurement timestamp as the second element of a rowkey in order to preserve temporal ordering. At this stage, we accomplish the goal (2).

After defining the first two elements of the row key, the tag names and tag values must be specified. In OpenTSDB, tags are used for filtering data. Based on the summary of common data access patterns above, we recognize that users may filter data by either a detailed location, or by a specific sensor, or by a single type of sensor reading. Therefore, the following tags are defined: (1) the *geohash* tag to store the full geohash string representing sensor station location; (2) the *sensorId* to present the full IRI of sensor that generates corresponding observation data; (3) the *readingtype* to indicate the observed property of observation data. These tags are then concatenated after the rowkey. The full form of our proposed row key design is depicted in Figure 6.6.

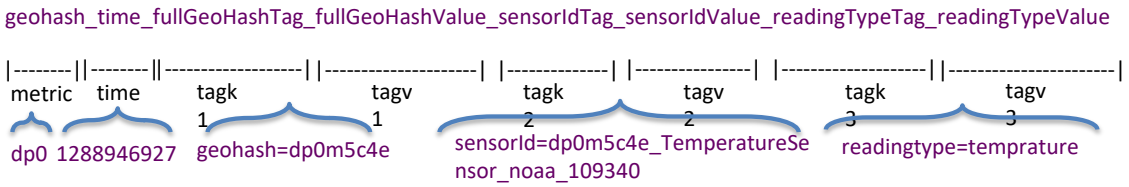


Figure 6.6: OpenTSDB rowkey design for storing observation data

6.2.3 Spatio-temporal Data Partitioning Strategy

Data partitioning has a significant impact on parallel processing platforms like OpenTSDB. If the sizes of the partitions, i.e., the amount of data per partition, are not balanced, a single worker node has to perform all the work while other nodes idle. To avoid this imbalance performance, in this section, we will present our data partitioning strategy that split data into multiple partitions and also exploits the spatio-temporal characteristics of sensor data.

As mentioned earlier, we store observation data in OpenTSDB *tsdb* table, which is originally an HBase table. By design, an HBase table can consist of many regions. A region is a table storage unit, that contains all the rows between the start key and the end key assigned to that region. Regions are managed by the Region Servers, as illustrated in Figure 6.7. Note that, in HBase, each region server serves a set of regions, and a region can be served only by a single region server. The HMaster is responsible to assign regions to region servers in the cluster.

Initially, when a table is created, it is allocated with a single region. Data are then inserted into this region. If the number of data records stored in this region exceeds the given threshold, HBase will partition it into two roughly equal-sized child regions. As more and more data are inserted, this splitting operation is performed recursively. Figure 6.8 describes the table splitting in HBase.

Basically, table splitting can be performed automatically by HBase. The goal of this operation is to avoid hot-spotting performance. However, table splitting is a costly task and can result in latency increased, especially during heavy write loads. In fact, splitting is typically followed by regions moving around to balance the cluster, which adds to the overhead and heavily affects

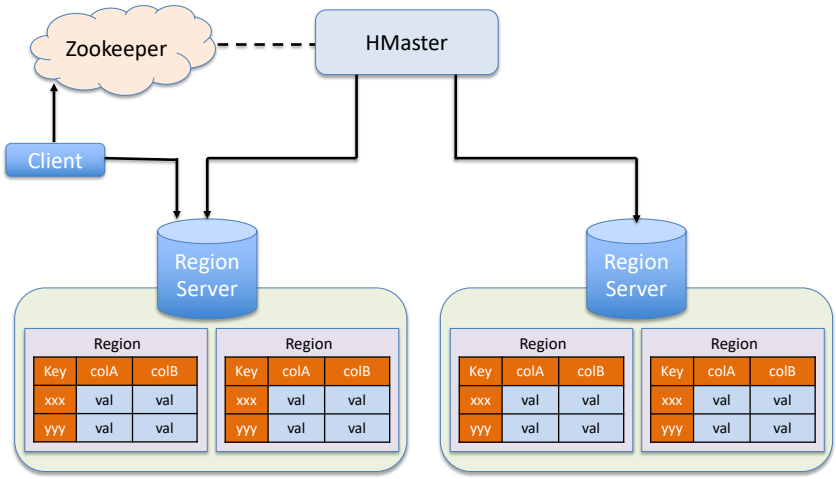


Figure 6.7: HBase tables and regions

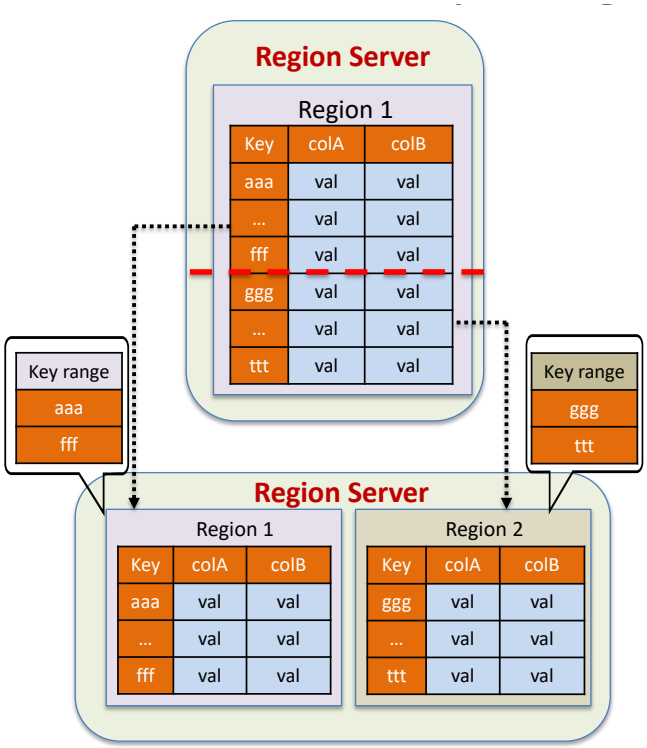


Figure 6.8: HBase table splitting

to cluster performance. Therefore, to avoid this costly operation, we partition the *tsdb* table at the time of table creation using the pre-splitting method. For different data sources, the data partitioning strategy might be varied as it is very dependent upon the rowkey distribution. Therefore, a good rowkey design is also a key factor of the effectiveness of the partitioning strategy.

Although HBase already includes partitioners, they do not make use of the spatio-temporal characteristics. In our approach, we partition the *tsdb* table into a pre-configured number of regions. Each region is assigned with a unique range of geohash prefix. Figure 6.9 illustrates our

spatio-temporal data partitioning strategy. In this figure, region 1 is assigned with a range *[ou1-9xz]*, indicating that all data records that have rowkey prefixes within the range of *[ou1-9xz]* will be stored in region 1. By applying the spatio-temporal partitioning strategy, we ensure that all sensor data that are near to each other in time and space will be stored in the same partition. As demonstrated later in our experiments in Section 6.5, with the help of this strategy, the EAGLE engine is able to quickly locate what partitions actually have to be processed for a query. For example, a spatial Intersect query only has to check the items of partitions where the partition bounds themselves intersect with the query object. Such a check can decrease the number of data items to process significantly and thus, also reduce the processing time drastically.

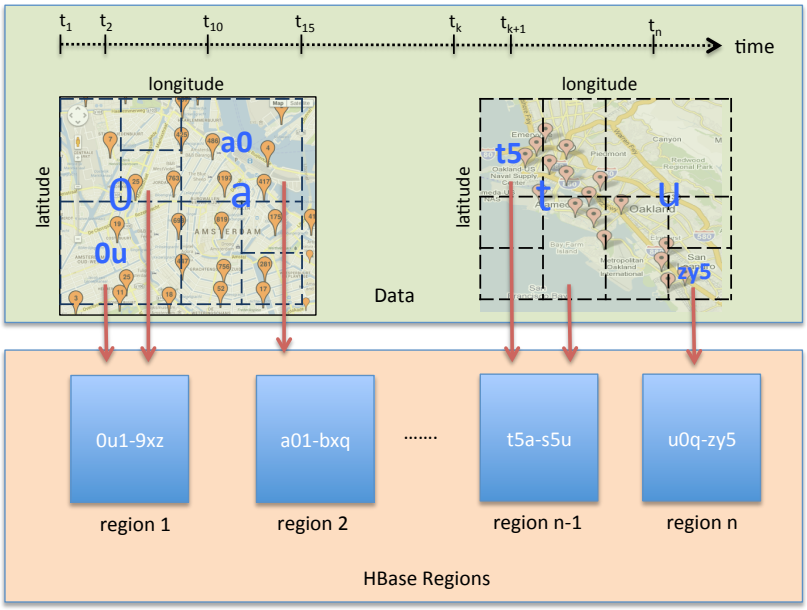


Figure 6.9: Spatio-temporal data partitioning strategy

6.3 SYSTEM IMPLEMENTATION

In this section, we will present in details the EAGLE's implementation based on the architecture presented in Section 6.1.

6.3.1 Indexing Approach

In order to ensure efficient execution of spatio-temporal queries in EAGLE, we must provide a means to extract and index portions of the sensor data based on spatial, temporal and text values. In this section, we firstly present how to define triple patterns for extracting spatial, temporal and text data. After that, we describe in detail the indexing schemes for each aspect of sensor data.

```
<#definition> a spatial:EntityDefinition ;
    spatial:entityField      "uri" ;
    spatial:geoField         "geo" ;
    # custom geo predicates for 1) Latitude/Longitude Format
    spatial:hasSpatialPredicate (
         [ spatial:latitude wgs84:lat ; spatial:longitude wgs84:long ]
         ) ;
    # custom geo predicates for 2) Well Known Text (WKT) Literal
    spatial:hasWKTPredicates (geo:asWKT) ;
    # custom SpatialContextFactory for 2) Well Known Text (WKT) Literal
    spatial:spatialContextFactory
         "com.spatial4j.core.context.jts.JtsSpatialContextFactory"
    .
```

Listing 6.3: Defening triple patterns to extract spatial data

6.3.1.1 *Defining Triple Patterns for Extracting Spatio-temporal Data*

As mentioned earlier, spatio-temporal and text data included in input RDF sensor data are extracted if their data graph matches the pre-defined triple patterns. In EAGLE, we support several common triple patterns already defined in a set of widely-used ontologies for annotating sensor data, such as GeoSPARQL, OWL-Time, WGS84, SOSA/SSN. For example, we support the GeoSPARQL pattern *(?s geo:asWKT ?o)* for extracting spatial data. In addition to the commonly used patterns, the ones with user-customized vocabularies are also allowed in our engine. All triple patterns for extracting spatio-temporal data are stored in the Data Analyzer component. In EAGLE's implementation, these patterns can be defined by either in the configuration files or via provided procedures. Listing 6.3 illustrates an example of using configuration file to define triple patterns for extracting spatial data. In this example, the defined predicates, *wgs84:lat/wgs84:long* and *geo:asWKT*, are used to extract the spatial information from input RDF graphs. To reduce the learning efforts, our configuration file syntax fully complies with the Jena assembler description syntax [95].

The process to define triple patterns for extracting temporal data is a bit more complicated. As mentioned in Section 6.2.3, in addition to the observation value and its timestamp, our OpenTSDB rowkey scheme also stores other attributes such as the full geohash prefix, observed property, sensor IRI, etc. Therefore, the triple patterns for extracting this additional information should also be defined. An example of defining triple patterns for extracting temporal data is illustrated in Listing 6.4.

```
<#definition> a temporal:EntityDefinition ;
    temporal:hasTemporalPredicate (
         [ temporal:value sosa:hasSimpleResult ; temporal:time sosa:resultTime]
         [ temporal:value sosa:hasSimpleResult ; temporal:time owl-time:inXSDDateTimeStamp]
         ) ;
     temporal:hasMetadataPredicate (
     [ temporal:sensor sosa:madeBySensor ; temporal:readingType sosa:observedProperty]
     ) ;
```

Listing 6.4: Temporal triple patterns and its metadata declaration

In the above example, triple patterns for extracting temporal value are defined as an instance of the *temporal:EntityDefinition*. Its property, *temporal:hasTemporalPredicate*, indicates the RDF predicates used in the matching process of temporal data and timestamp. For example, a pair *(temporal:value sosa:hasSimpleResult)* denotes that the *object* of triples that match pattern *(?s sosa:hasSimpleResult ?o)* will be extracted as a temporal value. Similarly, a pair *(temporal:time sosa:resultTime)* specifies the predicate *sosa:resultTime* used for extracting the timestamp. Finally, triple patterns that describe additional information are defined under the *temporal:hasMetadataPredicate* property, i.e, the sensor URI (extracted by *sosa:madeBySensor*) and the reading type (extracted by *sosa:observedProperty*). It is worth mentioning that the current generation of EAGLE supports only the time instant. Time interval support will be added in the next version.

6.3.1.2 *Spatial and Text Index*

We store spatial and text data in the ElasticSearch cluster. Therefore, we first need to define the ElasticSearch mappings for storing these data. In ElasticSearch, *mapping* is the process of defining how a document, and the fields it contains, are stored and indexed.

The ElasticSearch geo mapping for storing spatial objects is shown in Listing 6.5. In this mapping, the uri field stores the geometry IRI, and the full_geohash field stores the 12-bit geohash string of the sensor location. Similarly, the ElasticSearch mapping for storing text value is shown in Listing 6.6.

```
{
    "mappings" : {
            "geometries": {
                "properties": {
                        "uri": {"type": "string","index" : "not_analyzed"},
                        "full_geohash": {"type": "string","index" : "not_analyzed"},
                        "location": {
                                "type": "geo_shape",
                                    "tree": "geohash",
                                    "precision": "1m"
                        },
                        "coords": {
                            "type": "geo_point",
                            "geohash_prefix":     true,
                                    "geohash_precision":  "1km"
                    }
                }
            }
    }
}
```

Listing 6.5: ElasticSearch mapping for spatial index.

In EAGLE, we support both *bulk* and *near real-time* data indexing. *Bulk* index is used to import data that are stored in files or in the triple store. For this, we provide a procedure *build_geo_text_index()*. After having the ElasticSearch mappings defined, the *build_geo_text_index()* is called to construct a spatial index for a given dataset. The pseudo code of this procedure is given in Algorithm 6.1. In contrast to the *bulk* index, the *near real-time* index is used to index the data that are currently streaming to the engine. In this regard, a

```
{
    "mappings" : {
            "text": {
                "properties": {
                        "uri": {"type": "string","index" : "not_analyzed"},
                        "full_geohash": {"type": "string","index" : "not_analyzed"},
                        "country": {"type": "string","index" : "analyzed"},
                        "city": {"type": "string","index" : "analyzed"},
                        "label": {"type": "string","index" : "analyzed"},
                        "address": {"type": "string","index" : "analyzed"},
                }
            }
    }
}
```

Listing 6.6: ElasticSearch mapping for text index.

procedure *dynamic_geo_text_index()* is introduced to extract spatial and text data from a streaming triple and index them in ElasticSearch. Algorithm 6.2 describes the pseudo code of the *dynamic_geo_text_index()* procedure.

Algorithm 6.1: A procedure that will read a given dataset and index its spatial and text data in ElasticSearch.

```
function build_geo_text_index (datasetLocation);
Input  : datasetLocation: The dataset file or directory path
if geo_mapping does not exist then
    create geo_mapping
end
D = load_dataset(datasetLocation) ;
foreach triple t of dataset D do
    p = t.predicate;
    o = t.object;
    if isSpatialPredicate(p) and isSpatialValue(o) then
        geoDocument = build_ES_document(t);
        insert document (geoDocument)
        into geo_mapping
    end
    if isTextPredicate(p) and isTextValue(o) then
        textDocument = build_ES_document(t);
        insert document (textDocument)
        into text_mapping
    end
end
```

6.3.1.3 *Temporal Index*

We provide the procedure, namely *build_temporal_index*, to construct a temporal index for given sensor observation data. The *build_temporal_index* procedure is split into three steps, as illustrated in Algorithm 6.3. The procedure is explained as follows. Firstly, the sensor metadata is loaded into the system memory. This metadata is used later for quickly retrieving information needed for constructing OpenTSDB data row, such as sensor location, observed properties, etc. The loaded metadata can be stored in a key-value data structure such as hashmap, array, etc.

Algorithm 6.2: A procedure that will read a streaming triple and index its spatial and text data in ElasticSearch.

```
function dynamic_geo_text_index (Triple t);
Input  : Triple t
p = t.predicate;
o = t.object;
if isSpatialPredicate(p) and isSpatialValue(o) then
    geoDocument = build_ES_document(t);
    insert document (geoDocument)
    into geo_mapping
end
if isTextPredicate(p) and isTextValue(o) then
    textDocument = build_ES_document(t);
    insert document (textDocument)
    into text_mapping
end
```

Algorithm 6.3: A procedure that will read an observation data and index its temporal information in OpenTSDB.

```
function build_temporal_index (Observation obs);
Input  : Observation obs: An observation data in RDF format
// Step 1:  load sensor metadata file or graph and query it
if metadata is empty then
    metadata = query_dataset(datasetLocation, query) ;
end

// Step 2:  extract OpenTSDB indexing required information from observation data
foreach triple t of observation obs do
    if isTemporalPredicate(p) and isNumeric(o) then
        value = extract_value(obs);
    else if isTemporalPredicate(p) and isDateTime(o) then
        timestamp = extract_time(obs);
    else if isMetadataPredicate(p) and isIRI(o) then
        sensorIRI = extract_sensorIRI(obs);

    // Retrieve observed property and location from metadata based on sensorIRI
    property = retrieve_property(sensorIRI,metadata);
    location = retrieve_location(sensorIRI,metadata);

    // generate geohash prefix from sensor location
    geohash_prefix = build_geohash_prefix(location);
end

// Step 3:  build OpenTSDB record and store it via OpenTSDB APIs
record = build_OpenTSDB_record(value,timestamp,sensorURI,property,geohash_prefix);
put record into tsdb table
```

In the second step, we extract the observation value, its timestamp, and the IRI of source sensor based on the defined triple patterns. After having this information extracted, corresponding observed property and sensor location are then retrieved by querying the loaded metadata in step 1. After that, from the retrieved sensor location, the corresponding geohash prefix is generated via a *build_geohash_prefix* procedure. The second step is demonstrated in Figure 6.10.

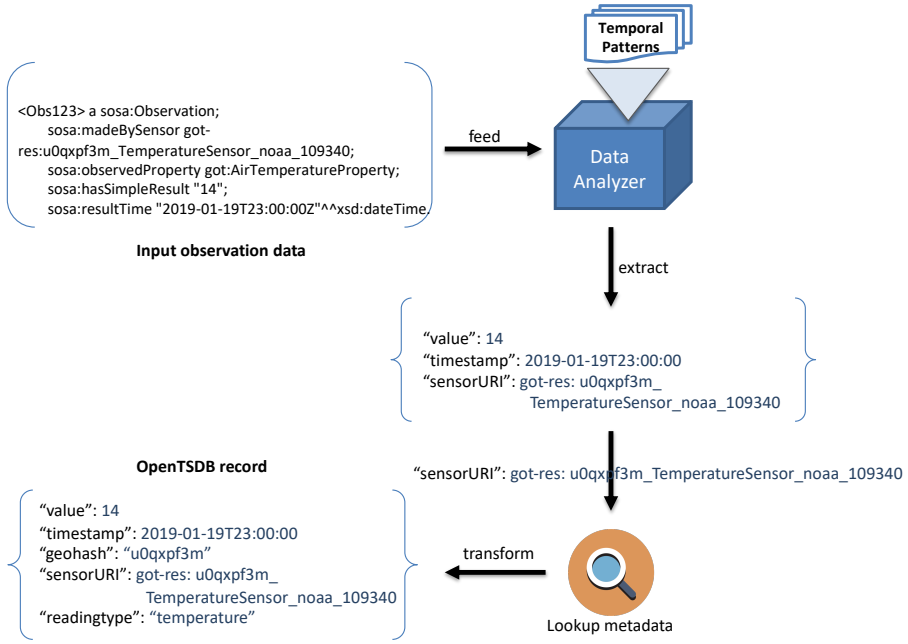


Figure 6.10: An example of temporal information extraction process

The final step is to generate an OpenTSDB data record from the data extracted in the previous steps and store it into OpenTSDB *tsdb* table. Data are indexed by calling OpenTSDB APIs such as *put* command, REST APIs, etc. Figure 6.11 illustrates a simple data insert operation in OpenTSDB using *put* command.

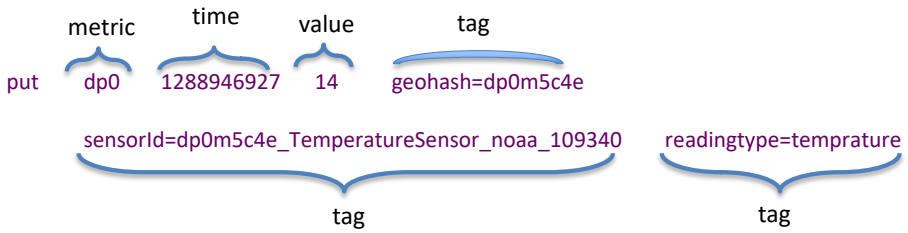


Figure 6.11: TSDB Put example

6.3.2 Query Delegation Model

Our query execution process of EAGLE is implemented by a query delegation model which breaks the input query into sub-queries that can be delegated to the underlying sub-components such as ElasicSearch, OpenTSBD, and Apache Jena. In this model, a spatio-temporal query can be represented by the SPARQL Query Graph Model (SQGM) [80]. A query translated into SQGM

can be interpreted as a planar rooted directed labeled graph with vertices and edges representing operators and data flows, respectively. In SQGM, an operator processes and generates either an RDF graph (a set of RDF triples), a set of variable bindings or a boolean value. Any operator has the properties *input* and *output*. The property *input* specifies the data flow(s) providing the input data for an operator and *output* specifies the data flow(s) pointing to another operator consuming the output data.

An evaluation process of the graph is implemented by following a post-order traversal, during which the data are passed from the previous node to the next. In this tree, each child node can be executed individually as asynchronous tasks, which can be carried out in different processes on different computers. Therefore, our system delegates some of those evaluation tasks to different distributed backend repositories, which can provide certain function sets, e.g., geospatial functions (by ElasicSearch), temporal analytical functions (by OpenTSDB), BGP matching (by Jena) and achieve the best performance in parallel. Figure 6.12 shows an example of SQGM tree on which a spatial filter node is rewritten to geospatial query and then delegated to ElasticSearch while the BGP matching query is executed by Jena.

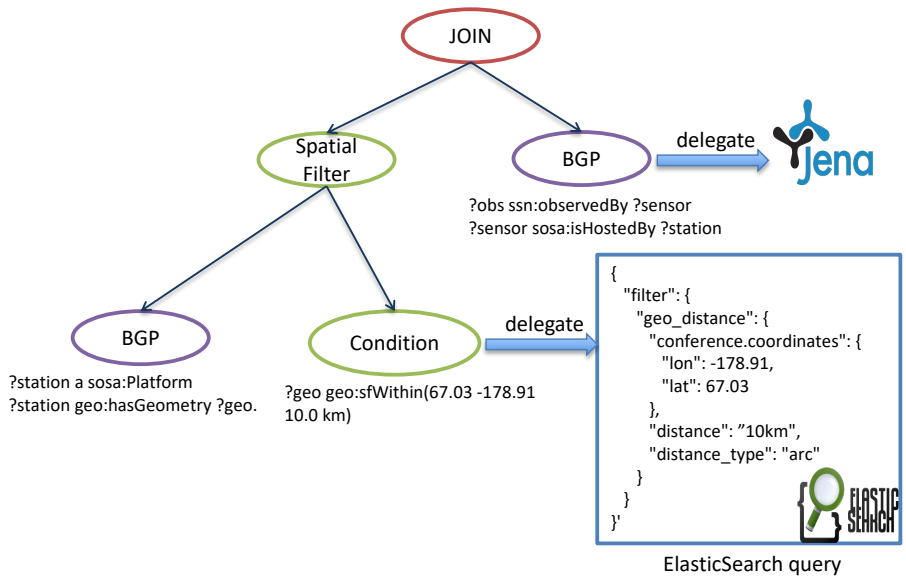


Figure 6.12: Delegating the evaluation nodes to different backend repositories

6.4 QUERY LANGUAGE SUPPORT

In this section, we present our SPARQL query language extensions for querying Linked Sensor Data. We adopt the GeoSPARQL syntax [19] into our proposed extensions for querying topological relations of spatial objects. Furthermore, we also introduce a set of novel temporal analytical property functions for querying temporal data.

6.4.1 Spatial Built-in Condition

Theoretically, a spatial built-in condition is used to express the spatial constraints on spatial variables. As previously mentioned, our proposed SPARQL's extensions adopt the GeoSPARQL syntax for querying spatial aspect of sensor data. Therefore, in this section, we paraphrase the notion of the spatial built-in conditions that are related to the EAGLE's implementation. More details of GeoSPARQL built-in condition can be found in [19]. It is important to mention that, within the scope of this thesis, we only focus on the qualitative spatial function.

A qualitative spatial function is a Boolean function f_s, defined as follows:

$$f_s : G \times G \rightarrow \mathbb{B} \tag{6.1}$$

where G is a set of geometries.

In the current version of EAGLE, several topological relations are supported: *disjoint, intersect, contains, within*. Following the *qualitative spatial function* definition, we then define a *qualitative spatial expression*, denoted by *se*:

$$< se > ::= f_s(g_1, g_2) \tag{6.2}$$

where $g_1, g_2 \in G \cup V$. V is a set of variables.

A ***spatial built-in condition*** is then defined by using the qualitative spatial function, logical connectives $\neg, \wedge, \vee$:

- If $< se >$ is a *qualitative spatial function*, then $< se >$ is a *spatial built-in condition*.
- If R_1, R_2 are *spatial built-in conditions*, then $(\neg R_1)$, $(R_1 \vee R_2)$, and $(R_1 \wedge R_2)$ are *spatial built-in conditions*.

6.4.2 Property Functions

In addition to spatial built-in condition, we also define a set of spatio-temporal and full-text search property functions. By definition, property function is an RDF predicate in SPARQL query that causes triple matching to happen by executing some specific data processing other than usual graph matching. Property functions must have fixed URI for the predicate and can not represent query variables. The subject or object of these functions can be a list.

In our query language support, the property functions are categorized into three types, depending on their function, which are spatial property function, temporal property functions, and full-text search property functions. These three types of property functions are assigned with different URIs (<*geo:*>, <*temporal:*>, <*text:*>). Spatial and temporal property functions are defined as follows.

Drawing upon the theoretical treatments of RDF in [140], we assume the existence of pairwise-disjoint countably infinite sets I, B and L that contain IRIs, blank nodes and literals respectively. V is a set of query variables. We denote a set of spatial property functions like I_{SPro}. Similarly, let I_{TPro} be a set of temporal property functions. I_{SPro}, I_{TPro} and I are also pairwise-disjoint.

A triple that contains a spatial property function is defined in the following form:

$$(I \cup B) \times I_{SPro} \times (I \cup L \cup V). \tag{6.3}$$

Example 1 The following is the example of spatial property function, namely *geo:sfWithin*, to find all the ?geo$_1$ objects that are within ?geo$_2$

?geo$_1$ **geo:sfWithin** ?geo$_2$.

Similarly, the temporal property function is defined as follows:

$$(I \cup B) \times I_{TPro} \times (I \cup L \cup V). \tag{6.4}$$

An example of temporal property function is *temporal:avg*. For the full-text search property function, we only support one property function, namely *text:match*. The usages of the property functions will be demonstrated via examples in Section 6.4.3.

6.4.3 Querying Linked Sensor Data by Examples

This section presents the syntax of our proposed SPARQL's spatio-temporal extensions through a series of examples involving Linked Sensor Data. The dataset used throughout the examples is the linked meteorological data described in Chapter 4. The namespaces used in the examples are listed in Appendix A.

Example 2 (Spatial built-in condition query). Return the IRIs and coordinates of all weather stations that locate in Dublin city. The query is shown in Listing 6.7.

```
SELECT ?weatherStation ?lat ?long
WHERE
{
    ?city dbo:type dbr:Capital;
          a dbo:Place;
          foaf:name "Dublin";
          geo:hasGeometry ?cityGeo.
    ?cityGeo geo:asWKT ?cityWkt.
    ?weatherStation geo:hasGeometry ?stationGeo.
    ?stationGeo       wgs84:lat ?lat.
    ?stationGeo       wgs84:long ?long.
    BIND (STRDT(CONCAT("POINT(",?long, " ", ?lat, ")"),sf:WktLiteral) as ?stationWkt) .
    FILTER(geo:sfWithin(?stationWkt,?cityWkt).
}
```

Listing 6.7: Spatial built-in condition query.

Let us now explain the query syntax by referring to the above example. Recall that our spatial query language adopts GeoSPARQL syntax, hence, all the GeoSPARQL prefixes, as well as its spatial datatypes, remain unchanged. As illustrated in the query, the spatial variables, the

```
SELECT count(?weatherStation)
WHERE
{
    ?weatherStation geo:hasGeometry ?stationGeo.
    ?stationGeo geo:sfWithin (59.783  5.35 20 'miles').
}
```

Listing 6.8: Spatial property function query.

?cityWkt and ?stationWkt, can be used in basic graph patterns and refer to spatial literals. Note that, a spatial variable is an object of a spatial predicate in the triple pattern. In this example, the spatial predicate is *geo:asWKT* defined in [19]. In addition to the basic graph patterns, the spatial variables are also used in the FILTER expression. Similar to the spatial predicate, the spatial built-in condition in FILTER expression is also assigned with a unique namespace. Current version of EAGLE supports several topological spatial relations such as *geo:sfWithin, geo:sfDisjoint,geo:sfIntersects, geo:sfContains*.

Example 3 (Spatial property function query). Given the latitude and longitude position, it retrieves the number of nearest weather stations that locate within 20 miles. The query is shown in Listing 6.8.

The above query demonstrates the usage of *geo:sfWithin* property function. When this property function is called, a dedicated piece of code will be executed to find all the geometries locate within an area. The area is specified by these arguments *(59.783 5.35 20 'miles')*. For each spatial object that satisfies the spatial condition, its IRI is bound to the *?stationGeo* variable that occurs in the triple representing the property function call. In addition to the default GeoSPARQL syntax of this function, we additionally extend its usage as follows:

> **GeoSPARQL syntax:** <*feature*$_1$> *geo:sfWithin* <*feature*$_2$>
>
> **Our extension:** <*feature*$_1$> *geo:sfWithin* (<*lat*> <*lon*> <*radius*> [<*units*> [<*limit*>]])

Table 6.2 describes the list of spatial property functions that are currently supported in EAGLE. These functions allow the user to specify the query bounding box area by either using the <*geo*> parameter or using the concrete coordinates via <*lat*>, <*lon*>, <*latMin*>, etc. The <*geo*> parameter can be a spatial variable or a spatial RDF literal. Similarly, the <*units*> can be an unit URI or a string value. The supported distance units are presented in Table 6.3. Finally, the <*limit*> parameter is to limit the number of results returned by the function.

Table 6.2: EAGLE's spatial property functions

Spatial functions	Description
<feature> **geo:sfIntersects** (<geo> | <latMin> <lonMin> <latMax> <lonMax> [<limit>])	Find features that intersect the provided box, up to the limit.
<feature> **geo:sfDisjoint** (<geo> | <latMin> <lonMin> <latMax> <lonMax> [<limit>])	Find features that intersect the provided box, up to the limit.
<feature> **geo:sfWithin** (<geo> | <lat> <lon> <radius> [<units> [<limit>]])	Find features that are within radius of the distance units, up to the limit.
<feature> **geo:sfContains** <geo> | <latMin> <lonMin> <latMax> <lonMax> [<limit>])	Find features that contains the provided box, up to the limit.

```
SELECT ?obs ?value ?time
WHERE {
    ?sensor sosa:isHostedBy got-res:gu9gdbbysm_ish_1001099999.
    ?sensor sosa:observes got:AirTemperatureProperty.
    ?obs sosa:madebySensor  ?sensor;
            sosa:resultTime ?time;
            sosa:hasSimpleResult ?value.
    ?value temporal:values ("10/03/2018"^^xsd:dateTime "15/03/2018"^^xsd:dateTime)
}
```

Listing 6.9: Temporal property function query.

Example 4 (Temporal property function query). Return the list of air temperature observation values that are generated by the station <*got-res:WeatherStation/gu9gdbbysm_ish_1001099999*> from 10th March 2018 to 15th March 2018. The query is shown in Listing 6.9.

The query demonstrates the example usage of one of our temporal property functions, called *temporal:values*. In this query, the property function *temporal:values* is called to retrieve all the temperature observation values that are generated within a specific time interval. Recall that, the prefix <*temporal:*> is to represent the temporal property function. Table 6.4 lists all the supported temporal property functions and their syntax. The usages of these functions will be demonstrated in the following examples.

Table 6.3: Supported units

URI	Description
units:kilometre or units:kilometer	Kilometres
units:metre or units:meter	Metres
units:mile or units:statuteMile	Miles
units:degree	Degrees
units:radian	Radians

Table 6.4: EAGLE's temporal property functions

Temporal functions	Description
?value **temporal:sum** (<startTime> <endTime> [<'groupin' down sampling function> <geohash prefix> <observableProperty>])	Calculates the sum of all reading data points from all of the time series or within the time span if down sampling.
?value **temporal:avg** (<startTime> <endTime> [<'groupin' down sampling function> <geohash prefix> <observableProperty>])	Calculates the average of all observation values across the time span or across multiple time series
?value **temporal:min** (<startTime> <endTime> [<'groupin' down sampling function> <geohash prefix> <observableProperty>])	Returns the smallest observation value from all of the time series or within the time span
?value **temporal:max** (<startTime> <endTime> [<'groupin' down sampling function> <geohash prefix> <observableProperty>])	Returns the largest observation value from all of the time series or within a time span
?value **temporal:values** (<startTime> <endTime> [<'groupin' down sampling function> <geohash prefix> <observableProperty>])	List all observation values from all of the time series or within the time span

Example 5 (Analytical spatio-temporal query). Detect all wind-speed observation in an area within 40 miles from the center of Ohio City during the time from 10th Jan 2017 to 10th Feb 2017. The Ohio City center coordinate is (40.417287 -82.907123). The query is shown in Listing 6.10.

```
SELECT ?weatherStation ?time ?value
WHERE
{
    ?weatherStation geo:hasGeometry ?stationGeo.
    ?stationGeo geo:sfWithin (40.417287 -82.907123 40 'miles').
    ?sensor sosa:isHostedBy ?weatherStation.
    ?sensor sosa:observes got:WindSpeedProperty.
    ?obs sosa:madebySensor  ?sensor;
            sosa:resultTime ?time;
            sosa:hasSimpleResult ?value.
    ?value temporal:avg ("10/01/2017"^^xsd:dateTime "10/02/2017"^^xsd:dateTime)
}
```

Listing 6.10: Analytical spatio–temporal query.

The query above demonstrates the mix of spatial and temporal property functions. The query use the spatial function, namely *geo:sfWithin*, to filter all weather stations that locate in the area *(40.417287 -82.907123 40 'miles')*. Additionally, it also retrieves the list of wind speed observation values generated by these station with the time constraint.

Example 6 (Analytical spatio-temporal query). Calculate the daily average windspeed at all weather stations that locate within 20 miles from London city center during the time from 10th March 2018 to 15th March 2018. The query is shown in Listing 6.11.

```
SELECT ?weatherStation ?time ?value
WHERE
{
    ?city dbo:type dbr:Capital;
          a dbo:Place;
          foaf:name "London";
          geo:hasGeometry ?cityGeo.
    ?cityGeo geo:lat ?lat; geo:long ?long.
    ?weatherStation geo:hasGeometry ?stationGeo.
    ?stationGeo geo:sfWithin (?lat ?long 20 'miles').

    ?sensor sosa:isHostedBy ?weatherStation.
    ?sensor sosa:observes got:WindSpeedProperty.
    ?obs sosa:madebySensor  ?sensor;
        sosa:resultTime ?time;
        sosa:hasSimpleResult ?value.
    ?value temporal:avg ("10/03/2018"^^xsd:dateTime "15/03/2018"^^xsd:dateTime
                              'groupin' '1d-avg')
}
```

Listing 6.11: Analytical spatio–temporal query.

The query demonstrates a complex analytical spatio-temporal query. In this query, we first retrieve the London geometry data (i.e, latitude and longitude) by querying the DBPedia dataset. After that, we use the spatial function, namely *geo:sfWithin*, to query all the stations that locate within 20 miles from London. In the temporal property function used in this query, we demonstrate the usage of the *downsampler* feature indicated by *groupin* keyword, and the *downsampling*

aggregation function. Given a brief description, the *downsampler* feature is our additional temporal query feature which aims to simplify the data aggregation process and to reduce the resolution of data. The data aggregation and data resolution are specified by the downsampling aggregation function, which is formed by <*time interval*>_<*aggregation function*>. The <*time interval*> is specified in the format <*size*><*units*> such as *1h* or *30m*. The aggregation function is taken from the list (*sum, average, count, min, max*). For example, as illustrated in the query, the downsampling aggregation function is *1d-avg*.

An example usage of downsampler can be described as follows. Lets say a wind-speed sensor is feeding observation data every second. If a user queries for data over an hour time span, she would receive 3,600 observation data point, something that could be graphed fairly easily in the result table. However, lets consider the case that the user asks for a full week of data. For that, she will receive 604,800 records, thus, leading to a very big result table. Using a downsampler, multiple data points within a time range for a single time series are aggregated together with an aggregation function into a single value at an aligned timestamp. In this way, the number of return values can be reduced significantly.

Example 7 (Analytical spatio-temporal query). Retrieve the weekly average temperature of area B which has geohash *"uoq"* in March 2018. This query illustrates the usage of two optional arguments in the temporal property functions, namely *geohash* and *observableProperty*. The query is shown in Listing 6.12.

```
 SELECT ?v
{ ?v temporal:avg ("01/03/2018"^^xsd:dateTime "31/03/2018"^^xsd:dateTime
                           'groupin' '1w-avg' 'u0q' got:AirTemperature). }
```

Listing 6.12: Analytical spatio–temporal query.

Example 8 (Full-text search query). Retrieve the total number of observation for each observed property of places that match a given keyword *'Cali'*. The query is shown in Listing 6.13.

```
SELECT ?place ?observedType  (count(?obs) as ?totalNumber)
WHERE
{
    ?station a got:WeatherStation.
    ?station geo:hasGeometry ?geoFeature.
    ?geoFeature text:match (geoname:parentCountry 'Cali*').
    ?geoFeature geoname:parentCountry ?place.
    ?sensor sosa:isHostedBy ?station;
        sosa:observes ?observedType.
    ?obs sosa:madebySensor  ?sensor.
}GROUP BY ?place ?observedType
```

Listing 6.13: Full-text search query.

The above query demonstrates the usage of full-text search feature via the *text:match* property function. The *text:match* syntax is described as follows:

<*subject*> *text:match* (<*property*> *'query string'* <*limit*>)

In the *text:match* function syntax, the <*subject*> implies the subject of the indexed RDF triple. It can be a variable, e.g, ?s, or an IRI. The <*property*> is an IRI of which the literal is indexed, e.g, *rdfs:label*, *geoname:parentCountry*. The *'query string'* is the query string fragment following the Lucence syntax[6]. For example, the parameter *'Cali*'* is to select all the literals that match prefix "Cali". The optional *limit* limits the number of literals returned. Note that, it is different than the number of total results the query will return. When a LIMIT is specified in the SPARQL query, it does not affect the full-text search, rather, it only restricts the size of the result set.

6.5 EXPERIMENTAL EVALUATION

In this section, we present a rigorous quantitative experimental evaluation of our EAGLE implementation. We divide the presentation of our evaluation into different sections. Section 6.5.1 describe the experimental setups which include platform and software used, datasets and queries descriptions. Section 6.5.2 presents the experimental results. In this section, we compare the data loading throughput and query performance of EAGLE against Virtuoso, Apache Jena and GraphDB. We also discuss the performance difference of EAGLE when applying our data partitioning strategy described Section 6.2.3. Finally, we evaluate the EAGLE engine on a Google Cloud cluster to demonstrate its elasticity and scalability on data loading and query performance.

6.5.1 Experimental Settings

6.5.1.1 *Platform and Software*

To demonstrate the EAGLE's performance and scalability, we evaluate it on a physical setup and a cloud setup. It is worth mentioning that our physical setup is dedicated to a live deployment of our GraphOfThings application at `http://graphofthings.org` which has been ingesting and serving data from more than 400,000 sensor data sources since June 2014. We compare EAGLE's performance against Apache Jena v3.12, Virtuoso v7 and GraphDB v8.9 (former OWLIM store [102]). Among them, Jena represents the state-of-the-art in terms of a native RDF store, Virtuoso is the widely used RDF store backed by RDBMS, and GraphDB is a clustered RDF store that has recently supported spatial query.

We deployed Apache Jena and Virtuoso v7 on a single machine with the same configuration as in our physical setup below. For EAGLE, we installed ElasticSearch v7 and OpenTSDB v2.3 for both the physical and cloud setups. Similarly, we also installed the GraphDB v8.9 on all setups.

Physical setup. We deployed a physical cluster that consists of four servers running on the shared network backbone with 10 Gbps bandwidth. Each server has the following configuration: 2x E5-2609 V2 Intel Quad-Core Xeon 2.5GHz 10MB Cache, Hard Drive 3x 2TB Enterprise Class SAS2 6Gb/s 7200RPM - 3.5" on RAID 0, Memory 32GB 1600MHz DDR3 ECC Reg w/Parity

6 https://lucene.apache.org/core/2_9_4/queryparsersyntax.html

DIMM Dual Rank. One server is dedicated as a front-end server and to coordinating the cluster, other 3 servers are used to store data and run as processing slaves.

Cloud setup. The cloud setup was used to evaluate the elasticity and scalability of the EAGLE engine. We deployed a virtual cluster on Google Cloud. The configuration of the Google Cloud instances we use for all experiments is "n1-standard-2" instance, i.e, 7.5 GB RAM, one virtual core with two Cloud Compute Units, 100 GB instance storage, Intel Ivy Bridge platform. In this evaluation, we focused more on showing how the system performance scales when increasing the number of processing nodes, rather than serving as a comparison of its performance with the physical cluster.

6.5.1.2 *Datasets*

For our experimental evaluation, we again reuse our linked meteorological dataset presented in Chapter 4. The dataset consists of more than 20K meteorological station allocated around the world and covers various aspects of data distribution. The window of archived data is spread over 10 years, from 2008 to 2018. It has more than 3.7 billion sensor observation records which are represented in the SSN/SOSA observation triple layout (7 triples/records). Hence, the data contains approximately 26 billion triples if it is stored in a native RDF store.

Additionally, in order to give a more practical overview of the engine, we evaluated it on even more realistic datasets, especially ones consisting of both spatial and text data. To meet such requirements, we select several datasets from GoT data sources [146]. In particular, to evaluate the spatial data loading throughput, in addition to the sensor station location, we also import the transportation dataset which contains 360 million spatial records. These records were collected from 317,000 flights and 20,000 ships during the time 2015-2016. Similarly, for the full-text data loading evaluation, we import a Twitter dataset that consists of five million tweets. The detailed statistics of all the datasets used for our evaluations are listed in Table 6.5.

Table 6.5: Dataset

Sources	Sensing objects	Historical Data	Archived window
Meteorological	26k	3.7B	since 2008
Flight	317k	317M	2014 - 2015
Ship	20k	51M	2015 - 2016
Twitter	–	5M	2014 -2015

6.5.1.3 *Queries*

The queries we used for our experiments are the same as the ones in Table 5.5 Chapter 5.2. Their SPARQL representation are described in Appendix A. Note that, these queries are selected in a way that they cover many operators with different levels of complexity, for instance, spatial filter, the mix of a spatial and temporal filter, full-text search and aggregations.

6.5.2 Experimental Results

6.5.2.1 *Data Loading Performance*

We evaluated the EAGLE's performance with respect to data loading throughput on our physical setup and compared it to the state-of-the-art systems. Benchmark data were stored in files and imported via bulk loading. Unlike the general performance comparisons that only focus on triple data loading performance, we measure separately the loading performance of spatial, text and temporal data. The loading speed was calculated via the number of objects that can be indexed per second, instead of the number of triples. This evaluation helped us to have a better understanding of the indexing behavior of the test engines for specific types of data such as geospatial and text.

Spatial data loading performance with respect to dataset size

Figure 6.13 depicts the average spatial data loading speed of the four evaluated RDF systems, with respect to various dataset sizes. The data loading time is shown in Figure 6.14. Overall, the results reveal that the increase in the data size can significantly affect the loading performance of all systems. Among them, Apache Jena has the worst performance. The average data loading speed is below 10,000 obj/sec for all dataset sizes, slower than other systems. Moreover, it takes almost two days (46.23 h) for loading 658 million spatial data. The data loading performance of EAGLE and GraphDB are very close, followed by Virtuoso. For example, EAGLE loads 658 million spatial objects in 7.74 h. Its average throughput is 23,620 obj/sec. In the meantime, GraphDB is one hour behind, resulting in 8.72 h and the average speed is 20,960 obj/sec. Virtuoso achieves the speed at 17,500 obj/sec. The slower insert speed of Virtuoso and Jena can be explained by the limit of the single data loading processes in these systems, which are deployed on a single machine. This sharply contrasts with the parallel data loading processes supported by the distributed back-end DBMS in EAGLE (ElasticSearch and OpenTSDB) and GraphDB.

We also learned that in the beginning, EAGLE performs slightly behind GraphDB in the case of loading a small dataset (<350 million). We hypothesize that this is due to several reasons such as load imbalance, increased I/O traffic and platform overheads in EAGLE. However, for loading larger dataset, this comparison result is reversed and the spatial data loading performance of GraphDB is slower than ours. This highlights the capabilities of our system for dealing with the "big data" nature of sensor data.

Text data loading performance with respect to dataset size

To evaluate the text data loading performance, we load the Twitter dataset that consists of 5 million tweets in the RDF format. The loading speed and loading time are reported in Figures 6.15 and 6.16, respectively. According to the results, EAGLE outperforms than the other systems. We can see in Figure 6.15 that its loading speed is just lightly affected by the data size increase. The highest speed EAGLE can reach is 11,800 obj/sec for loading 0.64 millions of tweet. We attribute this to the outstanding performance of EAGLE's databases, namely

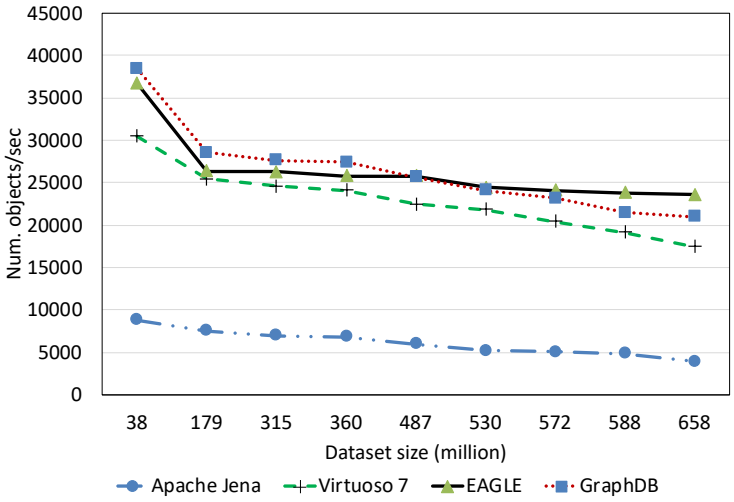


Figure 6.13: Average spatial data loading throughput

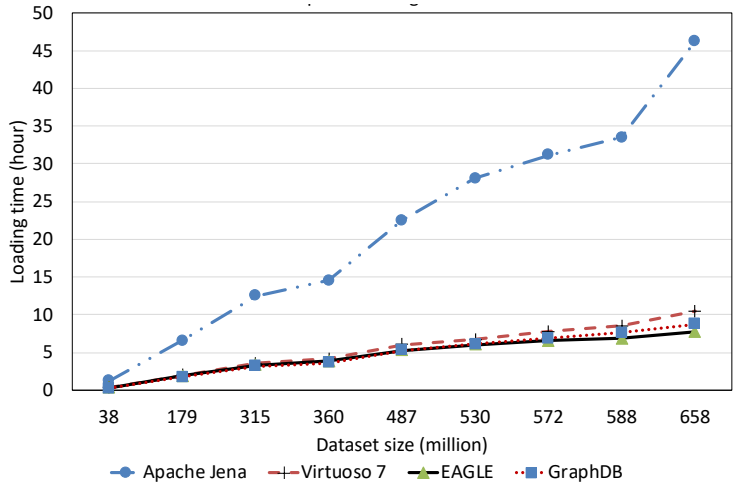


Figure 6.14: Spatial data loading time

ElasticSearch, which is originally document-oriented database. In the case of loading the same data size, GraphDB is slower than EAGLE. Its average speed is 9,700 obj/sec. Virtuoso and Jena follow at two and five times slower then EAGLE, respectively.

In comparison with the spatial data loading performance, the text data loading speed of EAGLE is much slower. This is reasonable because that in order to index the text data, the system needs to analyze the text and break it into a set of sub-strings. Consequently, this requires more computation and resource consumption, and hence, increases the overall loading time.

Temporal data loading performance with respect to dataset size

We evaluated the temporal data loading performance by importing our 10 years of historical linked meteorological data. In this evaluation, we also measured the performance of EAGLE when disabling the spatio-temporal partitioning feature, denoted by *EAGLE-NP*. Instead of

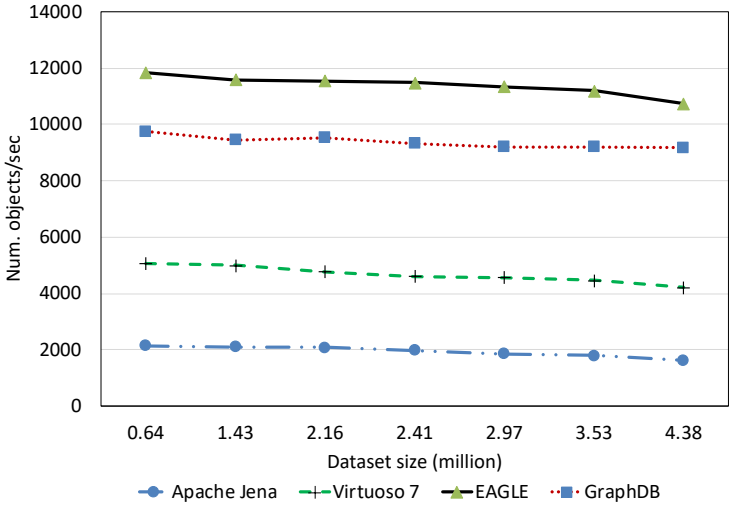


Figure 6.15: Average full-text indexing throughput

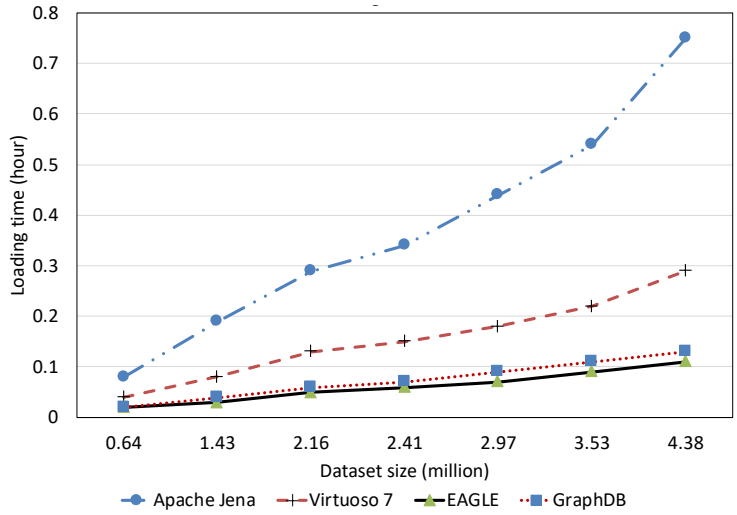


Figure 6.16: Text data loading time

loading the entire temporal dataset, we terminated the loading process at 7.78 billion triples due to the long data loading time and some of evaluated systems stop responding.

The results in Figures 6.17 and 6.18 draw our attention with regard to the performance of all systems when loading the small dataset. Regardless of the poor performance of Apache Jena, it is apparent that Virtuoso had better loading performance than EAGLE and GraphDB in the case of loaded data sizes under 100 million data point. A possible explanation for this phenomena is the communication latency of the distributed components in GraphDB and EAGLE. More precisely, in these distributed systems, the required time for loading data, plus the time for coordinating the cluster and the network latency are more than the data loading time in Virtuoso. Nevertheless, the difference is acceptable and we believe EAGLE is still applicable for interactive applications that only import a limited amount of data.

Another interesting finding is that our system performs differently if the spatio-temporal partitioning strategy is disabled. In this case, the highest insert speed that EAGLE-NP can achieve is 30,000 obj/sec. However, this speed drops dramatically with the growth of the imported data.

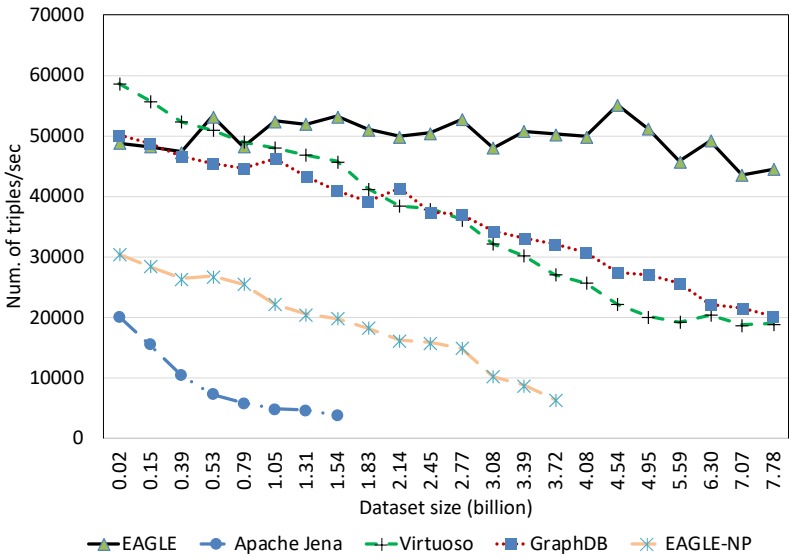


Figure 6.17: Average temporal indexing throughput

Moreover, we also observe that EAGLE-NP stops responding when the data size reaches to 3.72 billion records, as depicted in Figures 6.17 and 6.18. Looking at the system log files, we attribute this to a bottleneck in performance that happened with the OpenTSDB *tsdb* table. As previously explained in Section 6.2.3, if the spatio-temporal data partitioning is disabled, the *tsdb* table is not pre-split, thus, there is only one region of this table that is initialized. In this case, data are only inserted into this region. As a result, when the I/O disk writing speed cannot adapt to a large amount of fed data, the bottleneck phenomena happens.

The efficiency of EAGLE is demonstrated when applying our proposed spatio-temporal data partitioning strategy. It is even more explicit in the case of loading a large dataset. As evidenced in Figure 6.17, unlike the others, the average insert speed of EAGLE almost remains horizontal when the number of data instances increases. In particular, the highest speed that EAGLE can reach is 55,000 obj/sec and there is no significant difference when the number of data points rises from 0.02 to 7.78 billion. Moreover, for loading 7.78 billion temporal triples, EAGLE took only 48.51 h. However, in the same case, GraphDB and Virtuoso need 106.97 h and 113.71 h, respectively. The better rank of EAGLE is attributed to our data partitioning strategy in which we pre-split the *tsdb* table into multiple data regions in advance of the data loading operation. Because each region is assigned with a range of geohash prefixes, data that has different geohash prefixes managed by different regional servers can be inserted in parallel, resulting in a significant increase in terms of data loading performance. However, when the amount of data stored in a pre-split region reaches the given threshold capacity, the region will be re-split automatically. Together with the splitting process, all related data has to be transferred and distributed again. This step will cause additional cost and will affect the system performance. This explains the slight fluctuation of our system insert speed in Figure 6.17 during the data loading process.

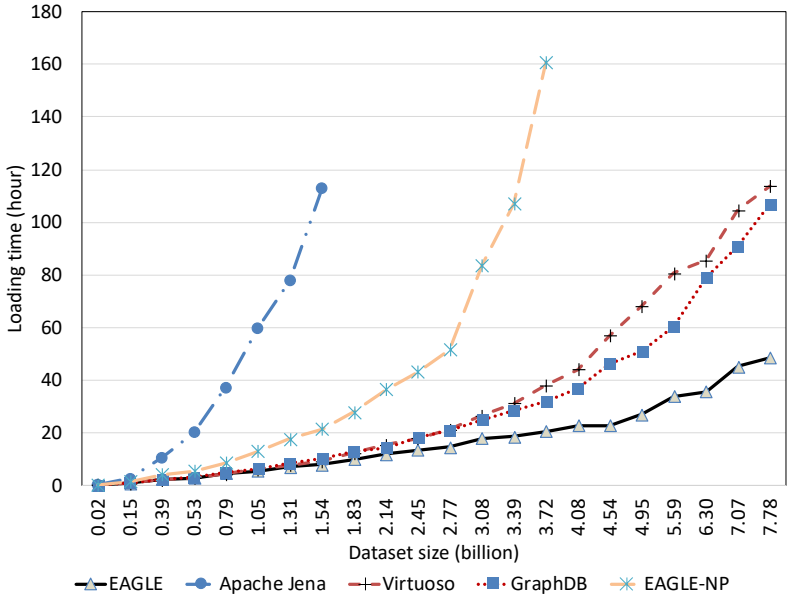


Figure 6.18: Temporal data loading time

6.5.2.2 *Query Performance with respect to Dataset Size*

This experiment is designed to demonstrate the query performance of all evaluated systems with respect to different data aspects and dataset size. In this experiment, for each query, we measure the average query execution time by varying the dataset imported. In order to give a more detailed view on the query performance, based on the query complexity, we group the test queries into several main categories: spatial query, temporal query, full-text search query, non-spatio-temporal query, and mixed query. These query categories are described in Table 6.6.

Table 6.6: Categorizing queries based on their complexity

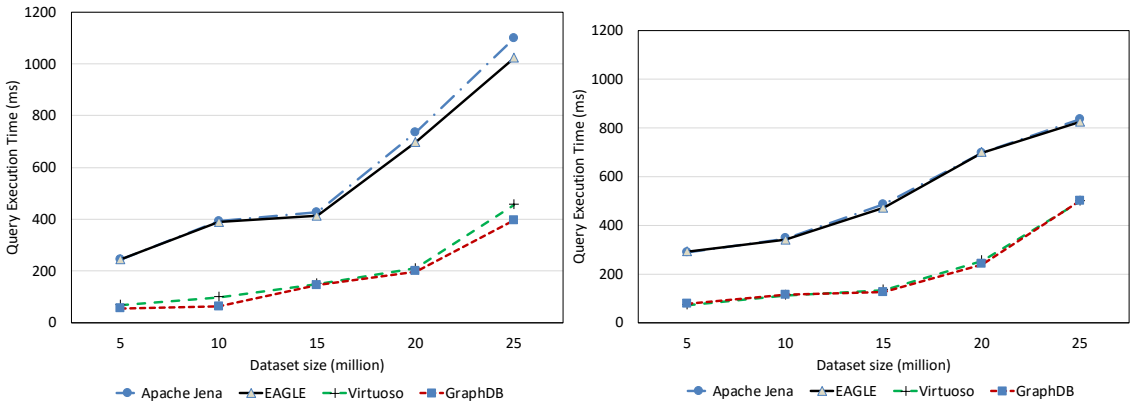
Category	non-spatio-temporal query	Spatial query	Temporal query	Full-text search query	Mixed query
Query	Q2,Q11	Q1	Q5, Q6	Q8, Q9	Q3,Q4,Q7,Q10

To conduct a precise performance comparison, we load different datasets that correspond to the query categories. For example, our spatial datasets are used to evaluate the spatial query performance while the sensor observation dataset is for queries that require a temporal filter. For the non-spatio-temporal queries, we use the static dataset that describes the sensor metadata. It is important to mention that our data partitioning approach is only applied for temporal data stored in OpenTSDB and does not explicitly affect the spatial and full-text search query performance in ElasticSearch. Therefore, in the experiments for spatial and full-text search query performance, the performances of EAGLE and EAGLE-NP are not differentiated.

Non-spatio-temporal queries performance

We first evaluated the performance of non-spatio-temporal queries, which were Q2 and Q11. These were the standard SPARQL queries which only query on the semantic aspect of sensor data and have neither spatio-temporal computation nor full-text search. The average query execution times are plotted in Figure 6.19. The results demonstrate the close performance

of EAGLE and Apache Jena in regard to non-spatio-temporal queries. This is explained by the use of similar processing engine. In fact, the SPARQL query processing components in EAGLE are extended from Apache Jena ARQ with some modifications. Meantime, Virtuoso and GraphDB prove their reputations in SPARQL query performance by being faster then EAGLE. However, the difference is still acceptable, in the order of *ms*.



(a) Q2 execution time

(b) Q11 execution time

Figure 6.19: non-spatio-temporal query execution time with respect to dataset size

Spatial queries performance

Figure 6.20 depicts the query execution time of the spatial query, which is represented by Q1, with respect to varying spatial dataset size. According to the evaluation result, we find that Apache Jena performs poorly. Its spatial query performance linearly increases with increments of the loaded data. On the contrary, Virtuoso, GraphDB and EAGLE perform closely and are weakly influenced by the data size. GraphDB is recognised as having good performance, followed by Virtuoso. Compared to these systems, EAGLE is slightly slower, only in the order of *ms*. A possible reason could be the overhead of the join operation between the BGP matching and the spatial filter results. Note that, parallel join operations are not yet supported in EAGLE and have to be performed locally in a single thread.

Full-text search queries performance

In the following, we discuss the performance of the full-text search queries (Q8, Q9) for the test systems. The evaluation results are reported in Figure 6.21. Despite the impressive query execution time of GraphDB and Virtuoso, which are generally less than 500 *ms* for both Q8 and Q9, EAGLE is still slightly faster. This is again thanks to the outstanding performance of ElasticSearch on full-text search queries. Note that, although Apache Jena, GraphDB and ElasticSearch support full-text search through the use of Lucene, ElasticSearch is notable for having a better optimization.

Temporal queries performance

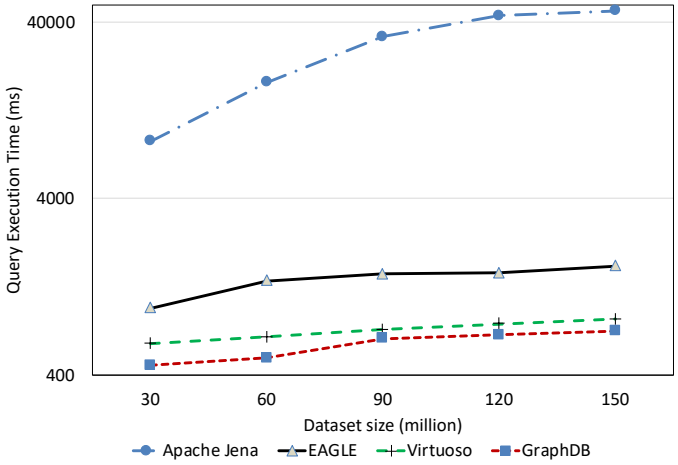
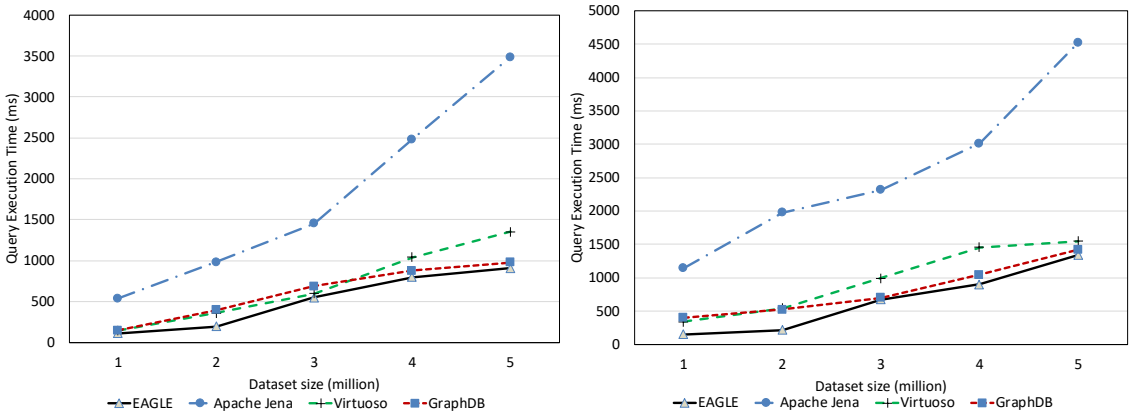


Figure 6.20: Q1 execution time with respect to spatial dataset size (in logscale)



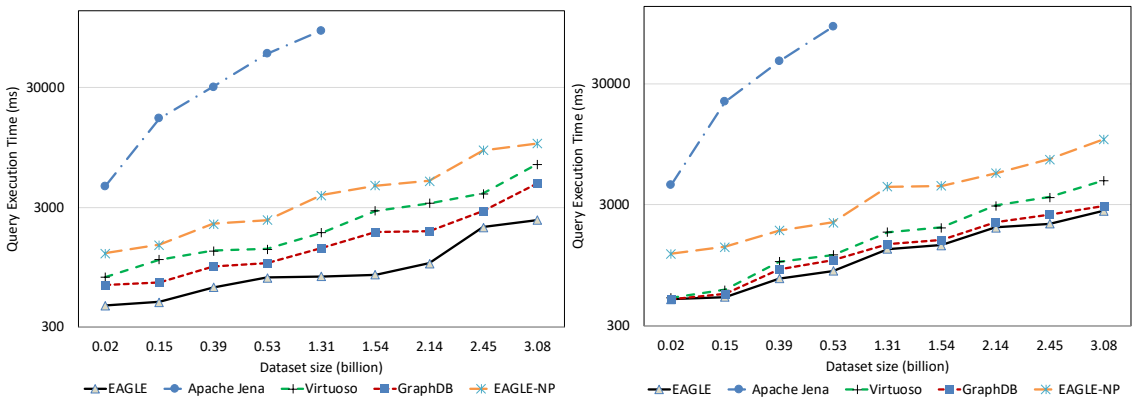
(a) Q8 execution time

(b) Q9 execution time

Figure 6.21: Text-search queries execution time with respect to dataset size

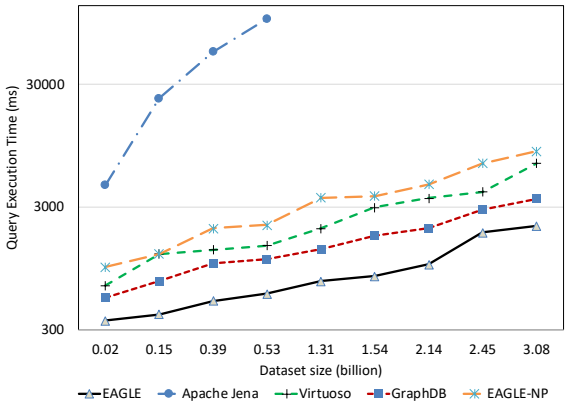
The temporal query performance is evaluated over our historical meteorological observation data. Figure 6.22 presents the query execution time of all systems with respect to observation dataset size. It is apparent that query performance is affected by an increase in the amount of data. For Apache Jena, along with a linear increase in the query execution time, we also notice that it is only able to run up to a certain amount of data. As we can see in the figure, it could ideally execute queries with datasets under 0.5 billion data points. However, when executing these queries on a dataset which is over 1.31 billion data points, Apache Jena stops responding. The performances of EAGLE and EAGLE-NP are significantly different. For example, when executing over a dataset of 0.53 billions data observations, if the spatio-temporal data partitioning strategy is not applied, the average execution time of Q6 in EAGLE-NP is 2147 (ms). Meanwhile, if the data partitioning strategy is enabled, EAGLE takes only 589 (ms) to execute the same query, resulting in it running four times faster. Furthermore, we also learn the better performance of the EAGLE system in comparison with Virtuoso and GraphDB. The explanations for this performance could be: (1) the effectiveness of our data partitioning strategy so that the engine can quickly locate the required data partition and then organize the scans for a large number of data rows; (2) the power of OpenTSDB query functions that we rely on,

especially for data aggregation.



(a) Q3 execution time

(b) Q5 execution time



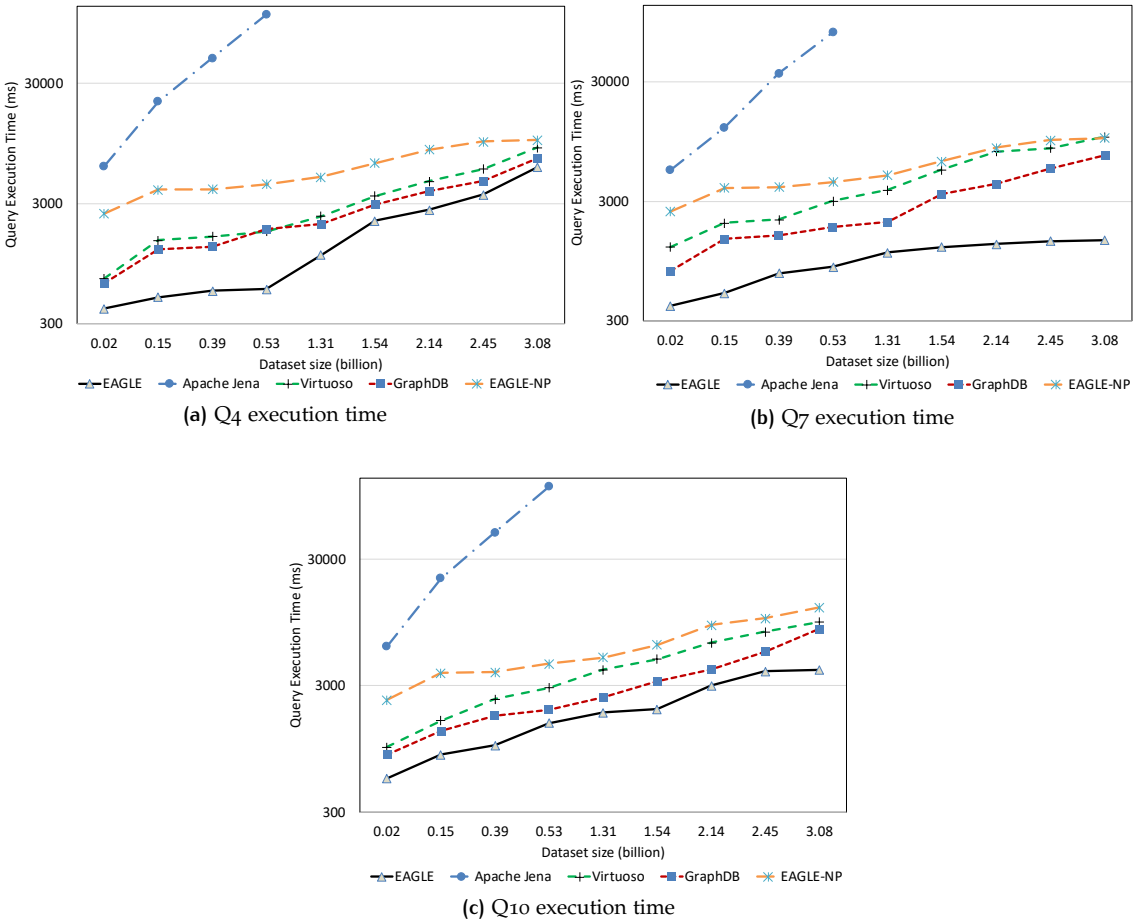
(c) Q6 execution time

Figure 6.22: Temporal queries execution time with respect to dataset size (in logscale)

Mixed queries performance

Another aspect to be considered is the performance of the analytics-based queries that require the mixing of spatial, temporal computations or full-text search (Q4, Q7, Q10). For these queries, we increase the query timeout to 120 (sec) due to their high complexities. The evaluation results are shown in Figure 6.23. Apache Jena reveals undergoes timed outs for all queries when the loaded data size is over 0.5 billion. Another fact that can be clearly observed is that EAGLE is orders of magnitude faster than the others. This is demonstrated by the case of Q7. Note that, this query implies a heavy computation on both spatial and temporal data. Additionally, it also requires that the results have to be ordered by time. As shown in Figure 6.23b, for executing query over the dataset with 3.08 billion records, EAGLE performs Q7 much better (1,420 ms), follows by GraphDB (7,289 ms) and Virtuoso (10,455 ms). There are can be several reasons for our impressive performance: (1) The effectiveness of the OpenTSDB time-series data structure such that data are already sorted by time during the loading process. Consequently, in EAGLE system, for the query that has an order-by operator on date-time, the ordering operation cost, in this case, is eliminated. (2) The second reason again sheds light on

the success of our data partitioning strategy and our row-key design so that the time cost for locating the required data partition and the data scan operation is significantly minimized.



(a) Q4 execution time

(b) Q7 execution time

(c) Q10 execution time

Figure 6.23: Mixed queries execution time with respect to dataset size (in logscale)

6.5.2.3 *Query Performance with respect to Number of Clients*

This experiment is designed to test the concurrent processing capability of EAGLE in a scenario where the system has to deal with a high volume of queries which are sent from multiple users. Rather than serving as a comparison with other stores, this experiment only focuses on analyzing the EAGLE's query processing behavior when receiving concurrent queries from multiple clients. This experiment is performed as follows. In the first step, a dedicated script is built to randomly select and send queries to the system. The query parameters are also randomly generated. In the second step, we perform measurement runs with 10, 100, 250, 500 and 1000 clients concurrently. Finally, for each query, the query execution times are summarized to compute the average value.

Figure 6.24 reports the evaluation results. In general, the execution time for all queries linearly rises when more clients are added. It can be clearly observed that, when the number of clients increases from 250 to 1000, the query execution time increases dramatically. Firstly, this is due to the growing workload applied to the system. Secondly, by deeply analyzing the query cost breakdown, another possible reason is the inefficiency of our query plan cache mechanism. According to our observation, the query cache only works for the non-spatio-temporal queries.

However, for the duplicated queries that share the same spatial, temporal and full-text filters, instead of reusing the cached query plan, the query optimizer has to re-generate a new query execution plan. For example, if there are 100 query instances of Q4 that have been sent from 100 clients, the query optimizer has to re-generate the query execution plan for Q4 for 100 times. Obviously, this leads to a dramatic increase for the total query execution time of Q4. As previously mentioned, the EAGLE's query processing engine has been implemented by extending the widely-known query engine, namely Jena ARQ, thus, its query cache is identical to the one in Jena ARQ. Unfortunately, the original one was only developed for standard SPARQL query and does not work for the spatio-temporal query. Moreover, we also learned that the Jena ARQ's query cache does not work correctly with the queries that share similar query patterns but different literals. This is also the case for our tested query patterns, of which the literals are randomly generated. We address this issue by proposing a novel learning approach for spatio-temporal query planning, that is described in Chapter 7.

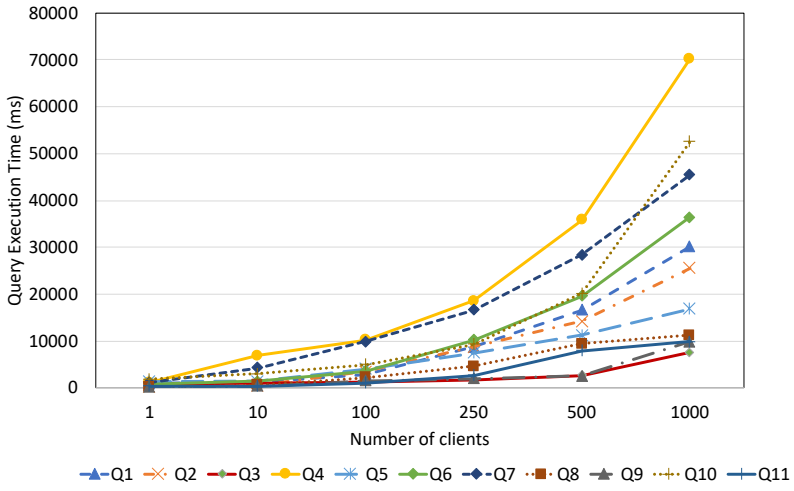


Figure 6.24: Average query execution time with respect to number of clients

6.5.2.4 *System Scalability*

In this experiment, we measure how the EAGLE's performance scales when adding more nodes to the cluster. We vary the number of nodes in the Google Cloud cluster with 2, 4, 8, 12 nodes, respectively.

Figure 6.25 presents the average loading throughput of spatial, temporal and text data when increasing the number of nodes. The results reveal that the index performance linearly increases with the size of the cluster. This is because scaling out of the cluster causes the working data that needs to be indexed on each machine to be small enough to fit into main memory, which dramatically reduces the required disk I/O operations.

In the following, we look at the query performance evaluation results shown in Figure 6.26. According to the results, the query execution times of Q2 and Q11 remain steady and are not affected by the cluster size. This is due to the fact that these queries are non-spatio-temporal queries and only query on the static dataset. Recall that, we store the static dataset on centralized storage (Apache Jena TDB), which is not scalable and is hosted on a single machine. Queries on static data are only executed on this machine. Therefore, it is understandable that, for the

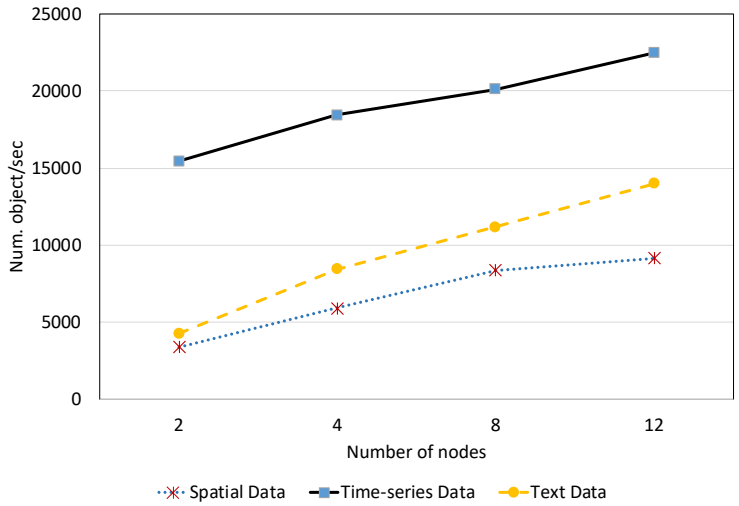


Figure 6.25: Average index throughput by varying number of cluster nodes

non-spatio-temporal queries being executed over the same dataset, the scaling out of the cluster has no effect on their performance.

Unlike Q2 and Q11, the performance of other queries scales perfectly with the cluster size. The results indicate that the EAGLE engine has a considerable decrease in query execution time for mixed queries (Q4, Q7, Q10). Meanwhile, other queries have a slightly decreased on query execution time. A representative example of mixed queries to demonstrate the scalability of EAGLE is Q4. This query required a heavy spatio-temporal computation on a large number of historical observation data items for a given year. However, along with the scaling out of the cluster, the amount of data processing for this query on each node was also reduced significantly. This explains the rapid drop in the query execution time from 1120 (ms) to 514 (ms) for Q4 when the cluster size scales out from 2 to 12 nodes, respectively.

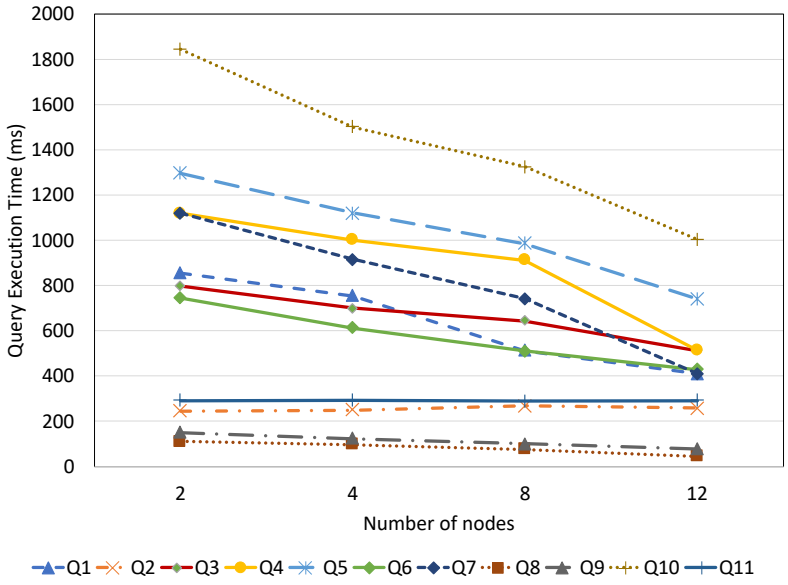


Figure 6.26: Average query execution time by varying number of cluster nodes

6.6 SUMMARY

In this chapter, we have introduced EAGLE, a scalable query processing engine for Linked Sensor Data, which is able to index, filter and aggregate a massive amount of sensor data. We have also presented an architecture of EAGLE that follows a loosely coupled hybrid architecture consisting of different databases for managing different aspects of sensor data. A detailed implementation of our approach has been also given. Moreover, to support spatio-temporal queires, we have extended the SPARQL query language by providing a set of spatio-temporal and full-text search query functions.

We have presented an extensive quantitative evaluation of the EAGLE engine implementation and conducted a comparison with a set of well-known RDF stores running on a single node as well as a cluster RDF store. To conduct a precise performance comparison, we measure separately the loading performance of spatial, text and temporal data. The experimental results show that the EAGLE engine performs better than other tested systems in terms of spatio-temporal and text data loading performance. For query performance, we have found that EAGLE is highly efficient for queries that require heavy spatio-temporal computations on a large amount of historical data. However, it is slightly behind Virtuoso and GraphDB for non-spatio-temporal queries. This is understandable as improving query performance on semantic data is not our main target.

Another fact that should be highlighted is the effectiveness of our spatio-temporal partitioning strategy and OpenTSDB rowkey scheme. This is evidenced by the evaluation results so that the EAGLE engine has outstanding performance when using the spatio-temporal partitioning strategy. If this data partitioning strategy is disabled, the engine has poor performance and stops responding at a certain dataset size.

For the scalability test, the evaluation result indicates that EAGLE scales perfectly with the cluster size. However, it also shows query planning issues with respect to multiple concurrent queries. Moreover, during the time we performed the experiments, we also observed that the query planning also worked inaccurately if there were any changes in the underlying environment such as removing or adding more nodes to a cluster, network latency during the busy hours, etc. We attribute this to the poor adaptability of the query cost model to changes in the distributed environment. Therefore, in the next chapter, we will propose our solution to address these query planning issues by applying machine learning techniques to quickly predict a query plan for a given query, so as to achieve even further better performance of EAGLE.

7 ADAPTIVE QUERY PLANNING WITH LEARNING

The evaluation of query performance in Chapter 6 indicates several issues that still existed in our EAGLE's implementation. The first issue is the poor processing capability of EAGLE when dealing with a high volume of concurrent queries. In addition to the growing workload applied to the system, we attribute this to the inefficiencies in EAGLE's query optimization so that it takes more time for generating a query execution plan. Moreover, we also observed that the query cache mechanism only works with standard SPARQL queries and works inaccurately with the spatio-temporal queries. In particular, for the concurrent duplicated spatio-temporal queries, instead of caching a query planning for later use, a new query plan is generated. This processing behavior consequently leads to an increment in the query processing time. Another problem is a lack of spatio-temporal statistics of sensor data that are used to estimate query processing cost in the optimization cost model. Due to the complexity and high-frequency updates of sensor data, statistics information is expensive to generate and maintain. To address these issues, in this chapter, we propose a learning approach for query planning on Linked Sensor Data that applies machine learning technique to predict the query planning for a given query based on the historical query execution plans.

This chapter is organized as follows. In section 7.1, we first introduce a motivation for our learning approach for query planning. Next, we review the existing works on query planning for Linked Data processing in Section 7.2. Section 7.3 describes our first approach to modeling and using learning techniques for our spatio-temporal SPARQL query. An experimental evaluation of our proposed learning approach follows in Section 7.5. Discussion and future work are also given in this section. Finally, we conclude our work in the last section.

This work is published in [155].

7.1 INTRODUCTION

Researchers in the Semantic Web community have proposed a substantial number of works that focus on query optimization in conjunction with supporting spatio-temporal computation. In many of these approaches, for a given query, the query optimizer will firstly analyze the correlation of the spatial, temporal and semantic aspects, and find the best execution plan based on either the complex cost model or by using data statistics and heuristics. However, these approaches have only proven to be effective in a centralized system where the optimizer can easily estimate the cost of all spatio-temporal query execution operations. In the context of a seman-

tic sensor data management system, which requires a distributed integration spatio-temporal query processing engine, the existing query optimization will not be sufficient because of the following reasons: (1) defining a cost model which has to express all spatio-temporal aspects of sensor data and is adaptable to changes in the underlying environment (data communication costs, characteristics of the network, cluster configuration, etc.) is a costly task and has not been fully addressed; (2) the spatio-temporal statistics of sensor datasets are often missing due to the (multidimensional) complexity and high-frequency updates of the data. They are expensive to generate and maintain.

An alternative solution to the above problem is a learning optimizer. The idea of this approach is to reuse existing execution plans to execute similar given queries. In this chapter, where we draw upon the ideas in [81], we propose a prediction framework that uses similarity identification between spatio-temporal SPARQL queries as well as machine learning techniques to predict a suitable query execution plan and query execution time for a new query. Specifically, the framework can accurately predict an execution plan that consists of the execution order of all the physical spatio-temporal operators. Moreover, we also aim to predict the query execution time to help the query engine have a better understanding about the query behavior prior to query execution. Accurately predicting the query execution time enables us to optimize our costly EAGLE's cloud infrastructure by rescheduling or preventing long-run queries that might cause too much resource contention or that will not be completed by a particular deadline. In summary, our main contributions are as follows:

1. A framework to predict the best query execution plan for an unforeseen spatio-temporal query

2. A technique to predict query execution times, which enables long-run and resource-consuming queries to be rescheduled or not run at all.

In the following section, we will review some existing works on query planning for Linked Data processing.

7.2 QUERY PLANNING FOR LINKED DATA PROCESSING

Lies at the heart of the query processing engine is the query optimizer. It aims to help the engine to determine an efficient query plan for executing a given query in a timely manner. In general, SPARQL query optimization can be distinguished into two broad categories: SQL-based and RDF native ones [192]. The SQL-based optimizers are for the RDF stores which are backed by RDBMS. In these, the RDF data are usually stored in relational tables and hence, rely on the optimization techniques of the underlying DBMS to efficiently evaluate SPARQL queries. It is worth mentioning that, query optimization has been extensively studied for many decades in the relational database literature [171, 74, 62, 63, 129, 151, 64]. For that reason, within the scope of this thesis, we only analyze the RDF native optimization. In this section, we will first review some existing SPARQL query optimization techniques. We then discuss a alternative optimization approach which applies machine learning techniques.

7.2.1 SPARQL Query Optimization

The current generation of SPARQL query optimization is typically based on data heuristics and statistics for selecting efficient query execution plans [186]. These two approaches are described as follows.

7.2.1.1 *Statistics-based Query Optimization*

The statistics-based approaches use the summary of the dataset for query cost estimation. RDF3X optimizer [131, 132, 134] is an example of using this approach. In here, the statistics information help to specify the join order of the query triple patterns, which can have a dramatic performance impact on query execution [131]. On the one hand, the approach firstly uses the dynamic programming [171] for iterating over all possible query execution plans. On the second hand, it relies on a cost model to estimate the number of intermediate results of the join operator in each plan. The intermediate results are calculated based on the statistics over the underlying data set. This information is used to decide the join order and algorithms to be used for query execution. Below is an example of RDF-3X cost functions for merge and hash joins (*MJ* and *HJ*, respectively).

$$\text{cost}_{\text{MJ}} = \frac{lc + rc}{100}, \text{cost}_{\text{HJ}} = 300,000 + \frac{lc}{100} + \frac{rc}{10}$$

where *lc* and *rc* are the cardinalities of the left and right inputs of the join operation (with $lc < rc$).

Along the line of statistics-based join ordering optimization in conjunction with dynamic programming, [125] and [47] propose the bottom-up and tom-down optimization approaches, starting with the simplest sub-queries or the entire query, respectively. However, the author in [61] mentioned that these approaches are only feasible for the queries that have up to 10-20 joins. Moreover, because of using a dynamic programming algorithm to produce logical plans, they need to accept the risk of exploitation of plan spaces and thus long optimization time for complex queries.

In [186], the author proposes a hybrid approach that uses both heuristics and statistics information. The optimizer introduced in this approach consists of three main sub-components. The first component is the BGP abstraction which is responsible to abstract a given SPARQL query as a directed acyclic graph. In this graph, each triple pattern is considered as a node and the join between them represented as an edge of the graph. The second component is the core optimization algorithm which uses minimum selectivity approach to generate the query execution plan. It should be noticed that, in this approach, the selectivity of a triple pattern is defined as the fraction of tuples that match the condition for each variable in the pattern. To generate an execution plan, the optimization algorithm firstly selects an edge with minimum selectivity. Next, the nodes correspond to this edge are marked as visited. Finally, these nodes are added to the execution plan and are ordered according to their estimated selectivities. The third component of the optimizer is responsible to help the optimization algorithm in selectivity estimation by using heuristics and statistics. Heuristics without pre-computed statistics and heuristics with pre-computed statistics are presented for triple patterns selectivity and joined triple patterns se-

lectivity. The evaluation shows that the algorithm enhances the performance for simple queries reduces the number of execution plans for static queries.

Similar to [186], [118] pre-compute the frequent paths (i.e., frequently occurring sequences of *S, P, O* labels) in the RDF data graph. In this approach, the statistics are kept for a small number of frequent/beneficial paths by using graph-mining techniques. This approach has been shown to be useful for selectivity estimation of star-joins and join chains.

In summary, the statistics-based query optimization techniques have been proved to be efficient if the statistics of underling data set are available. Besides, to avoid the exploitation of the query plan search spaces, these approaches are recommended to be used with queries that have small conjunctive patterns. According to [192], statistics-based optimization approaches also have two major drawbacks: (1) The statistics (e.g, histograms) about the underlying data set, that needed for cost-based optimization, are rarely available. In Linked Sensor Data context, this limitation is worse due to the high-frequency update rate and "big data" nature of sensor data. For that, the task of generating and maintain statistics data becomes more expensive. (2) Due to the graph-based data model and schema-less nature of RDF data, it is still not clear on what to create effective statistics for query cost estimation. To address these challenges, heuristics-based query optimization approaches are introduced. These heuristics-based optimization techniques are presented in the following section.

7.2.1.2 *Heuristics-based Query Optimization*

In contrast to statistics-based approaches, the heuristics-based optimization techniques generally work without any knowledge of the underlying database. Instead, they first define certain heuristics based on the observation on RDF data sources and then apply these heuristics to estimate the cost [183]. For example, in [90], Husain et al. discuss RDF query optimization in the cloud, which usually lacks data statistics. This approach aims to generate a query execution plan that has a minimal number of jobs. For this, the authors try to group together in a job as many joins as possible per join variable by employing the early elimination heuristic. Tsialiamanis et al. [192] follow a similar approach in which they try to maximize the number of merge joins by grouping together the triple patterns that share a common variable. In their work, the query optimizer with the help of heuristic algorithms is able to choose the join order by only looking at the syntactical form of the query, rather than relying on any statistics of the underlying data set. According to their evaluation, this approach out-stands many other approaches at time.

In [183], the author presents an approach for performing query planning and optimization based on an extended query pattern graph (ESG) and heuristics. An ESG is represented by vertices V (query expressions like BGP, filter etc.) and edges E. An edge links two vertices if they share at least one variable. In addition to the ESG, the authors also define a set of 8 heuristics:

- H1: The cost for executing query triple patterns is ordered as: *c(s,p,o)* <= *c(s,?,o)* <= ... <= *c(?,?,?)*
- H1*: The cost for execution query triple patterns is influenced by the distinct count of subject, predicate and object.
- H2: A triple pattern that is related to more filters has higher selectivity and cost less.

- H3: A triple pattern that has more variables appearing in filters has higher selectivity and less cost.
- H4: Basic query triples patterns have higher selectivity and cost less than named graphs.
- H5: A query executed with a specific graph pattern has higher selectivity and cost less.
- H6: The position of the join variable of two vertices influences the join selectivity.
- H7: Edges whose vertices share more variables are more selective.

The query planning and optimization are then divided into 3 steps: (1) generating the ESG graph; (2) estimating costs using heuristics; (3) finding best query plan using a greedy graph search algorithm. The evaluation is provided which describe the improvement in query performance on BSBM dataset [24]. No other datasets are tested.

Other approaches such as exploiting syntactic and structural variations of triple patterns in a query [186], and rewriting a query using algebraic optimization techniques [55] and transformation rules [79] are considered as heuristics-based approaches. In comparison with the statistics-based optimization approach, the heuristics-based ones address the problem of the lack of underlying data statistics. However, the major limitation of these approaches is that they are based on strong assumptions such as considering queries of certain structure less expensive than others. These assumptions may valid for some RDF data sets and may not valid for others.

7.2.2 Spatio-temporal Query Optimization

To process the spatio-temporal queries, the works in [72, 141, 106] build their own query parser to translate the given spatio-temporal query into an abstract syntax tree. The tree then is mapped to the internal query operators, resulting in a query plan tree. The physical spatio-temporal operators hence are implemented by either extending the Jena/Sesame frameworks or using existing spatial-temporal relational databases. However, these approaches mostly focus on enabling spatio-temporal query features, but hardly any of them fully address the query performance and optimization issues that are associated with querying billions of triples.

As regards dealing with the scalable processing of big spatio-temporal RDF data, distributed solutions are proposed in [108, 201]. These approaches adopt a loosely coupled hybrid architecture, which includes different database systems managed by a central middleware. The middleware works as an integration query processing engine that breaks an input query into sub-queries and delegates them to the proper underlying databases. For example, the spatial filter of the query is executed by a spatial database while the semantic patterns are processed by an RDF triple store like Jena, Sesame, etc. The query optimizer will then decide a query execution plan that consists of an execution order based on the selectivity estimation of each operator. It is important to note that the execution order has a significant impact on overall query performance. The first execution operator will prune all unnecessary results and hence will reduce the size of the search space for the next operator. However, generating an optimized execution order only based on a selectivity estimation model without taking into consideration the underlying environment performance - i.e, the data arrival rate, the characteristics of the network, cluster configuration - may face a significant risk that the optimization is critically wrong.

7.2.3 Machine Learning to Predict Query Performance

Query cost estimation made by statistics-based or heuristics-based cost model are good for comparing alternative query plans, but inefficient for predicting actual query performance [56, 7, 70]. In fact, they often result in inaccurate performance estimations, such as query execution time. This is because the actual query performance depends on so many environment-dependent parameters that are not possible to be captured into a cost model. In heterogeneous environments such as scalable sensor data management system, where the different hardware with different performance characteristics, building a reliable cost-model is even more complicated as the model would have to consider more things such as I/O costs for the selected devices, network latency, cluster configuration, etc. As coming up with a reliable cost-model is very difficult and often impossible to achieve, an alternative solution is to use machine learning techniques to learn query performance.

The recent success of numerous machine-learning-based applications has prompted the database community to investigate the possibilities for integrating machine learning techniques into the design of database systems, especially in query optimizers [5, 119]. This learning technique has become a good alternative to building complex cost models and is often easier to adapt and use with new configurations and data patterns. Moreover, it also enables the prediction of query execution performance which can benefit many system management decisions, including workload management, query scheduling, system sizing, and capacity planning. Following this direction, in the relational database area, approaches that exploit machine learning techniques to build predictive models have been proposed [7, 56]. These approaches predict the query execution time and the size of the results by comparing the similarity of query execution plan features. Instead of just predicting query performance, the works in [11, 200, 127] use machine learning to estimate the percentage of work done or to produce an abstract number that is intended to represent a relative query optimizer's cost.

In spite of sharp growth in the number of such published works, there are still limited works that use machine learning for query optimization on the Semantic Web. Hasan [81] proposed a state-of-the-art approach that uses machine learning to predict SPARQL query execution times. In this approach, Hasan firstly computes the similarity of a SPARQL query based on its algebraic expression and graph pattern and then predicts the execution time by applying multiple regression SVR. The model is evaluated using DBPSB benchmark queries [128] on an open-source triple store (Jena TDB). Following the success of this paper, Zhang et al. [202] present an improvement on this by replacing the clustering model with a hybrid feature vector. It is claimed that the training time is reduced significantly. However, in practice, we observed that applying these approaches for predicting query performance on spatio-temporal sensor data might give inaccurate results. This is because sensor data not only has semantic but also has spatio-temporal aspects, while these approaches only consider the semantic aspect of data in similarity computations. Our work will address this issue by taking into account the spatio-temporal dimensions and query cardinalities in similarity identification.

7.3 A LEARNING APPROACH FOR QUERY EXECUTION PLANNING

In this section, we describe our proposed learning approach for query planning on spatio-temporal sensor data. Our approach is illustrated in Figure 7.1.

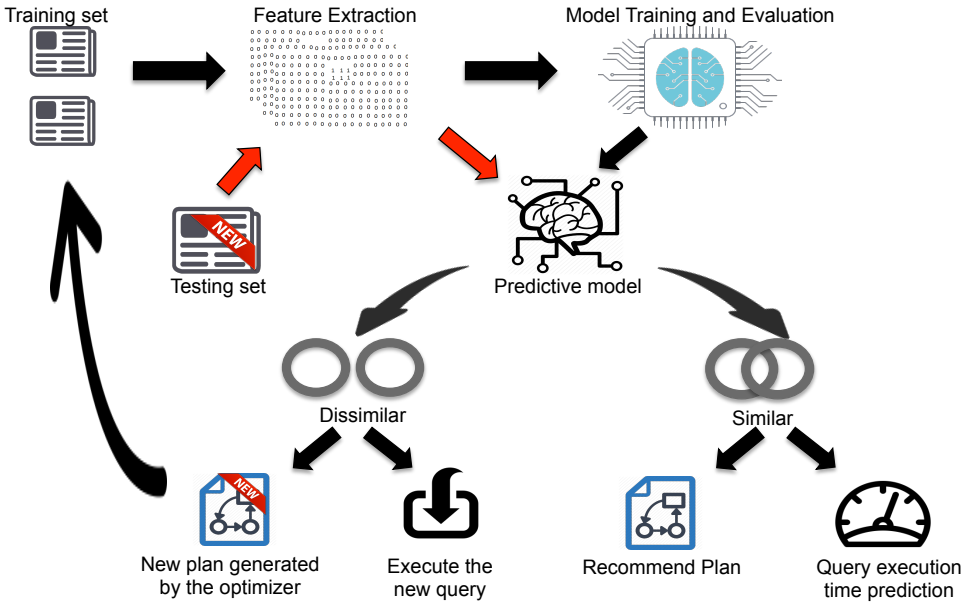


Figure 7.1: Overview of our approach

The approach consists of two distinct phases: an offline training phase and an online testing phase. Before starting the training phase, we first select the historical queries set and split it into three subsets for training, validation, and testing. These query sets will be executed on the target system from which the query execution plan and execution time are observed. To execute the query, we build an offline automatic script that runs separately to the training phase. The script is used to detect the optimized execution plan for each given query by considering all the possible plans. The plan which has the lowest cost will be selected. In the next step, the Features Extractor module will extract all the selected query features into the features matrix, which is a necessary input for machine learning algorithms. The features selection process will be presented in the following section.

The next component is Model Training and Evaluation, which is responsible for evaluating the efficiency of different learning models. For that, this component will train the models with the training queries set and then evaluate them by using the validation set. During the training and evaluation processes, the setting of parameters for each model can be optimized by using cross-validation procedures, i.e, the number of clusters, the k value for k-NN regression models, etc. The testing set, which is not used during model training, can be used for final testing and model comparison. By evaluating on a testing set, the learning model with the most accurate prediction will be selected as a Predictive Model, which will be used in the testing phase.

The second phase of our approach is online testing that runs the predictive model with a real-world query. For a given query Q_n, the Predictive Model will propose a suitable execution plan, already generated during the training phase, and reuse it to execute Q_n. For that, the Predictive

Model needs to compute the similarity of Q_n with trained queries that were previously executed by the optimizer. If similarity is detected between Q_n and a trained query Q_t , then the execution plan P_t used by the optimizer to execute Q_t will be reused to process Q_n. The same methodology is applied for query execution time prediction.

7.4 QUERY FEATURES EXTRACTION PROCESS

A typical setup for machine learning is to be given a collection of vectors containing all of the feature values for a given data set. Therefore, we represent our spatio-temporal query into its feature vector representation. In this study, we build upon the work in [81] which uses algebra and graph pattern features to build the features vector. The difference here is that we extend the SPARQL algebraic expression vector by adding spatial and temporal metrics. For the graph pattern, we not only extract the SPARQL basic graph pattern but also consider the spatial and temporal patterns in our graph similarity computation. In addition, as well as the algebra and graph patterns, we also include query cardinality features. According to our experimental results, this extra feature proves its effectiveness by improving the prediction results significantly.

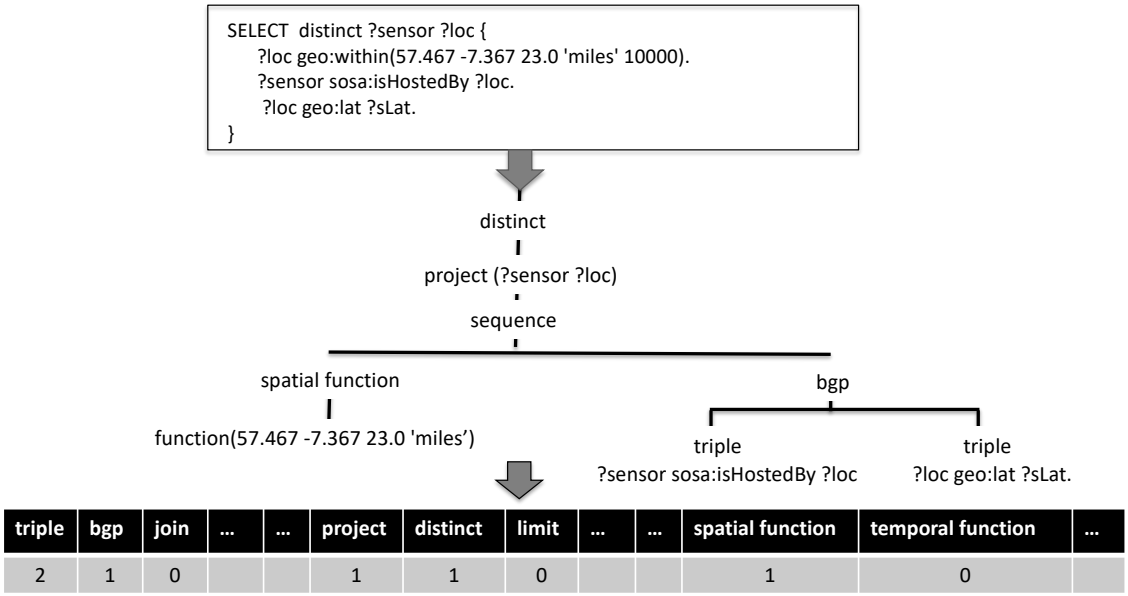
7.4.1 SPARQL Algebra Features with Spatio-temporal Extension

This step is used to transform the spatio-temporal SPARQL query strings to algebraic expressions. For an input query, the query engine will parse the query to an algebraic expression tree. In this tree, non-leaf nodes are algebraic operators such as joins, and leaf nodes are the variables present in the triple patterns of the initial query. Additionally, we add two more operators that are unique in our engine: the spatial and temporal operators. After parsing a query string to an algebraic expression, our Feature Extractor component will calculate the frequencies of all the algebraic operators as query features. Figure 7.2 shows an example of extracting the query algebra features vector from a spatio-temporal SPARQL query.

As shown later in our experiments, the prediction obtained using a similarity computation based on query algebra features is quite accurate in recognizing the queries that are almost identical. However, by carrying out a further analysis into the incorrectly predicted results, we recognized two significant risks:

1. The predictive model may judge that two queries are not similar when analyzing their algebraic expression trees, even though their execution plans are actually very similar.

2. The predictive model may judge that two queries have a high similarity measure when comparing their algebraic expressions, even though their execution plans are different. This is due to the fact that even small differences in query graph patterns or differences in the cardinality of each query operator, which would normally be ignored in the textual presentation of an algebraic expression, can be very decisive during the process of defining an execution plan.



triple	bgp	join	...	...	project	distinct	limit	...	...	spatial function	temporal function	...
2	1	0			1	1	0			1	0	

Figure 7.2: Algebra Feature Extractor

To address this problem, we take into account the query graph pattern similarity, which will be described in the next section.

7.4.2 Spatio-temporal Graph Patterns Features

In respect of query planning, judging two queries as being similar based on their algebraic expressions can present a significant risk in terms of an incorrect prediction. We realized when computing query similarities based on algebraic features that we only considered the frequency of the query operator, i.e, the number of triples in the basic graph patterns appearing the queries, but failed to represent the query graph structure. Recalling that a SPARQL query can also be considered as an RDF graph, it is therefore obvious that queries that have some structural similarity might potentially share the same execution plan and query performance.

To represent the spatio-temporal graph patterns, we propose building graph pattern features. Specifically, we transform the similarity problem of two query patterns to the similarity problem of two graphs. To compute the structural similarity between two query patterns, we first construct two graphs from the two query patterns, then compute the graph edit distance between these two graphs. In the following, we first shortly introduce the notion of graph edit distance and then present the graph patterns features extraction process.

7.4.2.1 *Graph Edit Distance*

We paraphrase the graph and the graph edit distance definitions from [161] as follows.

Definition 7.1 ***(Graph)*** *A graph g is a tuple* $g = (V, E, \mu, \nu)$ *where*

- *V is the finite set of nodes*

- $E \subseteq V \times V$ *is the set of edges*
- $\mu : V \rightarrow L$ *is the node labeling function*
- $\nu : E \rightarrow L$ *is the edge labeling function*

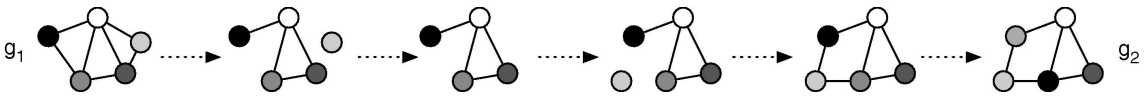


Figure 7.3: A possible edit path between graph g_1 and g_2 [161]

The graph edit distance between two graphs is the minimum amount of distortion that is needed to transform one graph into another. The amount of distortion is considered as the cost of a sequence of graph edit operations. A standard set of graph edit operations is given by deletions, insertions, and substitutions of both nodes and edges. Figure 7.3 demonstrates a possible edit path to transform graph g_x to graph g_y. In this example, the list of edit operations consists of three edge deletions, one node deletion, one node insertion, two edge insertions, and finally two node substitutions.

It is obvious that there is a number of different edit paths transforming g_x to g_y. Let $\Upsilon(g_x, g_y)$ be the set of all such edit paths. To find the most suitable edit path in $\Upsilon(g_x, g_y)$, a cost function is introduced. The cost function aims to define whether or not an edit operation represents a strong modification of the graph. Obviously, there should be an inexpensive edit path for two similar graphs, which represents low-cost operations. Meanwhile, for dissimilar graphs, an edit path with high costs is needed. As a result, the edit distance of two graphs is defined by an edit path with a minimum cost between the two graphs.

Definition 7.2 ***(Graph Edit Distance)*** *Let* $g_x = (V_x, E_x, \mu_x, \nu_x)$ *be the source and* $g_y = (V_y, E_y, \mu_y, \nu_y)$ *the target graph. The graph edit distance between* g_x *and* g_y *is defined by:*

$$d(g_x, g_y) = \min_{(e_1 \ldots e_k) \in \Upsilon(g_x, g_y)} \sum_{i=1}^{k} c(e_i) \tag{7.1}$$

7.4.2.2 *Graph Patterns Features Extraction Process*

Figure 7.4 illustrates the graph patterns features extraction process. The process can be split into 2 steps, which are briefly described as follows.

The first step is to cluster the structurally similar graph patterns in the training data into K_{med} clusters. We apply the same clustering algorithms proposed in [81], which are *K-mediods* clustering algorithm [98] and Risen's graph distance [161], to cluster the training queries. The *K-mediods* is used as it chooses data points as cluster centers and allows using an arbitrary distance function. However, in addition to the standard SPARQL graph patterns, we also consider the spatio-temporal graph patterns when computing the graph similarity. We built our own graph pattern extractor to not only extract the semantic graph pattern but also to extract the spatial and temporal patterns. The spatio-temporal graph pattern will thus be considered as a standard RDF graph in which the edges of the graph are spatio-temporal functions and the nodes of the graph are the input variables for the functions. We call such a graph a spatio-temporal graph. Figure

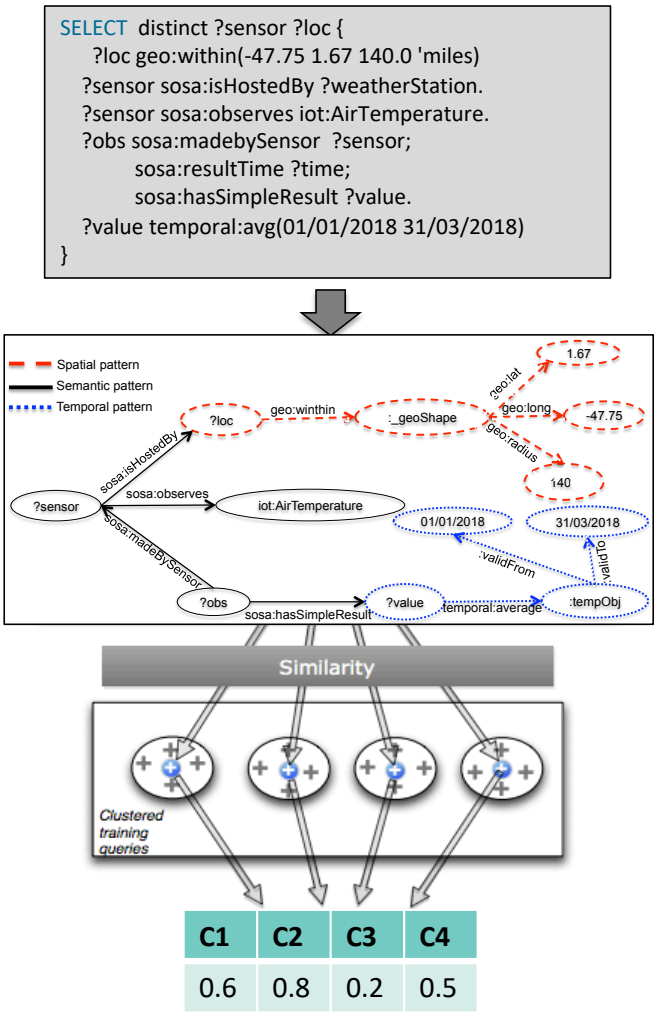


Figure 7.4: Graph pattern features extraction process

7.5 illustrates the graph representation of our sample spatio-temporal query. To ensure that the spatio-temporal triples are more important than standard triples, we heuristically improve the weighting function in the graph edit distance method by increasing the cost of edit operations on the spatio-temporal graph. The final output of this clustering step is set of K_{med} clusters, each cluster will have a center graph pattern which is a representative of query patterns in that cluster.

After having a set of K_{med} clusters in Step 1, we then compute the graph edit distance between the graph of each query q_i in the training set and the center graph patterns of each cluster. The structural similarity between these graphs are computed following the method described in [81] as below:

$$sim(q_i, C(k)) = \frac{1}{1 + d(q_i, C(k))} \tag{7.2}$$

where $d(p_i, C(k))$ is the graph edit distance between the query graphs q_i and the center graph $C(k)$ of the cluster. This formulation results in a score within the range $[0, 1]$. The score of 0 being the least similar and a score of 1 being the most similar. As a result, we obtain a K_{med}-dimensional feature vector for q_i, where K_{med} is the number of cluster.

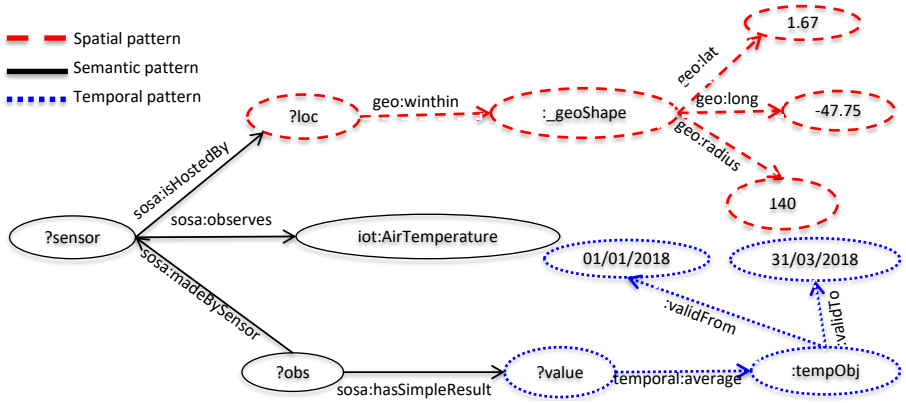


Figure 7.5: Mapping spatio-temporal query patterns to graph

7.4.3 Query Cardinality Features

We build query cardinality features by selecting the cardinality of each dimension of the input query. We observe that there are some queries that share structurally similar basic graph patterns and the same algebraic expressions, but neither the query execution plan nor the query execution time is similar. This is because the cardinalities (spatial cardinality, temporal cardinality, semantic cardinality) are not considered in the similarity computation for both the algebra and graph pattern features. Note that the cardinalities will help the optimizer to decide the execution order of each operator. For example, if the spatial cardinality of the query is small, the query plan should process the spatial operator first, pruning unnecessary results, followed by the semantic part of the query. For simplicity, we assume that the temporal operator always has the highest cardinality due to the massive amount of historical IoT data that can be available. Hence, we limit our study to selecting the spatial and semantic cardinalities in order to create a Query Cardinality features vector. We retrieve spatial cardinality quickly by performing the bounding box query defined in the spatial operator. We then reuse the work in [186] to estimate the semantic cardinality. A brief description of these features is given in Table 7.1.

Feature	Description
Sp_ARD	The estimated result size of spatial operator
Se_ARD	The estimated result size of semantic patterns

Table 7.1: Description of the Query Cardinality Feature

7.5 EXPERIMENTS

7.5.1 Experiments Setup

7.5.1.1 *Setup and Software*

In addition to the physical cluster that deploys our spatio-temporal query processing engine presented in Chapter 6, another separated server is dedicated for training our prediction model. The server configuration is: 2x E5-2609 V2 Intel Quad-Core 2.5GHz 10MB Cache, Hard Drive 3x 2TB 7200RPM, Memory 32GB1600MHz DDR3.

We experimented with Jena v2.13.1 to extract algebra features and have used Weka v3.8.2 to implement the machine learning algorithms. The graph edit distance was calculated using the Graph Matching Toolkit [160]. The Java version we used is JAVA 8.

7.5.1.2 *Dataset and Queries*

We again use our linked meteorological dataset presented in Chapter 4 and our benchmark query templates in Appendix A for training and testing our learning model. Each query template consists of the placeholders that can be replaced by specific terms. From these templates, the training, validation and test queries will then be generated. To generate queries, we randomly assign selected RDF terms from the meteorological dataset to the placeholders in the query template. We generate over 2244 queries that are divided equally between all templates. We take about 40% of the queries for training purposes, about 27% for validation and about 33% for testing. This process results in 900 training queries, 600 validation queries, and 744 testing queries.

7.5.1.3 *Predictive Models*

In our experiment, we evaluate three machine learning algorithms available in the Weka library. Each algorithm is considered as the most representative machine learning algorithm in its category. We firstly run our experiment with the Support Vector Machine for regression (SVR) [68]. The SVR creates a maximum-margin hyperplane to split the input features to a higher dimensional space and then performs a regression in that space via a linear regression function. We performed the SVR test with two commonly used kernels: the RBF and Polynomial kernels. The second algorithm is the k-Nearest Neighbors (k-NN) [9]. Conceptually, k-NN is a lazy learning algorithm that predicts based on the closet training data point. For our experiment, we apply two variations of k-NN by considering different distance functions: Euclidean distance and Manhattan distance. The last learning model in our experiment is Random Forest [114], which is a typical representation of a Decision Trees Regression model. Random Forest is an improvement upon bagged decision trees that disrupts the greedy splitting algorithm during tree creation so that split points can only be selected from a random subset of the input attributes.

7.5.1.4 *Evaluation Metrics*

Our work applies classification and regression models to query plans and query execution times, respectively. Therefore, we use a classification metric, called *accuracy*, to evaluate the prediction of query execution plans. For query execution time, we use *root mean square error* and the *coefficient of determination* as our prediction metrics. Details of each of these metrics are described as follows:

- **Accuracy**: The accuracy metric measures the ratio of correct predictions to the total number of instances evaluated. The prediction accuracy is defined as:

$$\text{Accuracy} = \frac{1}{n}\sum_{i=1}^{n} 1(\hat{y}_i = y_i)^2 \tag{7.3}$$

  where, $\hat{y}$ is the predicted value and y_i is the corresponding actual value, n is total number of observations

- **Root Mean Square Error (RMSE)**: RMSE is used to measure the differences between values predicted by a model and the actual values. The RMSE is defined as:

$$\text{RMSE} = \sqrt{\frac{\sum_{i=1}^{n} (\hat{y}_i - y_i)^2}{n}} \tag{7.4}$$

- **Coefficient of Determination (R^2)**: The R^2 provides a measure of how well future samples are likely to be predicted by the model. The R^2 value is always between 0 and 1. An R^2 score close to 1 indicates near perfect prediction. The R^2 is defined as:

$$R^2 = 1 - \frac{\sum_{i=1}^{n} (\hat{y}_i - y_i)^2}{\sum_{i=1}^{n} (\hat{y}_i - \bar{y})^2} \tag{7.5}$$

  where, $\bar{y}$ is the mean of actual value

7.5.2 Experiment Results

7.5.2.1 *Experiment 1: Train model with algebra feature*

In this experiment, we evaluate the performance of different learning algorithms, namely SVR-RBF and SRV-Polynomial (SVR-Pol), k-NN with Euclidean distance (KNN-Eu), k-NN with Manhattan distance (KNN-Mah), and Random Forest (RF). We first compare the prediction accuracy of these algorithms for the query execution plan and then present the results of their performance for query execution time. For the k-NN algorithms, we selected the k value, the number of neighbors, using cross-validation. We found that k=2 gives us the best results.

Table 7.2 reports the execution plan prediction accuracy. We observe that the accuracy values of these algorithms are just slightly different, rounding to 80%. The highest value is realized by the RF algorithm, followed by SVR-Pol and KNN-Eu.

	SVR-RBF	SVR-Pol	KNN-Eu	KNN-Mah	RF
Accuracy	79.704%	80.107%	79.973%	79.926%	80.127%

Table 7.2: Query plan prediction accuracy using Algebra features

In the following, we look at the query execution time prediction results. The regression performances of the learning algorithms are evaluated by RMSE and R^2, as shown in Table 7.3. One interesting fact we noticed is that, in contrast to the execution plan prediction, the SVR algorithms (SVR-Pol and SVR-RBF) have poor performance. The minimal values of R^2 (0.01 and 0.113) are given by the SVR algorithms while the k-NNs and RF perform in almost the same way with values equal to 0.976 and 0.975, respectively. The R^2 values indicate that the k-NN and RF algorithms predict more accurately than the SVR algorithms. Additionally, the RMSE values also imply the bad performance of the SVR algorithms with high values of 366.359 and 357.288 for SVR-RBF and SVR-Pol. To plot comparison of the predicted and actual execution time values, we select the algorithms that have the best performance in each group, which are SVR-Pol, KNN-Eu, and RF. The plots are shown in Figure 7.6. We use a log-scale to accommodate the wide range of query execution times. As illustrated in these figures, there are still a substantial number of outliers, which indicate incorrect prediction results. According to our observations, these incorrect predictions fall into the category of queries which share the same algebraic expression but have different query plans and execution times.

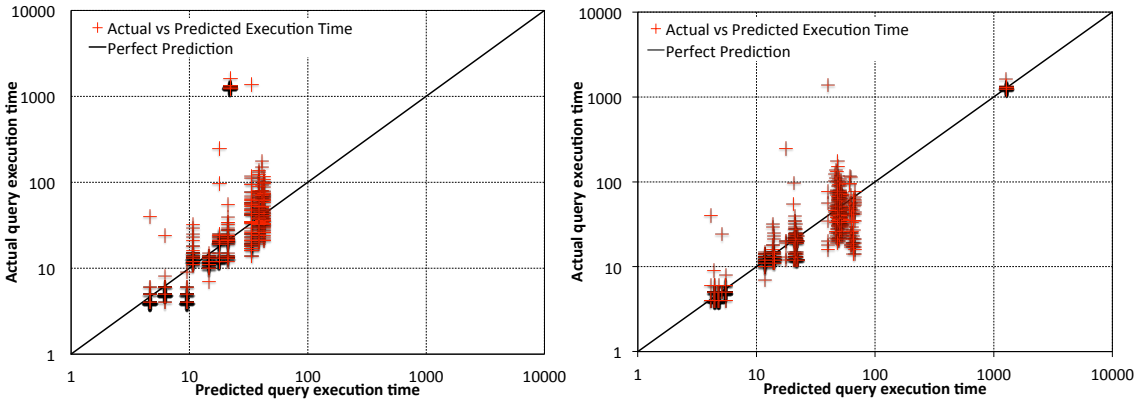
	SVR-RBF	SVR-Pol	KNN-Eu (k=2)	KNN-Mah (k=2)	RF
RMSE	366.359	357.288	54.809	154.245	54.927
R^2	0.01	0.113	0.976	0.976	0.975

Table 7.3: R^2 and RMSE values using Algebra features

7.5.2.2 *Experiment 2: Train model with algebra features and spatio-temporal graph patterns*

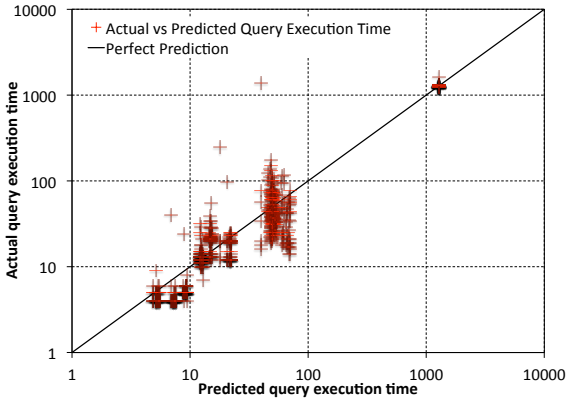
In the second experiment, we again use only three algorithms which are derived from the previous experiment, namely SVR with Polynomial kernel, k-NN with Euclidean distance and Random Forest. Using the cross-validation procedure, we are able to select and set the number of clusters for the k-mediods clustering algorithm at $K_{med} = 25$, and the number of neighbors for k-NN at k=2. We chose these values because they give us the lowest RMSE and highest R^2 values. Table 7.4 presents the query plan prediction accuracy. By adding the spatio-temporal graph pattern features in our similarity computation, we achieve the highest accuracy value of 87.68% with the RF algorithm. Following this is the KNN-Eu one with 86.42%. The SVR-Pol has increased slightly at 80.64%.

Figure 7.7 and Table 7.5 portray the prediction results for query execution time. As shown in Figure 7.7, the prediction results of KNN-Eu and RF move closer to a perfect prediction. The highest R^2 value is 0.979 (KNN-Eu). Analyzing the results, we realized that by adding the graph patterns features, the number of inaccurate predictions for two queries that have structurally similar spatio-temporal graph patterns but different algebraic expressions is reduced



(a) SVR-Polynomial kernel

(b) k-NN with Euclidean distance



(c) Random Forest

Figure 7.6: Regression-predicted vs. actual execution times using Algebra features

significantly. This achievement is also explained by the decrease in RMSE values as described in Table 7.5. The most considerable decrease in RMSE belongs to KNN-Eu, which is 54.809. The RMSE of the SVR-Pol algorithm also substantially reduces, which is now 129.23. These are the main improvements from the first experiment with the algebraic features model.

	SVR-Pol	KNN-Eu (k=2)	RF
Accuracy	80.645%	86.425%	87.688%

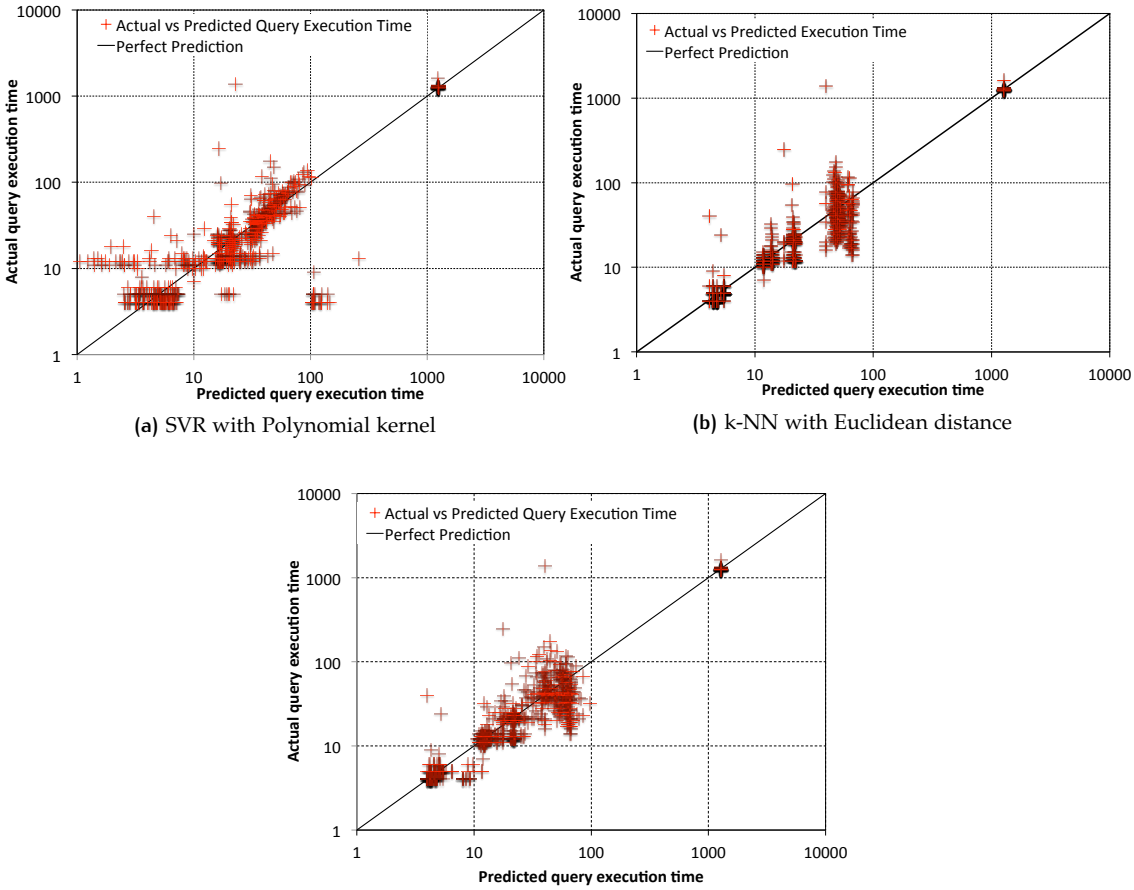
Table 7.4: Query plan prediction accuracy using Algebra and Graph patterns features

	SVR-Pol	KNN-Eu (k=2)	RF
RMSE	129.231	54.809	54.927
R^2	0.970	0.979	0.976

Table 7.5: R^2 and RMSE values using Algebra and Graph patterns features

7.5.2.3 *Experiment 3: Train a model with algebraic features, spatio-temporal graph patterns, and query cardinality features*

So far, we have described the experimental results for predicting query execution plans and query execution times using algebra and graph patterns as feature vectors. We also ran the experiment by adding query cardinality features. As mentioned in Section 7.3, this additional



(a) SVR with Polynomial kernel

(b) k-NN with Euclidean distance

(c) Random Forest

Figure 7.7: Regression-predicted vs. actual execution times using Algebra and Graph patterns features

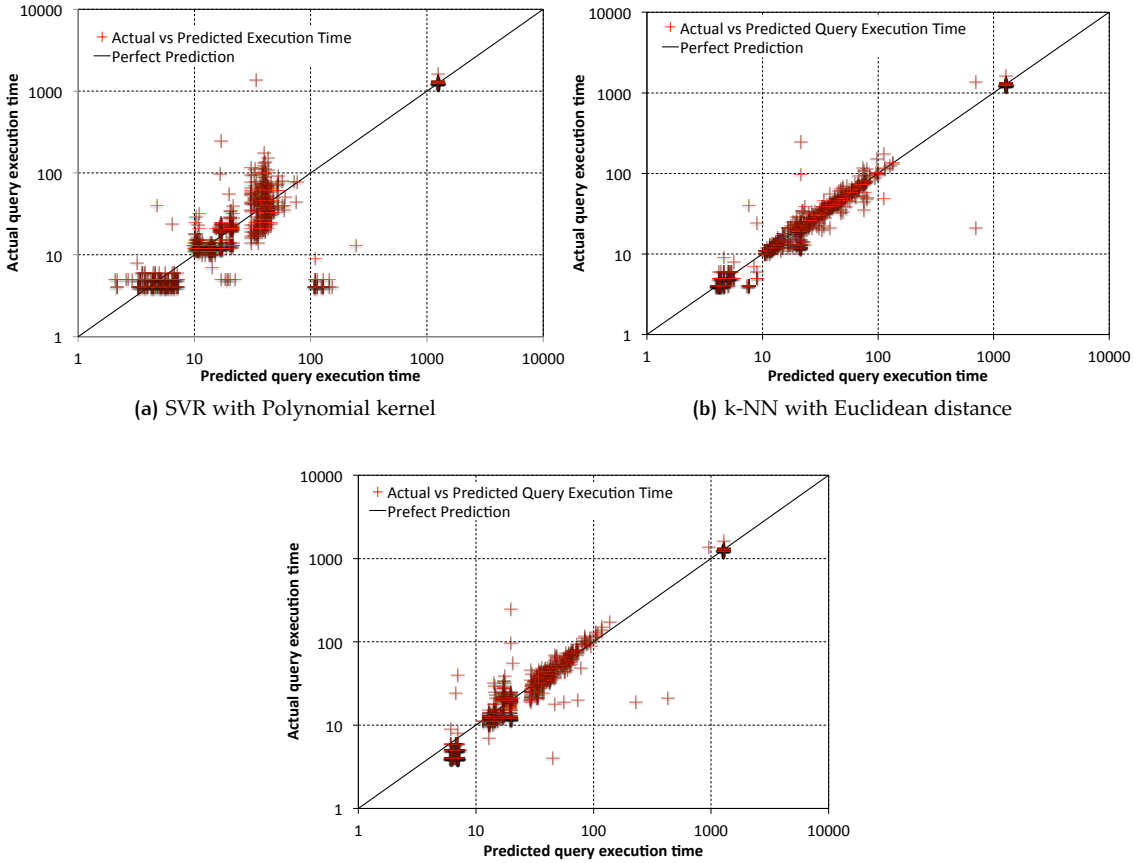
feature vector helps to improve the predictive accuracy of spatio-temporal queries that are similar in terms of algebraic expression and graph pattern representation, but that differ in query performance. Not surprisingly, the prediction results of this experiment are more accurate compared to Experiment 2. This is evidenced by the increased prediction accuracy values shown in Table 7.6, and the fewer outliers with respect to the perfect prediction line in Figure 7.8. Specifically, in Table 7.6, KNN and RF continually show their efficiency with the highest achieved accuracies of 95.054% and 95.860% with regard to query execution plan prediction. For the query execution time prediction results, as described in Table 7.7, we get an overall best R^2 value of 0.990 (using RF) and an overall lowest RMSE value of 35.746 (using RF). These results indicate that our model can accurately predict the query plan and execution time for a given query.

	SVR-Pol	KNN-Eu (k=2)	RF
Accuracy	93.414%	95.054%	95.860%

Table 7.6: Query plan prediction accuracy using Algebra, Graph patterns and Query cardinality

	SVR-Pol	KNN-Eu (k=2)	RF
RMSE	61.062	39.273	35.746
R^2	0.972	0.987	0.990

Table 7.7: R^2 and RMSE values using Algebra, Graph patterns and Query cardinality



(a) SVR with Polynomial kernel

(b) k-NN with Euclidean distance

(c) Random Forest

Figure 7.8: Regression-predicted vs. actual execution times using Algebra, Graph pattern and Query cardinality features

7.5.2.4 *Training and Prediction Required Time*

In this section, we report the total training time and average prediction time per query for our proposed prediction models. The results are shown in Table 7.8. In general, within the same training model, kNN-Eu has the least required time for training and prediction, followed by SVR-Pol and RF. Another interesting fact is that the required time for training and predictions of all models are opposite to their prediction accuracy. For example, according to the evaluation results in the previous sections, the training model with SPARQL algebra feature has the most inaccurate results. However, we can see in Table 7.8 that the required time for this model is the lowest one, in comparison with others. The average prediction time per query is also very low. We attribute this to the simplicity of the training models, that does not require too much time for building the algebra feature vector.

Not surprisingly, training models with additional spatio-temporal graph pattern features take more time to train, even this model is more efficient than the one with algebra feature. The reason for the increase in the training time is because it also consists the time for generating the distance matrix using approximated graph edit distance. In particular, this process itself takes almost 3100 seconds on average for 2244 queries. Additionally, it also includes the time required to cluster the training queries. Regarding the average prediction time per query, the average values for all algorithms are far higher than the ones with algebra feature. However, they are all

under 72 milliseconds, which still indicates a good prediction performance, especially for query over Linked Sensor Data.

Finally, due to their complexity, training models that combine algebra, graph pattern and cardinality features require most time for training process and query planning prediction. In all cases, the training time is more than 1 hour. However, the average prediction time per query using models with graph pattern features is within 100 milliseconds and is still acceptable. It is important to note that the training phase is an offline process and hence it does not influence the query prediction time.

Table 7.8: Required time for training and query planning predictions

Model	**Training time**	**Average prediction time per query**
kNN-Eu + algebra	158 sec	2.12 ms
SVR-Pol + algebra	222 sec	2.23 ms
RF + algebra	289 sec	2.20 ms
kNN-Eu + algebra + graph pattern	3175 sec	56.1 ms
SVR-Pol + algebra + graph pattern	3218.12 sec	68.9 ms
RF + algebra + graph pattern	3311 sec	70.6 ms
kNN-Eu + algebra + graph pattern + cardinality	3977 sec	88.4 ms
SVR-Pol + algebra + graph pattern + cardinality	4210 sec	92.7 ms
RF + algebra + graph pattern + cardinality	4356 sec	92.9 ms

7.5.2.5 *Performance Comparison Between Using Prediction Models and Cost-based Optimization for Query Planning*

We repeat the concurrent processing capability test that is already described in Section 6.5.2.3 of Chapter 6. However, in this experiment, we use our proposed learning approach for query planning. Figure 7.9 reports the performance comparison in the case of 1000 query clients between the two experiments. It is apparent that the concurrent processing capability of our system is enhanced for all test queries. Moreover, the performance is significantly improved in the case of a large number of query clients. This is evidenced by the considerable decrease in the query execution time for Q4, Q5, Q7, Q10 in comparison with the original experiment. We attribute this to the difference between the time needed for generating an execution plan and the query plan prediction required time. It is important to mention that, in the experiment in Section 6.5.2.3, for the spatio-temporal queries that have high complexity, the execution plan search space is also exploded. The required time for searching and generating a proper execution plan consequently increases. When the system is under a heavy workload, this can be even worse. Meanwhile, in our approach, regardless of the required training time, the query execution plan prediction time for a given query is less and is slightly affected by the query complexity. Moreover, because the prediction process is executed in a dedicated server, the influence that causes by the workloads of concurrent queries to the system hence is minimized.

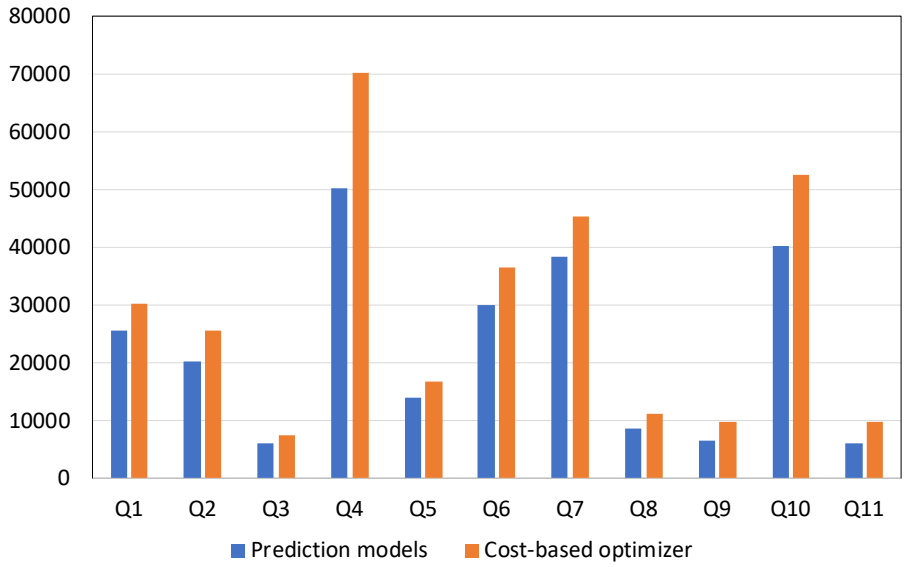


Figure 7.9: Performance comparison of concurrent queries (1000 clients) between prediction models vs cost-based optimization

7.6 SUMMARY

In this chapter, in order to address the query planning issues discussed in Chapter 6, we propose our approach to accurately predict both query planning and the query execution time using similarity identification techniques on spatio-temporal SPARQL queries. Our goal was to build a machine learning model that recognizes the similarity of queries, and only using the information that is available before the queries are to be executed. Our extensive experimental results demonstrate the efficiency of our learning approach for query planning on spatio-temporal sensor data. The most significant improvement in our approach is achieved by the inclusion of spatio-temporal information in the query similarity computation. Also, together with the algebraic expression and spatio-temporal graph patterns, we propose a new feature vector, namely query cardinality, to enhance the prediction accuracy in the case of queries that share similar algebraic expressions and graph patterns, but that have different query performance. The experimental results indicate that our approach can accurately predict over 95% of input queries. Furthermore, in comparison with the original cost-based query optimizer, the query performance of our learning approach is significantly improved with respect of a large number of query clients. With these results, we believe our prediction models can be complementary to the existing EAGLE query optimizer so that it can adapt to the changes in the underlying environment or dealing with high volume concurrent queries.

8 CONCLUSIONS AND FUTURE WORK

The need for efficient querying on a massive amount of sensor data lies at the heart of most sensor data analytics platforms. This thesis presents our recent efforts on leveraging the linked data and NoSQL technologies to effectively manage sensor data. Our approach provides not only complex spatio-temporal query functions to users but also proves the ability to handle billions of sensor data. In this chapter, we summarize a number of original contributions that have been produced in this thesis and suggest directions for future work arising from the developments undertaken.

8.1 CONTRIBUTIONS

Publishing Linked Meteorological Data on the LOD Cloud

For addressing a lack of spatio-temporal linked sensor datasets in the LOD could, in Chapter 4, we present a publishing process of our linked meteorological dataset. The dataset is built upon the ISH data, which is originally published by NOAA. Our linked meteorological dataset consists of 26 billion triples that are transformed from 3.7 billion ISH observation records. Moreover, the dataset is enriched with the additional semantic sensor description by interlinking the dataset with other data sources. The dataset has a vital role and has been used in all experimental evaluations presented in this thesis. Additionally, in Chapter 4, we also present our proposed linked sensor data model. The model is developed upon a combination of several existing ontologies such as SOSA/SSN, GeoSPARQL and OWL-Time, which aims to semantically describe the spatial and temporal data aspects of sensor data.

Systemic Evaluation of RDF Stores for Linked Sensor Data

The publishing process of our linked meteorological dataset on the LOD cloud raises a challenge of selecting appropriate RDF store for hosting the dataset. Unlike the traditional RDF stores, the selected one should support spatio-temporal queries so that spatio-temporal characteristics of the dataset can be preserved. Moreover, this store should also have capability to deal with "big data" challenge of processing sensor data. While there are no such studies on RDF stores assessment for Linked Sensor Data, this motivates us to conduct a first performance study of RDF stores for storing and querying sensor data, presented in Chapter 5. This work aims to analyze the weaknesses and strengths of current triple store implementations applied in Linked

Sensor Data context. In this study, we have summarized a list of fundamental requirements of RDF stores so that they can be used for managing and querying sensor data. We have also described the abstract architecture design of current RDF database technologies that support spatio-temporal queries. Additionally, we have provided a comparative analysis of data loading and query performance of a representative set of RDF stores. In this evaluation, particular attention has been given on evaluating the performance of geo-spatial search, temporal filter and full-text search over our linked meteorological dataset. Since such assessment aspects have not been fully considered and addressed by the existing evaluations, our study gives valuable insights about the advantages and limitations of the current RDF stores implementation when applying for Linked Sensor Data.

A Scalable Spatio-temporal Query Processing Engine for Linked Sensor Data

Following the assessments in Chapter 5, several drawbacks of the current RDF stores for Linked Sensor Data have been analyzed. One of the major limitations is the poor performance of these stores when dealing with a massive amount of sensor data. Another one is a lack of spatio-temporal analytical query support. These challenges motivate us to propose EAGLE - a scalable spatio-temporal query processing engine for Linked Sensor Data. EAGLE adopts a loosely couple hybrid architecture that consists of different clustered databases for managing different aspects of sensor data. This flexible architecture not only helps the engine to deal with the overhead of "big data" processing but also allows us to make use of the existing spatio-temporal query functions provided by the underlying databases. Another contribution is our spatio-temporal SPARQL language extensions that aim to support querying spatial and temporal RDF data. The query language extensions adopt the GeoSPARQL syntax and also provides a set of temporal analytical property functions.

Efficient Spatio-temporal Data Partitioning Strategy

Along with the query processing and spatio-temporal query language, we also propose the OpenTSDB-based storage model to address the performance issues of temporal data loading and spatio-temporal analytical queries. In this approach, we propose a rowkey design scheme that takes the geohash prefix of the sensor location and the observation result time as the first two encoded elements of the rowkey. Following them are the additional descriptions of observation data such as sensor URI, observed property, etc., that can be used for filtering data. Moreover, by using this rowkey scheme, the spatio-temporal locality of data is preserved. In particular, data of all the sensors that are located within the same area are ordered and stored in the same partitions on disk. This essential feature allows the query engine to accelerate range scans for the query that requests data of specific area over a time interval. In addition to the geohash-based rowkey design, we also propose a spatio-temporal data partitioning strategy that allows the OpenTSDB data tables to be pre-split at the time of table creation. Pre-splitting data tables helps the data to evenly distributed across all servers so that region hot-spotting issues can be avoided. Also, this spatio-temporal partitioning strategy helps the engine to minimize the time cost for locating the required data partition and the data scan operation.

Efficient and Adaptive Query Planning

To enhance query optimization, we propose a learning approach that can accurately predict query planning for a given spatio-temporal query based on the existing execution plans. Compared to the state of the art, a major contribution of our approach is the use of machine learning to improve similarity detection by not only taking into account semantic aspects but also spatio-temporal correlations. In addition to query planning, we also aim to predict the query execution time to help the query engine have a better understanding of the query behavior prior to query execution. Accurately predicting the query execution time enables us to optimize our costly cloud infrastructure by rescheduling or preventing long-run queries that might cause too much resource contention or that will not be completed by a particular deadline. Our experiments confirm that the learning approach can improve the performance of the query engine by orders of magnitude. Whilst our approach still has its limitations, it is a step towards providing an alternative solution that aims to address the query optimization challenge for the spatio-temporal management of sensor data.

8.2 FUTURE WORK

The results presented in this thesis open up several possibilities for future work. In the following, we discuss future directions of research on which this thesis can be extended.

Applied to Parallel Join Algorithms

The final implementation in Chapter 6 has demonstrated that EAGLE can very quickly execute a spatio-temporal query over large scale sensor data. Regardless, in the future, we still want to further improve the query performance by investigating parallel join algorithms. Currently, in EAGLE, the join operation of the main operators such as BGP, spatial and temporal operators, is performed by a single thread. To avoid the memory pressure and the potentially bottleneck performance in case of joining large volumes of intermediate query results, a more efficient parallel join algorithm is desirable.

Distributed RDF Store Integration

Currently, our EAGLE system uses Jena TDB for storing sensor metadata. However, because Jena TDB is a centralized triple store that does not support clustering feature, EAGLE might have bottleneck performance issues if the size of this non-temporal-spatial dataset is over the size that Jena TDB can handle. Therefore, we also intend to integrate a distributed triple store to EAGLE to handle larger non-temporal-spatial data partitions. We are looking into both commercial and

open-source clustered RDF stores such as CumulusRDF [109], AllegroGraph [1], Blazegraph[1], etc.

Support Allen's Temporal Relations

Another feature that we want to add in the next version of EAGLE is enabling Allen's temporal relations. In this regard, additional temporal index algorithms need to be further investigated. Moreover, our current spatio-temporal query language also needs to be extended. Several existing works can be referenced such as [142, 73, 65, 18].

Data Compression

Currently, EAGLE implementation over the distributed system mainly focuses on bulk operations, in which the transferred data is still in the original format and the data chunk size is normally very large. This can have negative impact on the system performance, especially in the case of slow network bandwidth connection. In this scenario, saving network communications would make a great improvement to the EAGLE performance. For that reason, additional techniques on this aspect can be further investigated. For example, efficient compression/decompression methods for data communication would be very relevant, although there will be a trade-off between the computation and communication cost.

Enhance Query Optimization

For query optimization, we want to address some of the drawbacks that still exist. One major limitation of our approach is its poor adaptability to dynamic streaming sensor data. We are investigating techniques to improve the learning model so as to be amenable to continuous retraining in the context of streaming queries. Another drawback is that temporal cardinality is not adjusted in our learning model. This is due to our implicit assumption that this cardinality always has the largest value compared to the other cardinalities. We have a work-in-progress to efficiently estimate the temporal cardinality, and hence, this issue will be resolved in the near future.

1 http://www.blazegraph.com/

Appendices

A EVALUATION QUERIES

PREFIXES

```
PREFIX sosa: <http://www.w3.org/ns/sosa/>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX temporal: <http://jena.apache.org/temporal#>
PREFIX wgs84: <http://www.w3.org/2003/01/geo/wgs84_pos#>
PREFIX got:<http://graphofthings.org/ontology/>
PREFIX geoname: <http://www.geonames.org/ontology#>
PREFIX spatial: <http://jena.apache.org/spatial#>
PREFIX dbr: <http://dbpedia.org/resource/>
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX dcterms: <http://purl.org/dc/terms/>
```

Query 1. Given the latitude and longitude position, it retrieves the nearest weather station within 10 miles

```
SELECT ?station ?coor
WHERE
{
    ?station a got:WeatherStation.
    ?station geo:hasGeometry ?geoFeature.
    ?geoFeature wgs84:geometry ?coor.
    #VOS FILTER (<bif:st_within>(?coor, <bif:st_point>($long$, $lat$),$radius$)).
    #EAGLE #RDF4J  ?geoFeature geo:sfWithin ($lat$ $long$ $radius$).
    #GraphDB ?geoFeature omgeo:within ($lat$ $long$ $radius$).
    #Stardog ?geoFeature geof:within ($lat$ $long$ $radius$).
    #Jena    ?geoFeature spatial:withinCircle ($lat$ $long$ $radius$).
}
```

Query 2. Given the country name, it retrieves the total number of weather station deployed in this country

```
SELECT (count(?station) as ?total)
WHERE
{
    ?station a got:WeatherStation.
    ?station geo:hasGeometry ?geoFeature.
    ?geoFeature wgs84:geometry ?coor.
    ?geoFeature geoname:parentCountry "$country name$".
}
```

Query 3. Given the country name and year, it detects the minimum temperature value that has been observed for that specified year

```
SELECT min(?value) as ?min  ?station
#EAGLE SELECT ?value ?station
WHERE
{
    ?station a got:WeatherStation.
    ?station geo:hasGeometry ?geoFeature.
    ?geoFeature geoname:parentCountry "$country name$".
    ?sensor sosa:isHostedBy ?station.
    ?sensor sosa:observes got:SurfaceTemperatureProperty.
    ?obs sosa:madebySensor  ?sensor;
                 sosa:resultTime ?time;
                 sosa:hasSimpleResult ?value.
    FILTER (year(?time)=$year$).
    #EAGLE ?value temporal:min ("$start time of year$" "$end time of year$").
}
```

Query 4. Given the area location and radius, it detects the hottest month of that area in a given year

```
SELECT ?month (avg(?value) as ?avgTemp)
WHERE
{
    ?station a got:WeatherStation.
    ?station geo:hasGeometry ?geoFeature.
    ?geoFeature wgs84:geometry ?coor.
    #VOS FILTER (<bif:st_within>(?coor, <bif:st_point>($long$,$lat$),$radius$)).
    #EAGLE #RDF4J  ?geoFeature geo:sfWithin ($lat$ $long$ $radius$).
    #GraphDB ?geoFeature omgeo:within ($lat$ $long$ $radius$).
    #Stardog ?geoFeature geof:within ($lat$ $long$ $radius$).
    #Jena    ?geoFeature spatial:withinCircle ($lat$ $long$ $radius$).
    ?sensor sosa:isHostedBy ?station.
    ?sensor sosa:observes got:SurfaceTemperatureProperty.
    ?obs sosa:madebySensor  ?sensor;
            sosa:resultTime ?time;
            sosa:hasSimpleResult ?value.
    FILTER (year(?time)=$year$).
    #EAGLE ?value temporal:avg ("$start time of year$" "$end time of year$").
}
GROUP BY (month(?time) as ?month)
ORDER BY DESC (avg(?value)) limit 1
```

Query 5. Given the station URI and year, it retrieves the average wind speed for each month of year

```
SELECT ?month (avg(?value) as ?avgTemp)
WHERE
```

```
{
    ?sensor sosa:isHostedBy $station URI$.
    ?sensor sosa:observes got:WindSpeedProperty.
    ?obs sosa:madebySensor  ?sensor;
         sosa:resultTime ?time;
         sosa:hasSimpleResult ?value.
    FILTER (year(?time)=$year$).
    #EAGLE ?value temporal:avg ("$start time of year$" "$end time of year$").
}
GROUP BY (month(?time) as ?month)
ORDER BY ?month
```

Query 6. Given a date, it retrieves the total number of observation that were observed in California state

```
SELECT count(?obs) as ?number
WHERE {
    ?station a got:WeatherStation.
    ?station geo:hasGeometry ?geoFeature.
    ?geoFeature geoname:parentADM1 "California".
    ?geoFeature geoname:parentCountry "United States".
    ?sensor sosa:isHostedBy ?station.
    ?sensor sosa:observes got:SurfaceTemperatureProperty.
    ?obs sosa:madebySensor  ?sensor;
         sosa:resultTime    ?time.
    FILTER (year(?time)=$year$ && month(?time)=$month$ && day(?time)=$day$).
    #EAGLE ?count temporal:count ("$start time$" "$end time$").
}
```

Query 7. Given the latitude, longitude and radius , it retrieves the latest visibility observation value of that area

```
SELECT ?value ?time
WHERE
{
    ?station a got:WeatherStation.
    ?station geo:hasGeometry ?geoFeature.
    ?geoFeature wgs84:geometry ?coor.
    #VOS FILTER (<bif:st_within>(?coor,<bif:st_point>($long$,$lat$),$radius$)).
    #EAGLE #RDF4J  ?geoFeature geo:sfWithin ($lat$ $long$ $radius$).
    #GraphDB ?geoFeature omgeo:within ($lat$ $long$ $radius$).
    #Stardog ?geoFeature geof:within ($lat$ $long$ $radius$).
    #Jena    ?geoFeature spatial:withinCircle ($lat$ $long$ $radius$).
    ?sensor sosa:isHostedBy ?station.
    ?sensor sosa:observes got:AtmosphericVisibilityProperty.
    ?obs sosa:madebySensor  ?sensor;
          sosa:resultTime ?time;
          sosa:hasSimpleResult ?value.
```

```
    #EAGLE ?value temporal:values ("$current time$").
}
ORDER BY DESC (?time)
LIMIT 1
```

Query 8. Given a keyword, it retrieves all the places matching a keyword

```
SELECT ?station ?place ?sc
WHERE
{
    ?station a got:WeatherStation.
    ?station geo:hasGeometry ?geoFeature.
    ?geoFeature geoname:parentADM1 ?place.
    #VOS ?place bif:contains "'$keyword$*'" OPTION (score ?sc).
    #Stardog ?place <tag:stardog:api:property:textMatch> "'$keyword$*'".
    #RDF4J ?geoFeature text:matches [text:query '$keyword$*'].
    #EAGLE ?geoFeature text:match (geoname:parentADM1 "'$keyword$*'").
    #Jena (?geoFeature ?sc) text:query (geoname:parentADM1 "'$keyword$*'" ).
    #GraphDB ?place luc:myIndex "$keyword$*".
}ORDER BY ?sc
```

Query 9. Given a place name prefix, it summaries the number of observation of places that match a given keyword. The results are grouped by place and observed property

```
SELECT count(?obs) as ?totalNumber ?place ?observedType
WHERE
{
    ?station a got:WeatherStation.
    ?station geo:hasGeometry ?geoFeature.
    ?geoFeature geoname:parentCountry ?place.
    #VOS ?place bif:contains "'$name prefix$'*" OPTION (score ?sc).
    #EAGLE ?geoFeature text:match (geoname:parentCountry "'$name prefix$*'").
    #Stardog ?place <tag:stardog:api:property:textMatch> "'$name prefix$*'".
    #RDF4J ?geoFeature text:matches [text:query '$name prefix$*'].
    #Jena (?geoFeature ?sc) text:query (geoname:parentCountry "'$name prefix$*'" ).
    #GraphDB ?place luc:myIndex "$name prefix$*".
    ?sensor sosa:isHostedBy ?station.
    ?sensor sosa:observes ?observedType.
    ?obs sosa:madebySensor  ?sensor.
}GROUP BY ?place ?observedType
```

Query 10. Given a keyword, it retrieves the average humidity value for places that matches a keywords since 2013

```
SELECT avg(?value) as ?avgValue ?place
WHERE
{
```

```
    ?station a got:WeatherStation.
    ?station geo:hasGeometry ?geoFeature.
    ?geoFeature geoname:parentCountry ?place.
    #VOS ?place bif:contains "'$keyword$*'" OPTION (score ?sc).
    #EAGLE ?geoFeature text:match (geoname:parentCountry "'$keyword$*'").
    #Jena (?geoFeature ?sc) text:query (geo:parentCountry "'$keyword$*'" ).
    #Stardog ?place <tag:stardog:api:property:textMatch> "'$keyword$*'".
    #RDF4J ?geoFeature text:matches [text:query '$keyword$*'].
    #GraphDB ?place luc:myIndex "$keyword$*".
    ?sensor sosa:isHostedBy ?station.
    ?sensor sosa:observes got:AtmosphericPressureProperty.
    ?obs sosa:madebySensor  ?sensor;
            sosa:resultTime ?time;
            sosa:hasSimpleResult ?value.
    FILTER(year(?time)>=2013)
    #EAGLE ?value temporal:avg ("01/01/2013").
}GROUP BY ?place
```

Query 11. It retrieves the total number of sensor for each observed properties

```
SELECT (count(?sensor) as ?number) ?obsType
WHERE
{
    ?sensor sosa:observes ?obsType
}GROUP BY ?obsType
```

BIBLIOGRAPHY

[1] J. Aasman. "Allegro graph: RDF triple database". In: *Cidade: Oakland Franz Incorporated* 17 (2006).

[2] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. "Scalable semantic web data management using vertical partitioning". In: *Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment. 2007, pp. 411–422.

[3] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. "SW-Store: a vertically partitioned DBMS for Semantic Web data management". In: *The VLDB Journal* 18.2 (2009), pp. 385–406.

[4] B. Adida and M. Birbeck. "RDFa primer: Bridging the human and data webs". In: *Retrieved June* 20 (2008), p. 2008.

[5] G. Adomavicius and A. Tuzhilin. "Toward the Next Generation of Recommender Systems: A Survey of the State-of-the-Art and Possible Extensions". In: *IEEE Trans. on Knowl. and Data Eng.* 17.6 (June 2005). ISSN: 1041-4347.

[6] R. Agarwal, D. G. Fernandez, T. Elsaleh, A. Gyrard, J. Lanza, L. Sanchez, N. Georgantas, and V. Issarny. "Unified IoT ontology to enable interoperability and federation of testbeds". In: *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*. IEEE. 2016, pp. 70–75.

[7] M. Akdere et al. "Learning-based Query Performance Modeling and Prediction". In: *Proceedings of the 2012 IEEE ICDE*. 2012.

[8] J. F. Allen. "Maintaining knowledge about temporal intervals". In: *Readings in qualitative reasoning about physical systems*. Elsevier, 1990, pp. 361–372.

[9] N. S. Altman. "An introduction to kernel and nearest-neighbor nonparametric regression". In: *The American Statistician* 46 (1992).

[10] I. C. F. D. Analytics. *Linked Open Data Cloud*. https://lod-cloud.net.

[11] M. Arlitt. "Characterizing web user sessions". In: *ACM SIGMETRICS Performance Evaluation Review* 28.2 (2000), pp. 50–63.

[12] N. Aswani, K. Bontcheva, and H. Cunningham. "Mining information for instance unification". In: *International Semantic Web Conference*. Springer. 2006, pp. 329–342.

[13] G. Atemezing, O. Corcho, D. Garijo, J. Mora, M. Poveda-Villalón, P. Rozas, D. Vila-Suero, and B. Villazón-Terrazas. "Transforming meteorological data into linked data". In: *Semantic Web* 4.3 (2013), pp. 285–290.

[14] M. Atre, J. Srinivasan, and J. A. Hendler. "BitMat: A main memory RDF triple store". In: *Tetherless World Constellation, Rensselar Plytehcnic Institute, Troy NY* (2009).

[15] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives. "Dbpedia: A nucleus for a web of open data". In: *The semantic web*. Springer, 2007, pp. 722–735.

[16] S. Auer, J. Lehmann, and S. Hellmann. "Linkedgeodata: Adding a spatial dimension to the web of data". In: *International Semantic Web Conference*. Springer. 2009, pp. 731–746.

[17] P. Barnaghi, M. Presser, and K. Moessner. "Publishing linked sensor data". In: *CEUR Workshop Proceedings: Proceedings of the 3rd International Workshop on Semantic Sensor Networks (SSN)*. Vol. 668. 2010.

[18] S. Batsakis and E. G. Petrakis. "SOWL: a framework for handling spatio-temporal information in OWL 2.0". In: *International Workshop on Rules and Rule Markup Languages for the Semantic Web*. Springer. 2011, pp. 242–249.

[19] R. Battle and D. Kolas. "Geosparql: enabling a geospatial semantic web". In: *Semantic Web Journal* 3.4 (2011), pp. 355–370.

[20] P. Bellini and P. Nesi. "Performance assessment of rdf graph databases for smart city services". In: *Journal of Visual Languages & Computing* 45 (2018), pp. 24–38.

[21] T. Berners-Lee. "Linked data". In: (). URL: http://www.w3.org/DesignIssues/LinkedData.html.

[22] E. Bertino, B. Catania, and W. Q. Wang. "XJoin Index: Indexing XML Data for Efficient Handling of Branching Path Expressions". In: *Proceedings of the 20th International Conference on Data Engineering*. ICDE '04. Washington, DC, USA: IEEE Computer Society, 2004. ISBN: 0-7695-2065-0.

[23] B. Bishop, A. Kiryakov, D. Ognyanoff, I. Peikov, Z. Tashev, and R. Velkov. "OWLIM: A family of scalable semantic repositories". In: *Semantic Web* 2.1 (2011), pp. 33–42.

[24] C. Bizer and A. Schultz. "The berlin sparql benchmark". In: *International Journal on Semantic Web and Information Systems (IJSWIS)* 5.2 (2009), pp. 1–24.

[25] M. Botts. "Ogc implementation specification 07-000: Opengis sensor model language (sensorml)". In: *Open Geospatial Consortium: Wayland, MA, USA* (2007).

[26] M. Botts, G. Percivall, C. Reed, and J. Davidson. "OGC® sensor web enablement: Overview and high level architecture". In: *International conference on GeoSensor Networks*. Springer. 2006, pp. 175–190.

[27] D. Brickley, R. V. Guha, and B. McBride. "RDF vocabulary description language 1.0: RDF Schema. W3C Recommendation (2004)". In: *URL http://www. w3. org/tr/2004/rec-rdf-schema-20040210* (2004).

[28] A. Brodt, D. Nicklas, and B. Mitschang. "Deep integration of spatial query processing into native rdf triple stores." In: *Proceedings of 18th SIGSPATIAL International Conference*. ACM, 2010, pp. 33–42.

[29] A. Brodt, D. Nicklas, and B. Mitschang. "Deep integration of spatial query processing into native RDF triple stores". In: *Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM. 2010, pp. 33–42.

[30] J. Broekstra, A. Kampman, and F. Van Harmelen. "Sesame: A generic architecture for storing and querying rdf and rdf schema". In: *International semantic web conference*. Springer. 2002, pp. 54–68.

[31] I. Budak Arpinar, A. Sheth, C. Ramakrishnan, E. Lynn Usery, M. Azami, and M.-P. Kwan. "Geospatial ontology development and semantic analytics". In: *Transactions in GIS* 10.4 (2006), pp. 551–575.

[32] J. J. Carroll, C. Bizer, P. Hayes, and P. Stickler. "Named graphs". In: *Web Semantics: Science, Services and Agents on the World Wide Web* 3.4 (2005), pp. 247–267.

[33] P. A. Champin and A. Passant. "SIOC in action representing the dynamics of online communities". In: *Proceedings of the 6th International Conference on Semantic Systems*. ACM. 2010, p. 12.

[34] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. "Bigtable: A distributed storage system for structured data". In: *ACM Transactions on Computer Systems (TOCS)* 26.2 (2008), p. 4.

[35] A. Chebotko, S. Lu, and F. Fotouhi. "Semantics preserving SPARQL-to-SQL translation". In: *Data & Knowledge Engineering* 68.10 (2009), pp. 973–1000.

[36] H. Chen, T. Finin, and A. Joshi. "The SOUPA ontology for pervasive computing". In: *Ontologies for agents: Theory and experiences*. Springer, 2005, pp. 233–258.

[37] L. Cheng, S. Kotoulas, T. Ward, and G. Theodoropoulos. "Runtime characterisation of triple stores: An initial investigation". In: (2012).

[38] E. I. Chong, S. Das, G. Eadon, and J. Srinivasan. "An efficient SQL-based RDF querying scheme". In: *Proceedings of the 31st international conference on Very large data bases*. VLDB Endowment. 2005, pp. 1216–1227.

[39] E. Clementini, P. Di Felice, and P. Van Oosterom. "A small set of formal topological relationships suitable for end-user interaction". In: *International Symposium on Spatial Databases*. Springer. 1993, pp. 277–295.

[40] M. Compton et al. "The SSN ontology of the W3C semantic sensor network incubator group". In: *Web Semantics: Science, Services and Agents on the World Wide Web* 17 (2012), pp. 25–32.

[41] S. Cox. "Observations and measurements". In: *Open Geospatial Consortium Best Practices Document. Open Geospatial Consortium* (2006), p. 21.

[42] S. Cox. "Observations and measurements-xml implementation". In: (2011).

[43] O. Curé and G. Blin. *RDF database systems: triples storage and SPARQL query processing*. Morgan Kaufmann, 2014.

[44] R. Cyganiak, D. Reynolds, and J. Tennison. "The rdf data cube vocabulary, w3c working draft 05 april 2012". In: *World Wide Web Consortium* (2012).

[45] R. Cyganiak, D. Wood, M. Lanthaler, G. Klyne, J. J. Carroll, and B. McBride. "RDF 1.1 concepts and abstract syntax". In: *W3C recommendation* 25.02 (2014).

[46] C. J. Date, H. Darwen, and N. Lorentzos. *Temporal data & the relational model*. Elsevier, 2002.

[47] D. DeHaan and F. W. Tompa. "Optimal top-down join enumeration". In: *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. ACM. 2007, pp. 785–796.

[48] N. Dimiduk, A. Khurana, M. H. Ryan, and M. Stack. *HBase in action*. Manning Shelter Island, 2013.

[49] L. Dodds and I. Davis. *Linked Data patterns: A pattern catalogue for modelling, publishing, and consuming Linked Data*. L. Dodds, I. Davis, 2011.

[50] O. Erling and I. Mikhailov. "Virtuoso: RDF support in a native RDBMS". In: *Semantic Web Information Management*. Springer, 2010, pp. 501–519.

[51] D. Evans. "The internet of things: How the next evolution of the internet is changing everything". In: (2011).

[52] D. C. Faye, O. Cure, and G. Blin. "A survey of RDF storage approaches". In: *Revue Africaine de la Recherche en Informatique et Mathématiques Appliquées* 15 (2012), pp. 11–35.

[53] L. Feigenbaum, G. T. Williams, K. G. Clark, and E. Torres. "SPARQL 1.1 Protocol". In: *Recommendation, W3C, March* (2013).

[54] J. D. Fernández, J. Umbrich, A. Polleres, and M. Knuth. "Evaluating query and storage strategies for RDF archives". In: *Semantic Web* 10.2 (2019), pp. 247–291.

[55] F. Frasincar, G.-J. Houben, R. Vdovjak, and P. Barna. "RAL: An algebra for querying RDF". In: *World Wide Web* 7.1 (2004), pp. 83–109.

[56] A. Ganapathi et al. "Predicting Multiple Metrics for Queries: Better Decisions Enabled by Machine Learning". In: *Proceedings of the 2009 ICDE*. IEEE Computer Society, 2009, pp. 592–603. ISBN: 978-0-7695-3545-6.

[57] G. Garbis, K. Kyzirakos, and M. Koubarakis. "Geographica: A benchmark for geospatial rdf stores (long version)". In: *International Semantic Web Conference*. Springer. 2013, pp. 343–359.

[58] *Geohash*. https://en.wikipedia.org/wiki/Geohash.

[59] L. George. *HBase: the definitive guide: random access to your planet-size data*. " O'Reilly Media, Inc.", 2011.

[60] *GeoSPARQL Ontology*. URL: http://docs.opengeospatial.org/per/16-039r2.html.

[61] F. Goasdoué, Z. Kaoudi, I. Manolescu, J.-A. Quiané-Ruiz, and S. Zampetakis. "Cliquesquare: Flat plans for massively parallel RDF queries". In: *2015 IEEE 31st International Conference on Data Engineering*. IEEE. 2015, pp. 771–782.

[62] G. Graefe. "The cascades framework for query optimization". In: *IEEE Data Eng. Bull.* 18.3 (1995), pp. 19–29.

[63] G. Graefe and D. J. DeWitt. *The EXODUS optimizer generator*. Vol. 16. 3. ACM, 1987.

[64] G. Graefe and W. J. McKenna. "The volcano optimizer generator: Extensibility and efficient search". In: *ICDE*. Vol. 93. 1993, pp. 209–218.

[65] F. Grandi. "T-SPARQL: A TSQL2-like Temporal Query Language for RDF." In: *ADBIS (Local Proceedings)*. Citeseer. 2010, pp. 21–30.

[66] S. Groppe. *Data management and query processing in semantic web databases*. Springer Science & Business Media, 2011.

[67] T. R. Gruber. "A translation approach to portable ontology specifications". In: *Knowledge acquisition* 5.2 (1993), pp. 199–220.

[68] S. R. Gunn et al. "Support vector machines for classification and regression". In: *ISIS report* (1998).

[69] Y. Guo, Z. Pan, and J. Heflin. "LUBM: A benchmark for OWL knowledge base systems". In: *Web Semantics: Science, Services and Agents on the World Wide Web* 3.2-3 (2005), pp. 158–182.

[70] C. Gupta, A. Mehta, and U. Dayal. "PQR: Predicting query execution times for autonomous workload management". In: *2008 International Conference on Autonomic Computing*. IEEE. 2008, pp. 13–22.

[71] C. Gutierrez, C. A. Hurtado, and A. Vaisman. "Introducing time into rdf". In: *IEEE Transactions on Knowledge and Data Engineering* 19 (2007), pp. 207–218.

[72] C. Gutierrez, C. A. Hurtado, and A. Vaisman. "Introducing Time into RDF". In: *IEEE Trans. on Knowl. and Data Eng.* 19 (2 Feb. 2007), pp. 207–218. ISSN: 1041-4347. DOI: 10.1109/TKDE.2007.34. URL: http://portal.acm.org/citation.cfm?id=1191548.1191755.

[73] C. Gutierrez, C. Hurtado, and A. Vaisman. "Temporal rdf". In: *European Semantic Web Conference*. Springer. 2005, pp. 93–107.

[74] L. M. Haas, J. C. Freytag, G. M. Lohman, and H. Pirahesh. *Extensible query processing in Starburst*. Vol. 18. 2. ACM, 1989.

[75] F. Hakimpour, B. Aleman-Meza, M. Perry, and A. P. Sheth. "Data Processing in Space, Time, and Semantics Dimensions". In: (2006).

[76] S. Harris and N. Gibbins. "3store: Efficient bulk RDF storage". In: (2003).

[77] S. Harris and N. Shadbolt. "SPARQL query processing with conventional relational database systems". In: *International Conference on Web Information Systems Engineering*. Springer. 2005, pp. 235–244.

[78] A. Harth and S. Decker. *Yet another rdf store: Perfect index structures for storing semantic web data with contexts*. Tech. rep. DERI Technical Report, 2004.

[79] O. Hartig and R. Heese. "The SPARQL query graph model for query optimization". In: *European Semantic Web Conference*. Springer. 2007, pp. 564–578.

[80] Hartig, O, and R. Heese. "The SPARQL query graph model for query optimization". In: ESWC'07. Tenerife, Canary Islands, Spain, 2007. ISBN: 3-540-68233-3, 978-3-540-68233-2.

[81] R. Hasan and F. Gandon. "A Machine Learning Approach to SPARQL Query Performance Prediction". In: *Proceedings of the 2014 IEEE/WIC/ACM WI-IAT - Volume 01*. IEEE Computer Society, 2014, pp. 266–273.

[82] T. Heath and C. Bizer. "Linked data: Evolving the web into a global data space". In: *Synthesis lectures on the semantic web: theory and technology* 1.1 (2011), pp. 1–136.

[83] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. *Generalized search trees for database systems*. September, 1995.

[84] J. Herring. "OpenGIS Implementation Standard for Geographic information-Simple feature access-Part 1: Common architecture". In: *OGC Document* 4.21 (2011), pp. 122–127.

[85] J. R. Herring. "OpenGIS implementation specification for geographic information-Simple feature access-Part 1: Common architecture". In: *Open Geospatial Consortium* (2006), p. 95.

[86] J. R. Hobbs, G. Ferguson, J. Allen, P. Hayes, I. Niles, and A. Pease. *A daml ontology of time*. 2002.

[87] J. R. Hobbs and F. Pan. "Time ontology in OWL". In: *W3C working draft* 27 (2006), p. 133.

[88] R. Hodgson, P. J. Keller, J. Hodges, and J. Spivak. "QUDT-quantities, units, dimensions and data types ontologies". In: *USA Available http://qudt. org March* (2014).

[89] A. Hogan, J. Umbrich, A. Harth, R. Cyganiak, A. Polleres, and S. Decker. "An empirical survey of linked data conformance". In: *Web Semantics: Science, Services and Agents on the World Wide Web* 14 (2012), pp. 14–44.

[90] M. Husain, J. McGlothlin, M. M. Masud, L. Khan, and B. M. Thuraisingham. "Heuristics-based query processing for large RDF graphs using cloud computing". In: *IEEE Transactions on Knowledge and Data Engineering* 23.9 (2011), pp. 1312–1327.

[91] A. Ibrahim, F. Carrez, and K. Moessner. "Geospatial ontology-based mission assignment in Wireless Sensor Networks". In: *2015 International Conference on Recent Advances in Internet of Things (RIoT)*. IEEE. 2015, pp. 1–6.

[92] A. Jaffri, H. Glaser, and I. Millard. "Uri disambiguation in the context of linked data". In: (2008).

[93] K. Janowicz, A. Haller, S. J. Cox, D. Le Phuoc, and M. Lefrançois. "SOSA: A lightweight ontology for sensors, observations, samples, and actuators". In: *Journal of Web Semantics* (2018).

[94] K. Janowicz, A. Haller, S. J. Cox, D. Le Phuoc, and M. Lefrançois. "SOSA: A lightweight ontology for sensors, observations, samples, and actuators". In: *Journal of Web Semantics* 56 (2019), pp. 1–10.

[95] *Jena Assembler Description*. http://jena.apache.org/documentation/assembler/assembler-howto.html.

[96] A. Jena. *API [May 15, 2013]*.

[97] Y. Katz and B. C. Grau. "Representing qualitative spatial information in OWL DL". In: (2005).

[98] L. Kaufman and P. Rousseeuw. *Clustering by means of medoids*. North-Holland, 1987.

[99] R. Khare and T. Çelik. "Microformats: a pragmatic path to the semantic web". In: *Proceedings of the 15th international conference on World Wide Web*. ACM. 2006, pp. 865–866.

[100] K. Kim, B. Moon, and H.-J. Kim. "R3F: RDF triple filtering method for efficient SPARQL query processing". In: *World Wide Web* 18.2 (2015), pp. 317–357.

[101] A. Kiryakov, D. Ognyanov, and D. Manov. "OWLIM - A pragmatic semantic repository for OWL". In: *Proceedings of WISE'05*. Springer, 2005.

[102] A. Kiryakov, D. Ognyanov, and D. Manov. "OWLIM–a pragmatic semantic repository for OWL". In: *International Conference on Web Information Systems Engineering*. Springer. 2005, pp. 182–192.

[103] P. Klinov. *HOW TO READ STARDOG QUERY PLANS*. https://www.stardog.com/blog/how-to-read-stardog-query-plans/.

[104] G. Klyne and J. J. Carroll. "Resource description framework (RDF): Concepts and abstract syntax". In: (2006).

[105] D. Kolas. "A benchmark for spatial semantic web systems". In: *International Workshop on Scalable Semantic Web Knowledge Base Systems*. 2008.

[106] M. Koubarakis and K. Kyzirakos. "Modeling and querying metadata in the semantic sensor web: the model strdf and the query language stsparql". In: *Proc. ESWC*. Vol. 12. 2010, pp. 425–439.

[107] M. Koubarakis and K. Kyzirakos. "Modeling and querying metadata in the semantic sensor web: The model stRDF and the query language stSPARQL". In: *Extended Semantic Web Conference*. Springer. 2010, pp. 425–439.

[108] K. Kyzirakos et al. "Strabon: a semantic geospatial DBMS". In: *ISWC*. Springer. 2012.

[109] G. Ladwig and A. Harth. "CumulusRDF: linked data management on nested key-value stores". In: *The 7th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2011)*. Vol. 30. 2011.

[110] D. Le Phuoc, H. Nguyen Mau Quoc, T. Tran Nhat, H. Ngo Quoc, and M. Hauswirth. "Enabling Live Exploration on The Graph of Things". In: *Proceedings of the Semantic Web Challenge*. 2014.

[111] L. Lefort, J. Bobruk, A. Haller, K. Taylor, and A. Woolf. "A linked sensor data cube for a 100 year homogenised daily temperature dataset". In: *Proceedings of the 5th International Conference on Semantic Sensor Networks-Volume 904*. CEUR-WS. org. 2012, pp. 1–16.

[112] L. Lefort, A. Haller, K. Taylor, G. Squire, P. Taylor, D. Percival, and A. Woolf. "The ACORN-SAT linked climate dataset". In: *Semantic Web* 8.6 (2017), pp. 959–967.

[113] J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P. N. Mendes, S. Hellmann, M. Morsey, P. Van Kleef, S. Auer, et al. "DBpedia–a large-scale, multilingual knowledge base extracted from Wikipedia". In: *Semantic Web* 6.2 (2015), pp. 167–195.

[114] A. Liaw et al. "Classification and regression by Random Forest". In: *R news* 2.3 (2002).

[115] B. Liu and B. Hu. "An evaluation of rdf storage systems for large data applications". In: *2005 First International Conference on Semantics, Knowledge and Grid*. IEEE. 2005, pp. 59–59.

[116] P. A. Longley, M. F. Goodchild, D. J. Maguire, and D. W. Rhind. *Geographic information systems and science*. John Wiley & Sons, 2005.

[117] J. N. Lott and R. Baldwin. "6.2 THE FCC INTEGRATED SURFACE HOURLY DATABASE, A NEW RESOURCE OF GLOBAL CLIMATE DATA". In: (2001).

[118] A. Maduko, K. Anyanwu, A. Sheth, and P. Schliekelman. "Estimating the cardinality of RDF graph patterns". In: *Proceedings of the 16th international conference on World Wide Web*. ACM. 2007, pp. 1233–1234.

[119] P. Marcel and E. Negre. "A survey of query recommendation techniques for data warehouse exploration." In: *EDA*. 2011, pp. 119–134.

[120] A. Matono, S. M. Pahlevi, and I. Kojima. "RDFCube: A P2P-based three-dimensional index for structural joins on distributed triple stores". In: *Databases, Information Systems, and Peer-to-Peer Computing*. Springer, 2006, pp. 323–330.

[121] B. McBride. "The resource description framework (RDF) and its vocabulary description language RDFS". In: *Handbook on ontologies*. Springer, 2004, pp. 51–65.

[122] D. L. McGuinness, F. Van Harmelen, et al. "OWL web ontology language overview". In: *W3C recommendation* 10.10 (2004), p. 2004.

[123] A. Miles and S. Bechhofer. "SKOS simple knowledge organization system reference". In: *W3C recommendation* 18 (2009), W3C.

[124] L. Miller. "Inkling: an RDF SquishQL implementation". In: (). URL: http://swordfish.rdfweb.org/rdfquery..

[125] G. Moerkotte and T. Neumann. "Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products". In: *Proceedings of the 32nd international conference on Very large data bases*. VLDB Endowment. 2006, pp. 930–941.

[126] D. R. Montello and A. U. Frank. "Modeling directional knowledge and reasoning in environmental space: Testing qualitative metrics". In: *The construction of cognitive maps*. Springer, 1996, pp. 321–344.

[127] J. Moore et al. "Data center workload monitoring, analysis, and emulation". In: *8th CAECW*. 2005, pp. 1–8.

[128] M. Morsey et al. "DBpedia SPARQL benchmark-performance assessment with real queries on real data". In: *ISWC*. Springer. 2011, pp. 454–469.

[129] M. Muralikrishna and D. DeWitt. "Equi-depth histograms for estimating selectivity factors for multi-dimensional". In: *Proc. Int'l Conf. Management of Data*. 1988.

[130] H. Naacke, B. Amann, and O. Curé. "SPARQL Graph Pattern Processing with Apache Spark". In: *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems*. GRADES'17. Chicago, IL, USA: ACM, 2017, 1:1–1:7. ISBN: 978-1-4503-5038-9. DOI: 10.1145/3078447.3078448. URL: http://doi.acm.org/10.1145/3078447.3078448.

[131] T. Neumann and G. Weikum. "RDF-3X: a RISC-style engine for RDF". In: *Proceedings of the VLDB Endowment* 1.1 (2008), pp. 647–659.

[132] T. Neumann and G. Weikum. "Scalable join processing on very large RDF graphs". In: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. ACM. 2009, pp. 627–640.

[133] T. Neumann and G. Weikum. "The RDF-3X engine for scalable management of RDF data". In: *The VLDB Journal* 19 (1 Feb. 2010), pp. 91–113. ISSN: 1066-8888. DOI: http://dx.doi.org/10.1007/s00778-009-0165-y. URL: http://dx.doi.org/10.1007/s00778-009-0165-y.

[134] T. Neumann and G. Weikum. "The RDF-3X engine for scalable management of RDF data". In: *The VLDB Journal—The International Journal on Very Large Data Bases* 19.1 (2010), pp. 91–113.

[135] H. Nguyen Mau Quoc, D. Le Phuoc, M. Serrano, and M. Hauswirth. "Super Stream Collider". In: *Proceedings of the Semantic Web Challenge*. 2012.

[136] I. Niles and A. Pease. "Towards a standard upper ontology". In: *Proceedings of the international conference on Formal Ontology in Information Systems-Volume 2001*. ACM. 2001, pp. 2–9.

[137] A. Owens, A. Seaborne, N. Gibbins, and mc schraefel. "Clustered TDB: A Clustered Triple Store for Jena". Nov. 2008. URL: http://eprints.soton.ac.uk/266974/.

[138] Z. Pan and J. Heflin. *Dldb: Extending relational databases to support semantic web queries*. Tech. rep. LEHIGH UNIV BETHLEHEM PA DEPT OF COMPUTER SCIENCE and ELECTRICAL ENGINEERING, 2004.

[139] H. Patni, C. Henson, and A. Sheth. "Linked sensor data". In: *Collaborative Technologies and Systems (CTS), 2010 International Symposium on*. IEEE. 2010, pp. 362–370.

[140] J. Pérez, M. Arenas, and C. Gutierrez. "Semantics and Complexity of SPARQL". In: *International semantic web conference*. Springer. 2006, pp. 30–43.

[141] M. Perry, P. Jain, and A. P. Sheth. "Sparql-st: extending sparql to support spatiotemporal queries". In: *Journal of Geospatial Semantics and the Semantic Web* 12 (2011).

[142] M. S. Perry. "A framework to support spatial, temporal and thematic analytics over semantic web data". In: (2008).

[143] M. Perry, F. Hakimpour, and A. Sheth. "Analyzing theme, space, and time: an ontology-based approach". In: *Proceedings of the 14th annual ACM international symposium on Advances in geographic information systems*. ACM. 2006, pp. 147–154.

[144] M. Perry and J. Herring. "OGC GeoSPARQL-A geographic query language for RDF data". In: *OGC implementation standard* (2012).

[145] M. Perry, A. P. Sheth, F. Hakimpour, and P. Jain. "Supporting complex thematic, spatial and temporal queries over semantic web data". In: *International Conference on GeoSpatial Sematics*. Springer. 2007, pp. 228–246.

[146] D. Le-Phuoc et al. "The Graph of Things: A step towards the Live Knowledge Graph of connected things". In: *Journal of Web Semantics* 37 (2016).

[147] D. Le-Phuoc, H. Q. Nguyen-Mau, J. X. Parreira, and M. Hauswirth. "A middleware framework for scalable management of linked streams". In: *Web Semantics: Science, Services and Agents on the World Wide Web* 16 (2012), pp. 42–51.

[148] D. Le-Phuoc, J. X. Parreira, M. Hausenblas, Y. Han, and M. Hauswirth. "Live linked open sensor database". In: *Proceedings of the 6th International Conference on Semantic Systems*. New York, NY, USA: ACM, 2010. ISBN: 978-1-4503-0014-8.

[149] D. Le-Phuoc, H. N. M. Quoc, Q. H. Ngo, T. T. Nhat, and M. Hauswirth. "Enabling Live Exploration on The Graph of Things". In: *Proceedings of the Semantic Web Challenge 2014, Organised in conjunction with the International Semantic Web Conference* (2014).

[150] D. Le-Phuoc, H. N. M. Quoc, J. X. Parreira, and M. Hauswirth. "The linked sensor middleware–connecting the real world and the semantic web". In: *Proceedings of the Semantic Web Challenge* 152 (2011), pp. 22–23.

[151] V. Poosala, P. J. Haas, Y. E. Ioannidis, and E. J. Shekita. "Improved histograms for selectivity estimation of range predicates". In: *ACM Sigmod Record*. Vol. 25. 2. ACM. 1996, pp. 294–305.

[152] A. Pugliese, O. Udrea, and V. Subrahmanian. "Scaling RDF with time". In: *Proceedings of the 17th international conference on World Wide Web*. 2008, pp. 605–614.

[153] R. Punnoose, A. Crainiceanu, and D. Rapp. "Rya: a scalable RDF triple store for the clouds". In: *Proceedings of the 1st International Workshop on Cloud Intelligence*. ACM. 2012, p. 4.

[154] H. N. M. Quoc and D. Le Phuoc. "An elastic and scalable spatiotemporal query processing for linked sensor data". In: *Proceedings of the 11th International Conference on Semantic Systems*. ACM. 2015, pp. 17–24.

[155] H. N. M. Quoc, M. Serrano, J. G. Breslin, and D. L. Phuoc. "A learning approach for query planning on spatio-temporal IoT data". In: *Proceedings of the 8th International Conference on the Internet of Things*. ACM. 2018, p. 1.

[156] H. N. M. Quoc, M. Serrano, N. M. Han, J. G. Breslin, and D. L. Phuoc. "A Performance Study of RDF Stores for Linked Sensor Data". Under reviewed. 2019. URL: http://www.semantic-web-journal.net/content/performance-study-rdf-stores-linked-sensor-data.

[157] H. N. M. Quoc, M. Serrano, N. M. Han, J. G. Breslin, and D. L. Phuoc. "Global Weather Sensor Dataset". Under reviewed. URL: http://www.semantic-web-journal.net/content/global-weather-sensor-dataset.

[158] D. A. Randell, Z. Cui, and A. G. Cohn. "A spatial logic based on regions and connection." In: *KR* 92 (1992), pp. 165–176.

[159] J. Renz and B. Nebel. "Qualitative spatial reasoning using constraint calculi". In: *Handbook of spatial logics*. Springer, 2007, pp. 161–215.

[160] K. Riesen et al. "A novel software toolkit for graph edit distance computation". In: *GbRPR*. Springer. 2013.

[161] K. Riesen and H. Bunke. "Approximate graph edit distance computation by means of bipartite graph matching". In: *Image and Vision computing* 27 (2009).

[162] K. Rohloff, M. Dean, I. Emmons, D. Ryder, and J. Sumner. "An evaluation of triple-store technologies for large data stores". In: *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*. Springer. 2007, pp. 1105–1114.

[163] D. J. Russomanno, C. R. Kothari, and O. A. Thomas. "Building a Sensor Ontology: A Practical Approach Leveraging ISO and OGC Models." In: *IC-AI*. 2005, pp. 637–643.

[164] M. Saleem. *Efficient Source Selection and Benchmarking for SPARQL Endpoint Query Federation*. IOS Press, 2018.

[165] A. Schätzle, M. Przyjaciel-Zablocki, T. Berberich, and G. Lausen. "S2X: Graph-Parallel Querying of RDF with GraphX". In: *Biomedical Data Management and Graph Online Querying*. Ed. by F. Wang, G. Luo, C. Weng, A. Khan, P. Mitra, and C. Yu. Cham: Springer International Publishing, 2016, pp. 155–168. ISBN: 978-3-319-41576-5.

[166] A. Schätzle, M. Przyjaciel-Zablocki, T. Berberich, and G. Lausen. "S2X: graph-parallel querying of RDF with GraphX". In: *Biomedical Data Management and Graph Online Querying*. Springer, 2015, pp. 155–168.

[167] A. Schätzle, M. Przyjaciel-Zablocki, S. Skilevic, and G. Lausen. "S2RDF: RDF Querying with SPARQL on Spark". In: *Proc. VLDB Endow.* 9.10 (June 2016), pp. 804–815. ISSN: 2150-8097. DOI: 10.14778/2977797.2977806. URL: http://dx.doi.org/10.14778/2977797.2977806.

[168] M. Schmachtenberg, C. Bizer, and H. Paulheim. "State of the LOD Cloud 2014". In: *University of Mannheim, Data and Web Science Group. August* 30 (2014).

[169] M. Schmidt, T. Hornung, N. Küchlin, G. Lausen, and C. Pinkel. "An experimental comparison of RDF data management approaches in a SPARQL benchmark scenario". In: *International Semantic Web Conference*. Springer. 2008, pp. 82–97.

[170] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. "SP^ 2Bench: a SPARQL performance benchmark". In: *2009 IEEE 25th International Conference on Data Engineering*. IEEE. 2009, pp. 222–233.

[171] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. "Access path selection in a relational database management system". In: *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*. ACM. 1979, pp. 23–34.

[172] N. Shadbolt, T. Berners-Lee, and W. Hall. "The semantic web revisited". In: *IEEE intelligent systems* 21.3 (2006), pp. 96–101.

[173] R. Shaw, R. Troncy, and L. Hardman. "Lode: Linking open descriptions of events". In: *Asian semantic web conference*. Springer. 2009, pp. 153–167.

[174] A. P. Sheth. "Semantic sensor web". In: (2008).

[175] A. Sheth, C. Henson, and S. S. Sahoo. "Semantic Sensor Web". In: *IEEE Internet Computing* (4 2008). ISSN: 1089-7801. DOI: 10.1109/MIC.2008.87. URL: http://portal.acm.org/citation.cfm?id=1444383.1444435.

[176] A. Sheth and M. Perry. "Traveling the semantic web through space, time, and theme". In: *IEEE Internet Computing* 12.2 (2008), pp. 81–86.

[177] B. Sigoure. "Opentsdb scalable time series database (tsdb)". In: *Stumble Upon* (2012).

[178] M. Sintek and M. Kiesel. "RDFBroker: A signature-based high-performance RDF store". In: *European Semantic Web Conference*. Springer. 2006, pp. 363–377.

[179] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz. "Pellet: A practical owl-dl reasoner". In: *Web Semantics: science, services and agents on the World Wide Web* 5.2 (2007), pp. 51–53.

[180] D. Smiley. "Geospatial Search Using Geohash Prefixes [Z]". In: *Open Source Search Conference*. 2011, pp. 22–22.

[181] R. T. Snodgrass. *The TSQL2 temporal query language*. Vol. 330. Springer Science & Business Media, 2012.

[182] J. Soldatos, N. Kefalakis, M. Hauswirth, M. Serrano, J.-P. Calbimonte, M. Riahi, K. Aberer, P. P. Jayaraman, A. Zaslavsky, I. P. Žarko, et al. "Openiot: Open source internet-of-things in the cloud". In: *Interoperability and open-source solutions for the internet of things*. Springer, 2015, pp. 13–25.

[183] F. Song and O. Corby. "Extended Query Pattern Graph and Heuristics-based SPARQL Query Planning". In: *Procedia Computer Science* 60 (2015), pp. 302–311.

[184] A. Sotnykova, C. Vangenot, N. Cullot, N. Bennacer, and M.-A. Aufaure. "Semantic mappings in description logics for spatio-temporal database schema integration". In: *Journal on Data Semantics III*. Springer, 2005, pp. 143–167.

[185] C. Stadler, J. Lehmann, K. Höffner, and S. Auer. "Linkedgeodata: A core for a web of spatial open data". In: *Semantic Web* 3.4 (2012), pp. 333–354.

[186] M. Stocker et al. "SPARQL basic graph pattern optimization using selectivity estimation". In: *WWW '08*. Beijing, China, 2008, pp. 595–604. ISBN: 978-1-60558-085-2.

[187] M. Stocker and E. Sirin. "PelletSpatial: A Hybrid RCC-8 and RDF/OWL Reasoning and Query Engine." In: *OWLED*. Vol. 529. Citeseer. 2009.

[188] C. Tao, W.-Q. Wei, H. R. Solbrig, G. Savova, and C. G. Chute. "CNTRO: a semantic web ontology for temporal relation inferencing in clinical narratives". In: *AMIA annual symposium proceedings*. Vol. 2010. American Medical Informatics Association. 2010, p. 787.

[189] J. Tappolet and A. Bernstein. "Applied temporal RDF: Efficient temporal querying of RDF data with SPARQL". In: *European Semantic Web Conference*. Springer. 2009, pp. 308–322.

[190] *TDB Optimizer*. https://jena.apache.org/documentation/tdb/optimizer.html.

[191] *The Linked Open Data Cloud*. URL: https://lod-cloud.net.

[192] P. Tsialiamanis, L. Sidirourgos, I. Fundulaki, V. Christophides, and P. Boncz. "Heuristics-based query optimisation for SPARQL". In: *Proceedings of the 15th International Conference on Extending Database Technology*. ACM. 2012, pp. 324–335.

[193] "UniProt: the universal protein knowledgebase". In: *Nucleic acids research* 45.D1 (2016), pp. D158–D169.

[194] C. Villalonga, M. Bauer, V. Huang, J. Bernat, and P. Barnaghi. "Modeling of sensor data and context for the real world internet". In: *2010 8th IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops)*. IEEE. 2010, pp. 1–6.

[195] W3C. *SPARQL query language for RDF*. http://www.w3.org/TR/rdf-sparql-query/.

[196] C. Weiss, P. Karras, and A. Bernstein. "Hexastore: sextuple indexing for semantic web data management". In: *Proceedings of the VLDB Endowment* 1.1 (2008), pp. 1008–1019.

[197] C. Welty, R. Fikes, and S. Makarios. "A reusable ontology for fluents in OWL". In: *FOIS*. Vol. 150. 2006, pp. 226–236.

[198] M. Wick. *GeoNames*. GeoNames, 2006.

[199] K. Wilkinson and K. Wilkinson. *Jena property table implementation*. 2006.

[200] R. M. Yoo et al. "Constructing a non-linear model with neural networks for workload characterization". In: *Workload Characterization, 2006 IEEE International Symposium on*. IEEE. 2006, pp. 150–159.

[201] L. Yu, Y. Liu, and J. Lee. "SSTDE: an open source semantic spatiotemporal data engine for sensor web". In: *ACM SIGSPATIAL Workshop*. ACM. 2012.

[202] W. Zhang et al. "Learning-based SPARQL Query Performance Prediction". In: *WISE*. Springer. 2016.

[203] Q. Zhou and R. Fikes. "A reusable time ontology". In: *Proceeding of the AAAI Workshop on Ontologies for the Semantic Web*. 2002.

[204] L. Zou, J. Mo, L. Chen, M. T. Özsu, and D. Zhao. "gStore: answering SPARQL queries via subgraph matching". In: *Proceedings of the VLDB Endowment* 4.8 (2011), pp. 482–493.