Compact Semantic Representations of Observational Data

Von der Fakultät für Elektrotechnik und Informatik
der Gottfried Wilhelm Leibniz Universität Hannover
zur Erlangung des akademischen Grades

Doktorin der Naturwissenschaften
Dr. rer. nat.

genehmigte Dissertation von

MSIT Farah Karim
geboren am 01.02.1983
in Mirpur AJK, Pakistan

2020

Referent: Prof. Dr. Sören Auer, Leibniz Universität Hannover
Korreferent: Prof. Vojtêch Svátek, University of Economics, Prague
Tag der Promotion: 18.03.2020

Printed and published with the support of the German Academic Exchange Service.

“Surround yourself with the dreamers and the doers, the believers and the thinkers, but most of all, surround yourself with those who see the greatness within you, even when you don’t see it yourself.”

— Edmund Lee

For my sister late Khalida Kanwal, father-in-law late Ahmed Ud Din, and my parents.

Acknowledgements

First and foremost, I would like to thank God, the almighty and omniscient, for giving me the strength, knowledge, ability and opportunity to undertake this research study and to persevere and complete it satisfactorily. Without his blessings and support, this achievement would not have been possible.

The path toward this dissertation has been circuitous. Its completion is thanks in large part to special people who challenged, supported, and stuck with me along the way. I would like to thank Prof. Dr. Sören Auer not only for giving me a chance to pursue the Ph.D. degree in Germany, but also for the continuous support of my Ph.D. study and research, for his patience, motivation, and immense knowledge. I appreciate all his contributions of time, ideas, and manifold conversations on an intellectual and personal level, which I will keep in remembrance. I could not have imagined having a better advisor and mentor for my doctoral studies. My sincere thanks also go to Prof. Dr. Maria-Esther Vidal. The door of Prof. Vidal's office was always open whenever I ran into a trouble spot or had a question about my research or writing. She consistently allowed this thesis to be my own work, but whenever she thought her support was required, steered me into the right direction. Without her precious support, it would not be possible to conduct this research. Thank you Prof. Vidal for being a consistent source of inspiration for me with the enthusiasm you have for research and endless patience. I thank Philipp Rohde for being a nice colleague and helping me in addressing the details regarding any kind of documentation required during my Ph.D. studies. I would like to thank Kemele for all his academic support. Also, I thank Maria Isabel, Samaneh, Lucie-Aimée, Ariam, Ahmad, Guillermo, Mohammad, and Akhilesh with whom I have had the pleasure to work. I wish to extend my special thanks to Katja Bartel for helping me enormously, especially with the mammoth administrative tasks.

A special thanks to my family. Words cannot express how grateful I am to my father-in-law late Ahmed Ud Din Qureshi, mother-in law Alam Bibi, my mother Irshad Begum, and father Abdul Karim Qureshi for all of the sacrifices that they've made on my behalf. Their prayer for me was what sustained me thus far. I thank my siblings and niblings for supporting me spiritually throughout writing this thesis and my life in general. Last, but not least, I owe my deepest gratitude towards my better half, my amazing husband Waheed Ahmed Qureshi, for his eternal support and understanding of my goals and aspirations. His patience and cheerfulness will remain my inspiration throughout my life. Moreover, I would like to thank him for the care of our daughter Bismah Qureshi during the time of my doctoral studies.

Farah Karim

Abstract

The Internet of Things (IoT) concept has been widely adopted in several domains to enable devices to interact with each other and perform certain tasks. IoT devices encompass different concepts, e.g., sensors, programs, computers, and actuators. IoT devices observe their surroundings to collect information and communicate with each other in order to perform mutual tasks. These devices continuously generate observational data streams, which become historical data when these observations are stored. Due to an increase in the number of IoT devices, a large amount of streaming and historical observational data is being produced. Moreover, several ontologies, like the Semantic Sensor Network (SSN) Ontology, have been proposed for semantic annotation of observational data-either streams or historical. Resource Description Framework (RDF) is widely adopted data model to semantically describe the datasets. Semantic annotation provides a shared understanding for processing and analysis of observational data. However, adding semantics, further increases the data size especially when the observation values are redundantly sensed by several devices. For example, several sensors can generate observations indicating the same value for relative humidity in a given timestamp and city. This situation can be represented in an RDF graph using four RDF triples where observations are represented as triples that describe the observed phenomenon, the unit of measurement, the timestamp, and the coordinates. The RDF triples of an observation are associated with the same subject. Such observations share the same objects in a certain group of properties, i.e., they match star patterns composed of these properties and objects. In case the number of these subject entities or properties in these star patterns is large, the size of the RDF graph and query processing are negatively impacted; we refer these star patterns as *frequent star patterns.* This thesis addresses the problem of identifying *frequent star patterns* in RDF graphs and develop computational methods to identify frequent star patterns and generate a *factorized RDF graph* where the number of frequent star patterns is minimized. Furthermore, we apply these factorized RDF representations over historical semantic sensor data described using the SSN ontology and present tabular-based representations of factorized semantic sensor data in order to exploit Big Data frameworks. In addition, this thesis devises a knowledge-driven approach named DESERT that is able to on-Demand factorizE and Semantically Enrich stReam daTa. We evaluate the performance of our proposed techniques on several RDF graph benchmarks. The outcomes show that our techniques are able to effectively and efficiently detect frequent star patterns and RDF graph size can be reduced by up to 66.56% while data represented in the original RDF graph is preserved. Moreover, the compact representations are able to reduce the number of RDF triples by at least 53.25% in historical observational data and upto 94.34% in observational data streams. Additionally, query evaluation results over historical data reduce query execution time by up to three orders of magnitude. In observational data streams the size of the data required to answer the query is reduced by 92.53% reducing the memory space requirements to answer the queries. These results provide evidence that IoT data can be efficiently represented using the proposed compact representations, reducing thus, the negative impact that semantic annotations may have on IoT data management.

Keywords *Internet of Things, Stream Data, Semantic Web, Linked Data, RDF Compaction, Semantic Enrichment*

Zusammenfassung

Das Konzept des Internet der Dinge (IoT) ist in mehreren Bereichen weit verbreitet, damit Geräte miteinander interagieren und bestimmte Aufgaben erfüllen können. IoT-Geräte umfassen verschiedene Konzepte, z.B. Sensoren, Programme, Computer und Aktoren. IoT-Geräte beobachten ihre Umgebung, um Informationen zu sammeln und miteinander zu kommunizieren, um gemeinsame Aufgaben zu erfüllen. Diese Vorrichtungen erzeugen kontinuierlich Beobachtungsdatenströme, die zu historischen Daten werden, wenn diese Beobachtungen gespeichert werden. Durch die Zunahme der Anzahl der IoT-Geräte wird eine große Menge an Streaming- und historischen Beobachtungsdaten erzeugt. Darüber hinaus wurden mehrere Ontologien, wie die Semantic Sensor Network (SSN) Ontologie, für die semantische Annotation von Beobachtungsdaten vorgeschlagen - entweder Stream oder historisch. Das Resource Description Framework (RDF) ist ein weit verbreitetes Datenmodell zur semantischen Beschreibung der Datensätze. Semantische Annotation bietet ein gemeinsames Verständnis für die Verarbeitung und Analyse von Beobachtungsdaten. Durch das Hinzufügen von Semantik wird die Datengröße jedoch weiter erhöht, insbesondere wenn die Beobachtungswerte von mehreren Geräten redundant erfasst werden. So können beispielsweise mehrere Sensoren Beobachtungen erzeugen, die den gleichen Wert für die relative Luftfeuchtigkeit in einem bestimmten Zeitstempel und einer bestimmten Stadt anzeigen. Diese Situation kann in einem RDF-Graph mit vier RDF-Tripel dargestellt werden, wobei Beobachtungen als Tripel dargestellt werden, die das beobachtete Phänomen, die Maßeinheit, den Zeitstempel und die Koordinaten beschreiben. Die RDF-Tripel einer Beobachtung sind mit dem gleichen Thema verbunden. Solche Beobachtungen teilen sich die gleichen Objekte in einer bestimmten Gruppe von Eigenschaften, d.h. sie entsprechen einem Sternmuster, das sich aus diesen Eigenschaften und Objekten zusammensetzt. Wenn die Anzahl dieser Subjektentitäten oder Eigenschaften in diesen Sternmustern groß ist, wird die Größe des RDF-Graphen und der Abfrageverarbeitung negativ beeinflusst; wir bezeichnen diese Sternmuster als *häufige Sternmuster*. Diese Arbeit befasst sich mit dem Problem der Identifizierung von *häufigen Sternenmustern* in RDF-Graphen und entwickelt Berechnungsmethoden, um häufige Sternmuster zu identifizieren und ein *faktorisiertes RDF-Graph* zu erzeugen, bei dem die Anzahl der häufigen Sternmuster minimiert wird. Darüber hinaus wenden wir diese faktorisierten RDF-Darstellungen über historische semantische Sensordaten an, die mit der SSN-Ontologie beschrieben werden, und präsentieren tabellarische Darstellungen von faktorisierten semantischen Sensordaten, um Big Data-Frameworks auszunutzen. Darüber hinaus entwickelt diese Arbeit einen wissensbasierten Ansatz namens DESERT, der in der

Lage ist, bei Bedarf Streamdaten zu faktorisieren und semantisch anzureichern (on-Demand factorizE and Semantically Enrich stReam daTa). Wir bewerten die Leistung unserer vorgeschlagenen Techniken anhand mehrerer RDF-Graph-Benchmarks. Die Ergebnisse zeigen, dass unsere Techniken in der Lage sind, häufige Sternmuster effektiv und effizient zu erkennen, und die Größe der RDF-Graphen kann um bis zu $66,56\%$ reduziert werden, während die im ursprünglichen RDF-Graph dargestellten Daten erhalten bleiben. Darüber hinaus sind die kompakten Darstellungen in der Lage, die Anzahl der RDF-Tripel um mindestens $53,25\%$ in historischen Beobachtungsdaten und bis zu $94,34\%$ in Beobachtungsdatenströmen zu reduzieren. Darüber hinaus reduzieren die Ergebnisse der Anfrageauswertung über historische Daten die Ausführungszeit der Anfrage um bis zu drei Größenordnungen. In Beobachtungsdatenströmen wird die Größe der zur Beantwortung der Anfrage benötigten Daten um $92,53\%$ reduziert, wodurch der Speicherplatzbedarf zur Beantwortung der Anfragen reduziert wird. Diese Ergebnisse belegen, dass IoT-Daten mit den vorgeschlagenen kompakten Darstellungen effizient dargestellt werden können, wodurch die negativen Auswirkungen semantischer Annotationen auf das IoT-Datenmanagement reduziert werden.
Schlagwörter: *Internet der Dinge, Streamdaten, Semantic Web, Verknüpfte Daten, RDF-Verdichtung, Semantische Anreicherung*

Contents

1 Introduction 1
1.1 Motivation 3
1.2 Existing Approaches 4
1.3 Problem Statement and Challenges 6
1.4 Research Questions 9
1.5 Thesis Overview 11
1.5.1 Contributions 11
1.5.2 List of Publications 14
1.6 Thesis Structure 15
1.7 Summary 16

2 Background 19
2.1 Data Integration System 19
2.1.1 Local-as-View (LAV) 20
2.1.2 Global-as-View (GAV) 21
2.1.3 IoT Data Integration 21
2.2 Semantic Web 22
2.2.1 The Semantic Sensor Network (SSN) Ontology 23
2.2.2 The Resource Description Framework (RDF) 23
2.2.3 RDF Schema 25
2.2.4 The SPARQL Query Language and SPARQL Protocol 26
2.2.5 C-SPARQL - A Continuous SPARQL Query Language 31
2.2.6 SPARQL Query Processing 35
2.2.7 C-SPARQL Query Processing 37
2.3 Summary 38

3 Related Work 39
3.1 Frequent Pattern Mining Approaches 39
3.2 Historical Data Representations 41
3.2.1 Data Compression Approaches 41
3.2.2 Data Compression based Query Optimization 44

3.2.3 Big Data Tools and RDF 45
3.3 Streaming Data Management 47
3.3.1 Building Knowledge Graph On-Demand 47
3.3.2 Query Processing for Streaming Data 48
3.4 Summary 49

4 Compact Representations 51
4.1 Motivating Example 53
4.2 RDF Graph Factorization Approach 55
4.2.1 Problem Statement 55
4.2.2 FSP Detection Approach 60
4.2.3 A Factorization Approach 64
4.3 Experimental Study 68
4.3.1 Efficiency of FSP Detection Approach 70
4.3.2 Effectiveness of FSP Detection Approach 71
4.3.3 Effectiveness of RDF Graph Factorization 74
4.4 Summary 75

5 Integration of Historical Semantic Sensor Data 77
5.1 Motivating Example 79
5.2 The Semantic Sensor Data Factorization Approach 81
5.2.1 Problem Statement 81
5.2.2 A Factorization Approach 92
5.2.3 Queries over Factorized RDF Graphs 95
5.3 Tabular Representation of RDF Graphs 98
5.4 Experimental Study 107
5.4.1 Efficiency and Effectiveness of Factorized RDF 109
5.4.2 Impact of Factorized RDF on Query Processing 110
5.5 Summary 116

6 Integration of Streaming Observational Data 119
6.1 Motivating Example 121
6.2 Problem Statement and Proposed Solution 123
6.2.1 Problem Statement 126
6.2.2 Proposed Solution 127
6.2.3 Knowledge Graph Description Model 128
6.3 The DESERT Architecture 130
6.4 Experimental Evaluation 133
6.4.1 Performance with Uniform Data Stream Speed and Window Size 134

6.4.2 Performance with Uniform Data Stream and Varying Window Size . . . 136
6.4.3 Performance with Varying Data Stream and Uniform Window Size . . . 138
6.4.4 Performance with Varying Data Stream and Window Size . 140
6.5 Summary . . . 142

7 Conclusion and Future Directions 143
7.1 Revising the Research Questions . . . 144
7.2 Limitations . . . 146
7.3 Future Directions . . . 147
7.4 Closing Remarks . . . 147

Bibliography 149

List of Figures

1.1 Representation and Processing of Observational Data 3
1.2 Thesis Challenges 7
1.3 Thesis Contributions 11

2.1 Semantic Sensor Network Ontology 24
2.2 Examples of RDF Graphs 26
2.3 An Execution Plan for Star-Shaped Groups 36
2.4 C-SPARQL Engine Architecture 37

3.1 Categories of the state-of-the-art Approaches 40

4.1 Challenges and Contributions to the Problem of Knowledge Graph Representations. 52
4.2 Motivating Example Illustrating Frequent Star Patterns 54
4.3 Graph Patterns Identified by gSpan 55
4.4 The Frequent Star Patterns Detection Problem 57
4.5 The RDF Graph Factorization Problem 58
4.6 Frequent Star Patterns Detection using *E.FSP* and *G.FSP* 63
4.7 Transformations in RDF Graph 66
4.8 RDF Graph Factorization Overhead 67
4.9 Percentage of Repeated RDF Triples with Observation Values 69
4.10 Nodes and Labeled Edges 72

5.1 Challenges and Contributions to the Problem of Efficient Representations of Historical Semantic Sensor Data 78
5.2 Motivating Example for Historical Semantic Sensor Data Representations 80
5.3 Example of a Simplified RDF 82
5.4 RDF Molecule Templates (RDF-MTs) and RDF-MT Linking 83
5.5 Compact Observation and Measurement Molecules 86
5.6 Instance of the Semantic Sensor Data Factorization Problem 88
5.7 Instance of the Query Evaluation Problem 91
5.8 Factorization of the Running Example 96
5.9 Example of Query Rewriting 97
5.10 Factorized Tabular Representation of RDF Graphs 99

5.11 Query Evaluation Over Universal and Factorized Tables 100
5.12 RDF-MT based Tabular Representation of RDF Graphs 102
5.13 Query Evaluation Over RDF-MT based Tables 104
5.14 Query Execution Time ET (ms Log-scale) over RDF3X on Cold Cache . . 111
5.15 Query Execution Time ET (ms Log-scale) over RDF3X on Warm Cache . 112
5.16 Query Execution Time ET (ms Log-scale) over Relations on Cold Cache . 113
5.17 Query Execution Time ET (ms Log-scale) over Relations on Warm Cache 114
5.18 Performance of Factorized RDF Graphs over SPARQL Endpoints 115

6.1 Challenges and Contributions to the Problem of Semantic Description of Streaming Observational Data 120
6.2 Motivating Example for Streaming Observational Data 122
6.3 Example of IoT Data Stream and Knowledge Graph 123
6.4 Instance of the On-demand Knowledge Graph Creation Problem 128
6.5 Example of Knowledge Graph Description 129
6.6 The DESERT Architecture 130
6.7 Query Decomposition 132
6.8 Performance with Uniform Data Stream Speed and Window Size 135
6.9 Performance with Uniform Data Stream Speed and Varying Window Size 137
6.10 Performance with Varying Data Stream Speed and Uniform Window Size 139
6.11 Performance with Varying Data Stream Speed and Window Size 141

List of Tables

4.1 Statistics of the Datasets with Observations 68
4.2 Observation and Measurement Classes 69
4.3 Efficiency of Frequent Star Patterns Detection 71
4.4 Values Computed for Formula 1 73
4.5 Percentage Savings in Labeled Edges after Factorization 74

5.1 Query Rewriting Rules 89
5.2 Description of the Semantically Described Sensor Datasets 105
5.3 Efficiency and Effectiveness of the Semantic Sensor Data Factorization . . 108

6.1 Data Stream Description for Uniform Data Stream Speed and Window Size Dimensions 134
6.2 Data Stream Description for Uniform Data Stream Speed and Varying Window Size Dimensions 136
6.3 Data stream Description for Varying Data Stream Speed and Uniform Window Size Dimensions 138
6.4 Data Stream Description for Varying Data Stream Speed and Window Size Dimensions 140

Chapter 1

Introduction

The Internet of Things (IoT) is of paramount importance in our increasingly data-driven society and has received growing attention by the research community. The IoT paradigm envisions physical and virtual devices (e.g., sensors, mobile phones, vehicles, social media, and news feeds) to interconnect and share data with each other. These IoT devices continuously generate observations in *data streams*, when these observations are stored the data streams become *historical data*. Each IoT device generates observational data by recording different real world phenomena, and shares the observational data to communicate with other IoT devices. Predictions suggest that the number of currently deployed five billion IoT devices will increase to be 50 billion IoT devices in the year 2020 [24]. With the proliferation of IoT, the amount of observational data produced by the IoT devices will grow significantly in the near future. Additionally, IoT accelerates the transition from a static Internet into a fully dynamic one, where a large number of interconnected IoT devices and applications are continuously generating observational data streams at various velocities. Moreover, recent research indicates that the semantic description of IoT observational data using Semantic technologies [18, 58, 91] allows IoT devices and applications to inter-operate and interact comprehensively [59, 77, 79]. In order to be able to manage the rapidly growing observational data generated by IoT devices, efficient ways for data integration and semantic description to enhance and facilitate the development of IoT applications have to be developed.

In real-world IoT applications, a large number of IoT sensors and devices generate observations with the same observed values over time. A modern wind park, for example, comprises hundreds of wind turbines, each of which is equipped with hundreds of sensors generating values in frequencies of 1-100 Hz. In such a scenario, the number of observations generated by these sensors is much higher than the uniqueness of observed values. Such observations and their meanings are represented using standards for knowledge description, e.g., in a knowledge graph using the Resource Description Framework (RDF). These representations comprise patterns containing facts that are shared by a large number of observations. In case these patterns commonly appear, they negatively impact not only on the required amount of memory but also on the processing of observational data. Thus, techniques for efficiently representing observations containing frequent patterns are still required, as well as the processing over the generated representations.

Knowledge graphs have gained momentum as flexible and expressive structures for representing not only data and knowledge, but also actionable insights [103]; they provide the basis for effective and intelligent applications. Currently, knowledge graphs are utilized in diverse domains, e.g., DBpedia [62], Google Knowledge Graph [97], and KnowLife [35]. The Resource Description Framework (RDF) [61] has been adopted as a formalism to represent knowledge graphs. In fact, in the Linked Open Data cloud [19], there are 1,239 RDF knowledge graphs available[1] in march 2019. RDF models knowledge in the form of graphs where nodes represent entities and connections between nodes correspond to RDF triples composed of subject, property, and object. The subjects and objects are denoted by nodes, and an edge denotes a property that relates a subject with an object. Diverse applications have been developed on top of knowledge graphs [11, 43, 103]. However, the adoption of knowledge graphs as *de facto data* structure demands efficient and scalable techniques for creating, managing, and answering queries over knowledge graphs. Thus, efficient graph representations are still demanded to enhance and facilitate the development of applications over knowledge graphs.

In real-world applications, a group of entities can share the same values in a set of features. For example, several sensor observations can sense the same temperature, in a given timestamp and city. This situation can be depicted in an RDF graph with four triples per sensor observation obs_i, i.e., (obs_i *temperature temp*), (obs_i *unit uom*), (obs_i *timestamp ts*), and (obs_i *gps_coordinates gc*). All the resources corresponding to these sensor observations match the variable ?*obs* in the star pattern (SGP) composed by the conjunction of the following triple patterns (?*obs temperature temp*) (?*obs unit uom*), (?*obs timestamp ts*), and (?*obs gps_coordinates gc*) [85]. In case the star patterns are instantiated with many entities, a large number of RDF triples will have the same properties and objects and the corresponding star pattern will be repeatedly instantiated; we name these star patterns *frequent star patterns.* Although RDF triples that instantiate a frequent star pattern correctly model the real world, the size of the knowledge graph as well as the performance of the tasks of management and processing, can be negatively affected whenever a large number of triples of frequent star patterns populate the knowledge graph. Since frequent star patterns appear commonly in knowledge graphs, techniques are required to enable both the efficient representation of the

---

[1] `https://lod-cloud.net/`

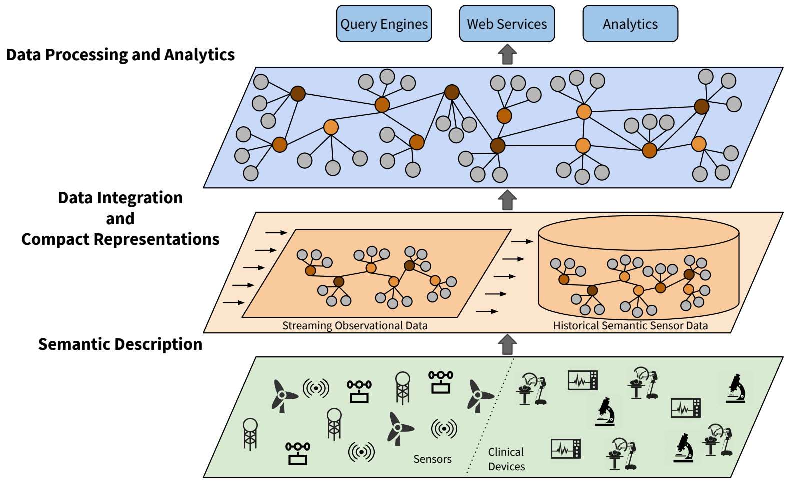

Query Engines
Web Services
Analytics
Data Processing and Analytics
Data Integration and Compact Representations
Streaming Observational Data
Historical Semantic Sensor Data
Semantic Description
Sensors
Clinical Devices

Figure 1.1: **Representation and Processing of Observational Data**. Observational data produced by diverse IoT devices is consumed either directly from data streams and processed in real time, or stored as historical data. Observational data and the meaning encoded in the data are represented and integrated in order to be accessed by higher level applications to perform data analytics and other processing tasks. Higher level services access and process the integrated view of the streaming and historical observational data to get actionable insights.

knowledge encoded in these star patterns, as well as the processing and traversal of the represented knowledge.

This thesis provides criteria to recognize frequent star patterns in knowledge graphs describing historical data. Furthermore, we exploit these frequent star patterns to generate efficient representations of the knowledge graphs to facilitate query processing and data analytics tasks over this data. The proposed representations empower the storage and processing of historical data while the knowledge encoded in the data is preserved. In addition, this thesis proposes techniques to efficiently describe the semantics of streaming data that is continuously and massively produced by IoT devices with varying data stream speeds. We devise a knowledge-driven approach that exploits the semantics encoded in the data that is extracted on-demand from data streams using a continuous query against the data streams while the completeness of query answers is respected.

1.1 Motivation

Providing an integrated view and semantic description of data generated by diverse IoT devices creates new capabilities and actionable insights. Several steps of processing are required to be performed in order to provide an integrated view for exploring and querying this data. Figure 1.1 shows different processing steps in establishing an integrated view of data produced by various IoT sensors and devices. The first step, *Semantic Description*, allows for the semantic description of data coming from IoT devices. This data can be consumed and processed either directly as it arrives in data streams, i.e., streaming data, or can be stored and processed later, i.e., historical data. The second step, *Data Integration and Compact Representations*, involves representing the data and the encoded knowledge. IoT devices produce data continuously with varying data stream speeds. These streaming sources are active on longer time scales on which a wide range of data streams are continuously arriving from disparate sources. Hence, maintaining a fresh and uniform integrated view for unlimited discovery and analysis of tremendously growing data is quite strenuous task. The third step, *Data Processing and Analytics*, allows higher level services to access and process the integrated data. Exploration of trends and patterns and discovering relationships in such a massive amount of dynamic data are highly challenging. Eventually, passing through all the described steps, the data originated from diverse IoT devices become available as integrated knowledge for further processing. However, providing an integrated view and processing of continuously and massively produced data are quite challenging tasks. Therefore, largely generated streaming data demand efficient ways for data integration and processing.

1.2 Existing Approaches

Database and Semantic Web communities have addressed the problem of representing relational and graph data models. These research communities have proposed a variety of representation methods and data structures that take into account the main features of a relational or graph model with the aim of speeding up relation and graph based analytics [2, 6, 8, 20, 37, 42, 51, 55, 63, 69, 71, 75, 83, 108]. Compression techniques [2, 108] over the column-oriented databases [21, 48, 67, 98], use the decomposition storage model [29] to maintain data, where each attribute value and a surrogate key, from the conceptual schema, are stored in a binary relation. A relation stored using the decomposition storage model cannot easily exploit compression unless surrogate keys are repeated [29]. Further, the decomposition model stores two copies of a binary relation, also the surrogate keys are required to be stored repeatedly for each attribute causing an increase in the storage space requirements. Lehmann et al. [63] exploit k^2-tree [23] structures to generate compact representations of graph data in order to efficiently perform *two-way regular-path* queries. k^2-tree structures resort to the sparseness and clustering features of adjacency matrix associated with a graph to generate compact binary representations, which require a customized engine to process queries over graphs. Neo4j[2] graph database efficiently stores graphs [42]. A node traversal using Neo4j is costly as all the edges incident on a node in the graph are inspected and also the cost of visiting the neighbours of a node in the graph depends on the node degree [66]. Sparksee [69] splits huge graphs into small data structures to favor the caching of the significant part of data in main memory to achieve efficient storage and processing [70]. Nevertheless, Neo4j and Sparksee offer numerous advantages such as reliable storage, access control, and transactions outperforming the traditional relational databases; however, they do not exhibit good performance particularly when the size of intermediate results is huge [44].

In the context of RDF graph, the scientific community has also actively contributed; approaches like [8, 20, 37, 75] generate compact binary representations for RDF knowledge graphs. RDF binary compression techniques do not take into account the semantics encoded in knowledge graphs; they require customized engines to perform query processing. Moreover, there have been defined compression approaches [51, 55, 71, 83] for RDF graphs able to exploit semantics encoded in RDF triples. The approaches [71, 83] are application dependent and require compression rules and constraints as input from a user. Alternatively, compression approaches tailored for ontology properties [55] have shown to be effective, but they require prior knowledge of classes and properties involved in repeated graph patterns to generate compact representations. Lastly, techniques proposed by Joshi et al. [51] require decompression to access and process data involving extra processing over data. Albeit effective in reducing the storage space, existing compression methods add overhead to the process of data management, and particularly, query execution time can be negatively impacted. gSpan [105] and GRAMI [33] are state-of-the-art algorithms that aim to identify frequent patterns; however, only patterns with constants are considered and they are neither able to identify star patterns nor decide *frequentness*. In this thesis, we implement an exhaustive algorithm that resorts to gSpan enumeration of frequent patterns to identify the frequent star patterns in an RDF knowledge graph; this approach corresponds to the baseline of our empirical evaluation.

To scale-up to large RDF datasets, existing approaches [32, 57, 68, 74, 76, 87, 92, 93, 94] exploit distributed and parallel processing frameworks. RDF storage layouts and partitioning techniques proposed in [32, 57, 68, 93, 94] utilize distributed and parallel processing frameworks for the efficient processing of RDF data. Similarly, RDF data partitioning and indexes presented in [74, 76, 87, 92] efficiently process queries over RDF data using Big Data frameworks. These approaches use HDFS and Hadoop MapReduce frameworks to store and process

---

[2] `https://neo4j.com/`

RDF data. In the context of query processing, efficient SQL query processing techniques based on the factorization of the data are proposed in [13, 14]. Despite these storage and processing techniques, the tremendously growing observational data require efficient representations to facilitate the storage and processing of such data. In this thesis, we propose factorization techniques for semantic sensor data where RDF triples related to the redundant observation values are represented only once. To scale up to large datasets, tabular representations based on the factorized semantic sensor data are presented to facilitate storage and processing of large semantic sensor data using Big Data frameworks.

The Semantic Web community has extensively studied methods for the semantic description of IoT data [26, 82], and continuous query processing over the semantically described streaming observational data [15, 81]. Albeit effective for semantic description and processing of streaming observational data, such approaches are less feasible and do not scale for large streaming observational data generated by a huge number of IoT devices. Moreover, redundantly observed values increase the size of the semantically described streaming observational data generated using these approaches. In addition, the presence of redundantly observed values in the streaming observational data demand techniques to reduce these redundant values from a knowledge graph describing observational data, while preserving the completeness of query answers. This thesis overcomes the limitations of the existing linked data stream processing techniques, and tackles the problem of *on-demand knowledge graph creation.* We provide techniques for on-demand building of knowledge graph describing observational data required to answer an input continuous SPARQL query.

1.3 Problem Statement and Challenges

For making IoT a reality, observations produced by sensors, smart phones, watches, and other IoT devices need to be integrated; moreover, the meaning of observations should be explicitly represented. This thesis aims at integrating and semantically describing observational data produced by diverse IoT devices. The following research problems are addressed; **1)** *identifying redundant observations* in RDF knowledge graphs; **2)** *efficient query processing over historical semantic sensor data*; and **3)** *on-demand knowledge graph creation.* Since IoT devices continuously produce data, this data can be consumed in real time or collected and stored, i.e., *streaming* or *historical* data. With the proliferation of IoT devices, huge volumes of streaming and historical data are being generated. Furthermore, processing streaming data in real time poses different challenges than processing historical data. To address the above mentioned problems, this thesis identifies several challenges in terms of streaming and historical data, as shown in Figure 1.2.

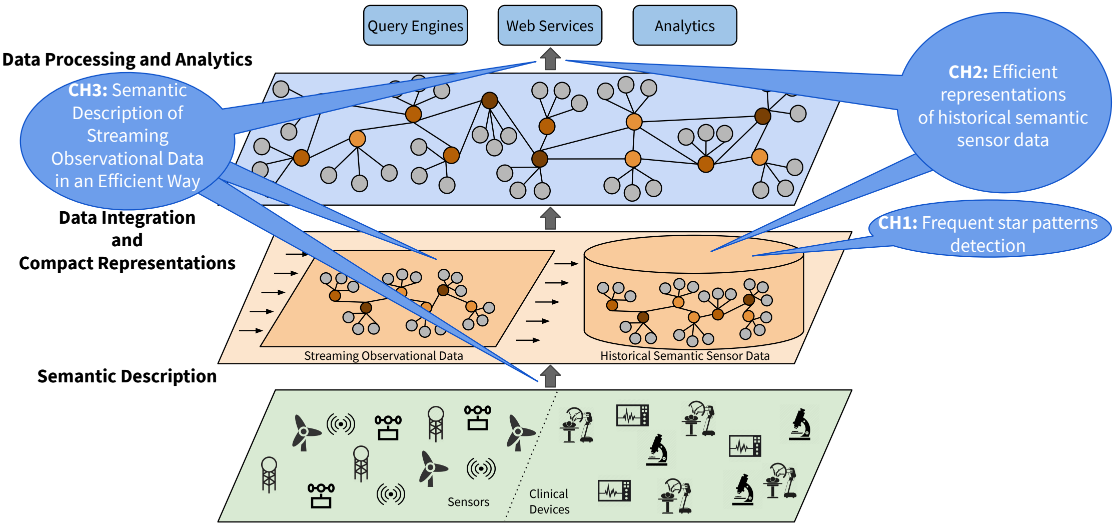

Query Engines
Web Services
Analytics
Data Processing and Analytics
CH3: Semantic Description of Streaming Observational Data in an Efficient Way
CH2: Efficient representations of historical semantic sensor data
Data Integration and Compact Representations
CH1: Frequent star patterns detection
Streaming Observational Data
Historical Semantic Sensor Data
Semantic Description
Sensors
Clinical Devices

Figure 1.2: **Thesis Challenges.** Three challenges are identified in this thesis, to generate efficient representations of observational data. The first challenge **CH1**, is to detect frequent star patterns in knowledge graphs representing observational data. The second challenge **CH2**, is to generate efficient representations of historical semantic sensor data without losing the encoded information. The third challenge **CH3** is the semantic description of streaming data in an efficient way.

Challenge 1: Frequent Star Patterns Detection

Data produced by diverse IoT devices comprise observations with related properties, e.g., feature of interest, timestamp, observed value, and unit of measurement. A large number of these observations can share the same values for a certain set of properties. Representing such observations and their meaning in RDF knowledge graphs generate sets of RDF triples encompassing similar properties and relevant object values. These sets of RDF triples correspond to a star pattern that is a conjunction of a set of RDF triples containing the properties and corresponding objects that are shared among the sets of RDF triples. The subject of the star pattern is a variable to which all the entities representing the observations are mapped. In case the number of entities mapping to the star pattern is high, the star pattern is referred to as a frequent star pattern. Existence of frequent star patterns in a knowledge graph negatively impact on the size and management tasks of the knowledge graph. At the *Data Integration and Compact Representations* step in Figure 1.2 involving historical data, the first challenge is to find out the properties and corresponding objects that are repeatedly shared by several entities causing unnecessary growth of knowledge graphs describing observations and their meaning. Therefore, identifying the existence of such frequent star patterns in knowledge graphs is crucial in order to represent them in an efficient way without losing encoded information.

Challenge 2: Efficient Representations of Historical Semantic Sensor Data

RDF representations of IoT data are being generated [40, 49, 82] in order to add semantics to the data and to turn the data into meaningful actions for providing the IoT applications with new capabilities, facilitate knowledge sharing and exchange, and richer experiences. The Semantic Sensor Network (SSN) Ontology [28] is a W3C standard to describe the data generated by sensors, refer as semantic sensor data. The SSN Ontology consists of several classes and corresponding properties to describe the meaning of sensor data in terms of sensor capabilities, observations, and measured values in an RDF graph. However, RDF representations generate an enormous amount of data, as a result, efficient representations of sensor data are required. Furthermore, several sensor observations with the same measurement values generate RDF data redundancy. These data redundancies negatively impact on the size of the semantic sensor data, and hence the RDF storage and processing over diverse implementations for RDF data. The second challenge is to have efficient representations of semantic sensor data in order to store and process large amounts of sensor data using different RDF implementations. This challenge spans over the *Data Integration and Compact Representations* and *Data Processing and Analytics* steps involving historical observational data as shown in Figure 1.2.

Challenge 3: Semantic Description of Streaming Observational Data in an Efficient Way

IoT sensors and devices continuously generate observations in data streams, and integration and semantic description of these observations require dealing with the volume and velocity aspects of the produced data. Describing semantics of streaming data brings more benefits to IoT applications. Nevertheless, semantic description tremendously increases the size of integrated streaming data. Moreover, the presence of redundant observations further increases data size without adding anything new in terms of data comprehension. Furthermore, query execution over continuously generated data consumes a lot of resources, e.g., time and memory. In addition, not all the observations generated in data streams are required to answer a query. Moreover, the existence of repeated measurement values, as well as varying data stream speeds makes the task even more challenging. The third challenge is to develop techniques for on-demand building of knowledge graphs and query processing against data streams. In addition, the presence of duplicated measurement values in streaming data demands techniques to reduce these duplicated measurements from a knowledge graph on the fly while all the query answers are produced. The third challenge, spans over the three steps; *Semantic Description*, *Data Integration and Compact Representations*, and *Data Processing and Analytics*, involving streaming observational data as shown in Figure 1.2.

1.4 Research Questions

RQ1: What are the criteria to identify frequent star patterns?

To answer this question, we investigate the state-of-the-art frequent patterns detection techniques. We propose the concept of star patterns; a star pattern comprises a set of properties and corresponding objects to which one or more subject entities can map, and compute their frequencies. The frequency of a star pattern is the number of subject entities that map to the star pattern. A star pattern with a high frequency is referred to as a frequent star pattern. Moreover, we exploit these frequent star patterns to generate efficient representations of the knowledge encoded in these patterns. We evaluate the efficiency of detecting frequent star patterns in comparison to the state-of-the-art approaches. Furthermore, effectiveness of the proposed frequent star patterns detection approach is assessed by considering different sets of features involved in star patterns. In addition, we evaluate the effectiveness of the generated representations. The experimental results show that the proposed techniques for frequent star patterns detection are able to effectively and efficiently recognize frequent star patterns. Moreover, the proposed representations effectively reduce the size of observational data while the encoded knowledge is preserved.

RQ2: How can efficient representations be exploited to manage historical semantic sensor data?

To answer this research question, we study the storage and processing of historical data using different data models. We propose techniques for efficient storage and processing of historical semantic sensor data. We devise factorized representations of semantic sensor data where repeated RDF triples corresponding to the same observed value are factorized and stored only once. We exploit these factorized RDF representations of sensor data to generate tabular-based representations to scale up to large datasets using Big Data tools. We represent semantic sensor data using universal tabular representations such that all the attributes of sensor data are stored in a single universal table. Furthermore, we present RDF molecule template (RDF-MT) based tabular representations of semantic sensor data. RDF molecule templates are abstract descriptions of data sources in terms of classes, class attributes, and relationships among the classes [34]. We evaluate the performance of the proposed representations and their impact on query processing using the state-of-the-art RDF and Big Data engines. The results show that the proposed representations enhance the performance of RDF and Big Data engines.

RQ3: How can on-demand knowledge graph building reduce the size of the streaming observational data?

We analyse the state-of-the-art techniques for semantically describing streaming data. We propose techniques for on-demand knowledge graph creation from streaming observational data. A continuous SPARQL query is used to retrieve data from data streams. The data extracted from the data streams encompass the observations required to answer the input query. These observations are factorized and semantified such that the observations corresponding to the same observed value are described by the same set of RDF triples in the knowledge graph. We evaluate the impact of proposed on-demand factorization and semantification techniques on the size of streaming observational data in two dimensions, i.e., by varying data stream windows size and data stream speeds. The experimental results suggest that the proposed techniques effectively reduce the size of generated knowledge graphs while complete answers for input queries are returned.

RQ4: How can on-demand knowledge graph building speed up query processing?

To answer this research question, we study the continuous SPARQL query processing techniques over streaming RDF data. We propose a continuous SPARQL query engine for streaming observational data. The proposed query engine receives a continuous SPARQL query and executes the query against a knowledge graph and a data stream, where the knowledge graph contains factorized RDF representations of the streaming data. The query engine rewrites the input query into a query against the knowledge graph describing the factorized data. Moreover, we use description of the knowledge graph to keep track of the observations that are already integrated in the knowledge graph. The knowledge graph description is used to decompose the rewritten query into subqueries against the knowledge graph and the data streams. The subqueries against the data streams are forwarded to the customized wrappers which transform the subqueries into calls to the corresponding IoT streaming data sources. The results from the subqueries are used to produce the final query results. We evaluate the impact of the proposed continuous SPARQL query engine on the streaming observational data by using different combinations of the data stream windows size and data stream speeds. The observed results suggest that the proposed techniques are able to effectively

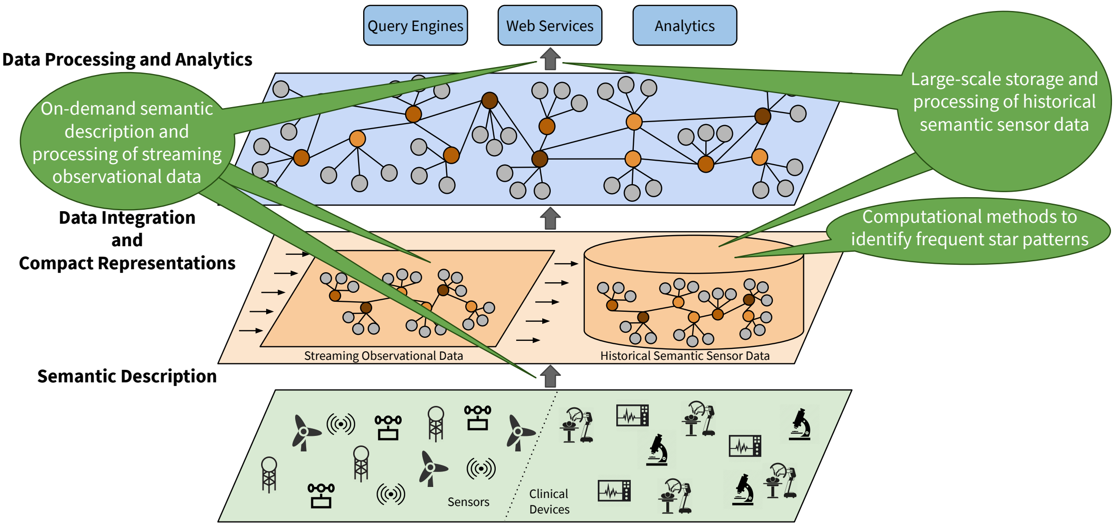

Query Engines
Web Services
Analytics
Data Processing and Analytics
On-demand semantic description and processing of streaming observational data
Large-scale storage and processing of historical semantic sensor data
Data Integration and Compact Representations
Computational methods to identify frequent star patterns
Streaming Observational Data
Historical Semantic Sensor Data
Semantic Description
Sensors
Clinical Devices

Figure 1.3: **Thesis Contributions.** Three main contributions of this thesis towards the efficient representations and processing of streaming and historical data produced by diverse IoT devices: computational methods to detect frequent star patterns in RDF knowledge graphs representing historical observational data, large-scale storage and processing of historical semantic sensor data, and on-demand semantic description and processing of streaming observational data.

and efficiently execute queries.

1.5 Thesis Overview

Intending to prepare the reader for the rest of the document, we present an overview of the main contributions of this thesis and references to the scientific publications covering this work. The thesis contributions are shown in Figure 1.3.

1.5.1 Contributions

- **Contribution 1:** In this thesis, we devise the concept of *factorized RDF graphs*, which corresponds to a compact graph with a minimized number of frequent star patterns. Furthermore, we develop computational methods to detect frequent star patterns in RDF graphs and to generate a *factorized RDF graph*. These methods are able to identify entities and properties in frequent star patterns in RDF graphs, and generate factorized RDF graphs by representing frequent star patterns with compact RDF molecules. A compact RDF molecule of a frequent star pattern is an RDF subgraph that instantiates the star pattern; a surrogate entity stands for the entities that satisfy the corresponding frequent star pattern. The surrogate entity is linked to the properties and the corresponding objects in the frequent star pattern. The entities, initially matching the frequent star pattern, are also linked to the surrogate entity of the compact RDF molecule. Compact RDF molecules significantly reduce the size of the RDF graph by replacing labeled edges and entities connected to the objects in the frequent star pattern, with edges linking the entities to the surrogate entity of a compact RDF molecule. We study the effectiveness of our factorization techniques over the *LinkedSensorData* benchmark [77]; it describes more than 34,000,000 weather observations collected by around 20,000 weather stations in the United States since 2002. Experiments are conducted against three *LinkedSensorData* RDF graphs by gradually increasing the graph size. The observed results evidence that frequent star patterns characterize the best set of properties relating several entities of a class to the same objects in an RDF graph. Moreover, our techniques reduce RDF graphs size by up to 66.56% using properties and classes recommended by the frequent star patterns detection approach. These results allow us to answer the research question **RQ1**.

- **Contribution 2.** This thesis proposes the *Compacting Semantic Sensor Data (CSSD)* approach for efficient storage of historical semantic sensor data that enhance the performance of query engines over the diverse implementations for RDF data. The *CSSD* approach is based on factorizing the data and storing only a *compact* or *factorized* representation of semantic sensor data, where repeated values are represented only once. Besides, tabular-based representations leveraging the columnar-oriented *Parquet* storage format for HDFS are utilized to scale up to even larger RDF datasets of factorized semantic sensor data. We represent RDF graphs using universal tables [100] where all the attributes describing sensor observations are stored in a single giant "universal table". In addition, RDF molecule template (RDF-MT) based tabular representations of RDF graphs are presented. An RDF molecule template is an abstract description of entities belonging to an RDF class. RDF molecule templates are proposed by Endris et al. [34] to describe data sources in terms of abstract representations of classes, attributes of classes, and their connections to other classes in the same datasets (Intra-Linking) and to classes in other datasets (Inter-Linking). An RDF-MT based tabular representation includes all attributes of a class in one table, whereas a separate table for each intra- and inter-link of the class is created. The effectiveness of the proposed factorization techniques are empirically studied, as well as the impact of factorizing semantic sensor data on query processing over several RDF storage implementations. The effects of storing factorized RDF data over diverse RDF implementations using the state-of-the-art RDF and Big Data engines are evaluated. In this thesis, we study the effectiveness of the proposed compact representations over the *LinkedSensorData* benchmark [77]. The *LinkedSensorData* contains almost 2 billion RDF triples to describe more than 34 million weather observations collected from around 20,000 weather stations during blizzard and storm seasons in the United States since 2002. The experiments are conducted over gradually increasing three *LinkedSensorData* RDF graphs. The observed results demonstrate that the proposed factorization techniques are able to effectively reduce the size of semantic sensor data while the encoded information is preserved. Furthermore, the results for query processing indicate that the proposed factorization techniques are able to enhance the performance of RDF and Big Data query engines and query execution time can be reduced by up to two orders of magnitude, answering the research question **RQ2**.

- **Contribution 3.** In this thesis, we propose DESERT, a continuous SPARQL query engine able to on-Demand factorizE and Semantically Enrich stReam daTa. DESERT is implemented on top of the C-SPARQL engine [15], a query engine for the continuous SPARQL queries processing over RDF data [30]. DESERT receives continuous SPARQL queries and builds a knowledge graph on-demand. The input queries are decomposed into subqueries, and forwarded to the customized wrappers, which transform these subqueries into calls to the IoT stream data sources and retrieve the query results. DESERT semantically describes and integrates these results into the knowledge graph. In addition, DESERT uses semantics encoded in IoT stream data for semantification of data. DESERT extends the factorization techniques for RDF data proposed by Karim et al. [55] for building IoT knowledge graphs on-demand. The quality of DESERT on-demand factorization and semantification techniques has been empirically evaluated in ten continuous SPARQL queries from SRBench[107] against the IoT stream data generated from weather observations in the United States during the year 2003. The goal of the experiments is to analyze the performance of DESERT when continuous SPARQL queries are executed against IoT stream data in two dimensions, i.e., window size and data stream speed. All queries are executed using the RDF stream processing CSPARQL engine and the on-demand knowledge graph techniques proposed in DESERT considering different combinations of the uniform and varying streaming window size and data stream speed. Empirical results clearly show up to 92.53% and 94.34% savings in the knowledge graph size and number of produced triples, respectively, for on-demand factorization and semantification techniques while the complete query answers are generated. Moreover, DESERT exhibits more than 90% improvements in throughput keeping the memory usage less than 25%. These results confirm that on-demand building of knowledge graphs can effectively augment the savings in terms of knowledge graph size and memory while continuous SPARQL queries are effectively and efficiently evaluated over data streams, answering the research questions **RQ3** and **RQ4**.

1.5.2 List of Publications

This thesis is based on the following publications.

Peer-Reviewed International Journals

- **Farah Karim**, Ioanna Lytra, Christian Mader, Sören Auer, Maria-Esther Vidal. *DESERT: a continuous SPARQL query engine for on-demand query answering.* In: International Journal of Semantic Computing 12.03 (2018), pp. 373–397.

Papers in Proceedings of Peer-Reviewed Conferences

- **Farah Karim**, Ola Al Naameh, Ioanna Lytra, Christian Mader, Maria-Esther Vidal, Sören Auer. *Semantic enrichment of IoT stream data on-demand.* In: 2018 IEEE 12th International Conference on Semantic Computing (ICSC), pp. 33–40. IEEE (2018)
- **Farah Karim**, Mohamed Nadjib Mami, Maria-Esther Vidal, Sören Auer. *Large-scale storage and query processing for semantic sensor data.* In: Proceedings of the 7th International Conference on Web Intelligence, Mining and Semantics, p. 8. ACM (2017)
- **Farah Karim**, Maria-Esther Vidal, Sören Auer. *Efficient processing of semantically represented sensor data.* In: WEBIST, pp. 252–259 (2017)
- **Farah Karim**, Maria-Esther Vidal, Sören Auer. *Factorization techniques for longitudinal linked data.* In: OTM Confederated International Conferences" On the Move to Meaningful Internet Systems", pp. 690–698. Springer (2016)

Peer-Reviewed Book Chapters

- Maria-Esther Vidal, Kemele M. Endris, Samaneh Jozashoori, **Farah Karim**, Guillermo Palma. *Semantic data integration of big biomedical data for supporting personalised medicine.* In: Current Trends in Semantic Web Technologies: Theory and Practice, pp. 25–56. Springer (2019)

1.6 Thesis Structure

The rest of the thesis is structured as follows: Chapter 2 introduces the basic concepts in the fields of data integration system, Semantic Web, knowledge graphs that are necessary to understand the work presented in the thesis. Chapter 3 discusses the state-of-the-art research work related to this thesis. The related approaches are categorized under three topics. First, we discuss state-of-the-art approaches for graph patterns mining. Then, we present the existing approaches for efficient representations and processing of historical data. Furthermore, the approaches exploiting Big Data tools to efficiently process large amount of RDF data are explored. Finally, we present the existing techniques for integration and processing of streaming data. Chapter 4 presents two algorithms for frequent star patterns detection in RDF knowledge graphs. The first algorithm, E.FSP, extracts frequent star patterns by exhaustively searching the space of frequent graph patterns generated by the state-of-the-art frequent pattern mining algorithms, like gSpan. The second algorithm, G.FSP, adopts a Greedy approach to traverse the space of star patterns for detecting frequent star patterns. Furthermore, factorization techniques utilizing frequent star patterns to generate compact representations of knowledge graphs are proposed. We present a detailed analysis of the proposed techniques for frequent star patterns detection and factorization techniques over the state-of-the-art benchmarks. The results show that G.FSP identifies frequent star patterns faster than E.FSP. Moreover, factorizing frequent star patterns reduces the size of RDF knowledge graphs while the knowledge encoded in the data is preserved. Chapter 5 presents the *Compacting Semantic Sensor Data (CSSD)* approach for efficient storage of semantic sensor data. In Chapter 5, we devise factorized RDF representations for semantic sensor data and these factorized RDF representations are utilized to generate tabular-based representations for the semantic sensor data. The empirical evaluation using existing benchmarks shows that the proposed representations are able to efficiently store and process semantic sensor data using RDF and Big Data frameworks. In Chapter 6, we present DESERT, a continuous SPARQL query engine able to on-Demand factorizE and Semantically Enrich stReam daTa. DESERT integrates streaming observational data into knowledge graphs by semantically describing the observations in streaming data that are required to answer an input continuous query against data streams. DESERT implements factorization techniques to reduce the redundant measurements in knowledge graphs. Experimental evaluation shows that DESERT is able to speed up query processing over streaming observational data while the generated knowledge graph contains no redundancies. Finally, Chapter 7 concludes the work presented in this thesis and discusses the limitations of the work. Moreover, Chapter 7 proposes some future work in related areas of research.

1.7 Summary

Connecting the physical world to the Internet of Things (IoT) allows for the development of a wide variety of applications. Things can be searched, managed, analyzed, and even included in collaborative games. Industries, health care, and cities are exploiting IoT data-driven frameworks to make their organizations more efficient, thus, improving the lives of citizens. A large amount of observational data is being produced by the diverse IoT devices. This data can be ingested in real time as streaming data or can be stored as historical data. For making IoT a reality, data produced by billions of IoT sensors and devices need to be integrated and the meaning of IoT data should be explicitly described. Knowledge graphs have been extensively used as expressive data structures for representing data, the knowledge encoded in the data, and actionable insights. Resource Description Framework (RDF) has been widely adopted to represent knowledge graphs. In real-world scenarios, IoT sensors and devices generate observational data that contain observations sharing the same values for a set of features. Representations of such observations in RDF graphs generate star patterns. A star pattern constitutes RDF triples that contain the same object values for a set of properties, while the observation entities mapping the subject of the star pattern are different. If the number of entities mapping these star patterns is soaring these patterns are referred as frequent star patterns. Frequent star patterns negatively impact on the size and processing of large volumes of observational data. Furthermore, large amount of streaming observational data is being produced by IoT devices, and semantically describing this tremendous amount of streaming observational data and their meaning increases the size of observational data manifolds. These challenges, imposed by the big data and streaming nature of IoT observational data, need to be addressed in order to provide scalable and efficient IoT data-driven infrastructures. In this thesis, we tackle the problems of detecting frequent star patterns, query processing over historical semantic sensor data, and on-demand knowledge graph creation from streaming observational data. This thesis presents techniques for frequent star pattern detection and devices factorized representations of historical observational data based on these frequent star patterns without losing any knowledge encoded in the data. The empirical evaluation results show that the proposed techniques are able to efficiently and effectively detect frequent star patterns. Furthermore, these factorized representations are exploited to generate tabular-based representations for semantic sensor data to process large datasets using Big Data frameworks. The experimental evaluation shows that the proposed factorized representations improve the query processing performance over RDF and Big Data engines. In addition, we devise DESERT, a continuous SPARQL query engine able to on-demand factorize and semantically describe streaming observational data in a knowledge graph. Resulting knowledge graphs model the semantics or meaning of merged data in terms of entities that satisfy the SPARQL queries and relationships among those entities; thus, only data required for query answering is included in the knowledge graph. We empirically evaluate the results of DESERT on SRBench, a benchmark of Streaming RDF data. The experimental results suggest that DESERT allows for speeding up query execution while the size of the knowledge graphs remains relatively low. The findings of this thesis indicate that the proposed compact representations are able improve IoT data management by efficiently representing streaming and historical IoT data.

Chapter 2

Background

In this chapter, we present the basic concepts and theoretical foundations for the research conducted in this thesis. In Section 2.1, the basic components of a data integration system are described. Furthermore, we discuss the challenges for IoT data integration. In section 2.2, we explain the vision and major concepts of Semantic Web. We give an overview of the Semantic Sensor Network (SSN) Ontology used to semantically describe observational data. Moreover, the Semantic Web technologies, i.e., Resource Description Framework (RDF), RDF Schema, and SPARQL query language are defined. Regarding SPARQL query processing, we look into a state-of-the-art SPARQL query engine RDF-3X. In addition, we enumerate the essential features of a continuous SPARQL query language C-SPARQL used to query RDF data streams. Moreover, this chapter describes the components of the C-SPARQL engine architecture that is used to execute continuous SPARQL queries against RDF data streams.

2.1 Data Integration System

Internet of Things (IoT) offers visibility and remote control over enterprise-wide processes. IoT enables end-to-end integration of various business processes and units, providing a better coordination between these entities in order to enhance business performance. Technology and business leaders are strongly emphasizing the fact that the real worth of IoT lies in the data generated by IoT sensors and devices. To leverage IoT data for the desired practical purposes, it is vital to collate IoT data produced by diverse IoT devices. For enterprises and their decision makers, IoT data integration is crucial to have the comprehensive integrated view of activities in their organisation and its environment. Data integration involves combining data from different sources to provide a unified view of these data sources to the user [45, 47, 99]. Data integration systems are characterized by an architecture based on a set of sources and a global schema. The sources contain real data, whereas the global schema provides an integrated view of these sources. A uniform access to the underlying data sources requires establishing the relationships between the sources and the global schema. Thus, a data integration system comprises a global schema, sources, and mappings between the global schema and the sources. A data integration system is formally defined as follows [64]:

Definition 2.1.1 (Data Integration System [64]). *A data integration system $\mathcal{I}$ is defined in terms of a triple $\langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$, where*

- *$\mathcal{G}$ is the global schema, expressed in a language $\mathcal{L}_{\mathcal{G}}$ over an alphabet $\mathcal{A}_{\mathcal{G}}$. The alphabet comprises a symbol for each element of $\mathcal{G}$ (i.e., relation if $\mathcal{G}$ is relational, class if $\mathcal{G}$ is object-oriented, etc.).*
- *$\mathcal{S}$ is the source schema, expressed in a language $\mathcal{L}_{\mathcal{S}}$ over an alphabet $\mathcal{A}_{\mathcal{S}}$. The alphabet $\mathcal{A}_{\mathcal{S}}$ includes a symbol for each element of the sources.*
- *$\mathcal{M}$ is the mapping between $\mathcal{G}$ and $\mathcal{S}$, constituted by a set of assertions of the forms*

  $$q_{\mathcal{S}} \rightsquigarrow q_{\mathcal{G}},$$

  $$q_{\mathcal{G}} \rightsquigarrow q_{\mathcal{S}}$$

  where $q_{\mathcal{S}}$ and $q_{\mathcal{G}}$ are two queries of the same arity over the source schema $\mathcal{S}$, and over the global schema $\mathcal{G}$. Queries $q_{\mathcal{S}}$ are expressed in a query language $\mathcal{L}_{\mathcal{M},\mathcal{S}}$ over the alphabet $\mathcal{A}_{\mathcal{S}}$, and queries $q_{\mathcal{G}}$ are expressed in a query language $\mathcal{L}_{\mathcal{M},\mathcal{G}}$ over the alphabet $\mathcal{A}_{\mathcal{G}}$. Intuitively, an assertion $q_{\mathcal{S}} \rightsquigarrow q_{\mathcal{G}}$ specifies that the concept represented by the query $q_{\mathcal{S}}$ over the sources corresponds to the concept in the global schema represented by the query $q_{\mathcal{G}}$ (similarly for an assertion of type $q_{\mathcal{G}} \rightsquigarrow q_{\mathcal{S}}$).

Specification of the correspondence between the global schema and the source data is one of the most important tasks in the design of a data integration system. The correspondence between the data at the sources and the global schema is established using mappings. The queries posed to the system are answered using this correspondence. For establishing the mappings two approaches have been defined in the literature, called *global-as-view* (GAV) [45], and *local-as-view* (LAV) [99].

2.1.1 Local-as-View (LAV)

Mappings established using the local-as-view (LAV) approach represents the data sources as views over the global schema. The mappings in $\mathcal{M}$ associate each element s in source schema $\mathcal{S}$ with a query $q_{\mathcal{G}}$ defined over the global schema. The mappings established by LAV comprise a set of assertions, one for each element s of source schema $\mathcal{S}$, given as:

$$s \rightsquigarrow q_{\mathcal{G}}$$

where $s \in \mathcal{S}$ and $q_{\mathcal{G}}$ is a query defined over the global schema $\mathcal{G}$ A query over the global schema. Since each data source is mapped to the global schema independent of the other data sources, therefore, extending the system with a new data source is straightforward and requires only the enrichment of mappings with new assertions.

2.1.2 Global-as-View (GAV)

Global-as-view (GAV) approach represents the concepts in the global schema as a set of views over the data sources. The mappings in $\mathcal{M}$ associate each element g in the global schema with a query $q_{\mathcal{S}}$ over the data source. The mappings established by GAV encompass a set of assertions, one for each element g of $\mathcal{G}$, given as:

$$g \rightsquigarrow q_{\mathcal{S}}$$

where $g \in \mathcal{G}$ and $q_{\mathcal{S}}$ is query defined over the sources in $\mathcal{S}$. A query against the global schema $\mathcal{G}$ needs to be rewritten with the views defined in the form of assertions in $\mathcal{M}$. The process of rewriting a query against a global schema into a query over data sources is referred to as query *unfolding.* GAV mappings compute tuples of global schema relations from tuples in the data sources, making the query unfolding easier. The mappings defined by GAV involve the knowledge about all the data sources to associate them with the global view. Therefore, adding or removing a source is quite complicated process, and requires updating all the mappings defined for various elements of the global schema.

2.1.3 IoT Data Integration

To potential benefits of IoT, driving insights from continuously produced IoT data poses various challenges in data integration. Since, massive IoT devices are deployed in IoT ecosystem, data can be generated from multiple sources with heterogeneous data formats. Lack of a common data model and semantic description of data leads to several interoperability conflicts, e.g., representation, schematic, entity matching, structuredness, and domain conflicts. These conflicts emerged because IoT data sources have different data models, diverse schemes for representation of data, and complementary information [31]. Therefore, representations and semantic descriptions of tremendously growing IoT data in efficient ways becomes fundamental to solve information retrieval and interoperability conflicts.

IoT sensors are distributed in dynamic environments at a large scale and produce ample amounts of IoT data with diverse characteristics, e.g., spatio-temporal.

Moreover, IoT sensors and devices repeatedly generate observations with same observed values. Such observations increase the memory usage and processing costs due to the repeated information about the observed data that add no new insights to the data. IoT data integration demands efficient ways to gain insights from such a massive amount of data with a lot of duplicated information, while minimizing the memory requirement and processing costs imposed by big nature of IoT data.

Data retrieval under dynamic IoT environments where streaming data sources are continuously generating observations, is extremely challenging. Integration of IoT data that is massively and continuously generated from disparate and uncontrolled data sources demands maintaining an up to date consistent version of data. Furthermore, scalable and efficient retrieval of valuable information and continuous queries must be generated over the large sized streaming IoT data.

In this thesis, we focus on the heterogeneous, voluminous and streaming nature of IoT data and address various data management challenges imposed by these characteristics of IoT data. We resort to an on-demand semantification and factorization approach to integrate streaming IoT data produced by heterogeneous sources. Furthermore, we address the challenges introduced by huge volumes of IoT data containing a lot of redundancies. We propose techniques to detect these redundancies and devise RDF and tabular based representations. Moreover, we exploit RDF and Big Data frameworks to store and process large amounts of data.

2.2 Semantic Web

The Semantic Web is an extension of the current web of documents in which documents on the Web are made machine-readable by annotating them and making their meanings explicit. According to the W3C, "The Semantic Web provides a common framework that allows data to be shared and reused across application, enterprise, and community boundaries"[1]. The Semantic Web exists as a vision to extend principles of the existing *Web of Documents* to *Web of Data.* The Semantic Web provides formalism for representing and accessing data. A set of standards and technologies are used to create data store, vocabularies, and rules are written for handling data. The core of the Semantic Web standards is the Resources Description Framework (RDF) for semantic markup and data interchange on the web, RDF schema language RDF Schema (RDFS), and the Web Ontology Language (OWL). These standards adopt the principles of knowledge representation languages in the context of Web, where several participants author knowledge in a decentralized fashion [31]. Linked Data is a set of best practices for interlinking and publishing machine-readable data on the Web [65]. Linked Data makes available the semi-structured data sources on the Web for both machines and human.

---

[1] `https://www.w3.org/2001/sw/`

Datasets must be released under an open license which does not prevent their free re-usage to ensure that Linked Data reaches its full potential. Linked Open Data (LOD) is Linked Data released under and open license [65]. The LOD initiation encouraged data providers to publish a large linked datasets from diverse domains leading to to the creation of semantically linked global data-space referred to as the Linked Open Data Cloud (LOD Cloud) [19]. DBpedia, Wikidata, YAGO, and Bio2RDF are the prominent datasets in LOD Cloud. In 2007, there were only 12 datasets in the LOD Cloud, which has grown to the 1,239 datasets and 16,147 links among these datasets in March 2019[2].

2.2.1 The Semantic Sensor Network (SSN) Ontology

The Semantic Sensor Network (SSN) Ontology [28], developed by the W3C Semantic Sensor Network Incubator Group[3], is an OWL ontology that allows for the description of sensor devices, their capabilities, observations, and other sensor-related concepts. The SSN ontology is commonly employed to address observational data semantic enrichment and interoperability problems when integrating heterogeneous data sources in IoT applications [5, 38, 46, 80]. The SSN ontology comprises 50 RDF classes and 55 properties to describe sensor data in terms of observations, features of interest, observed properties, and measurement values and units. Figure 2.1 illustrates a portion of the SSN ontology composed of classes and properties to describe sensors, observations, and measurement values. Sensors generate observations by detecting certain properties of features of interest and produce the observed values as sensor output. A feature of interest represents a physical object whose property is being estimated or calculated in the course of an observation by a sensor. Property is a relation that links an observation to the property of a feature of interest being observed. Sensor output contains information reported from a sensor about a property of a feature of interest during the act of carrying out an observation.

2.2.2 The Resource Description Framework (RDF)

The Resource Description Framework (RDF) is a semantic graph-based data model to represent information about the real-world or abstract concepts on the Web[5]. RDF is a W3C standard [88] that specifies syntax, architecture and semantics to describe resources on the Web. The main building block of RDF is a *triple*,

---

[2] `https://lod-cloud.net/`

[3] `https://www.w3.org/2005/Incubator/ssn/`

[4] `https://www.w3.org/TR/vocab-ssn/`

[5] `https://www.w3.org/TR/rdf11-concepts/`

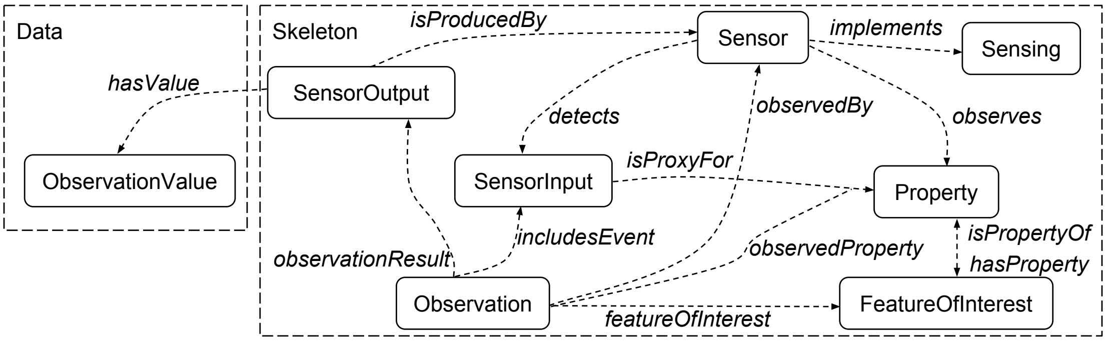


Figure 2.1: **Semantic Sensor Network (SSN) Ontology**. A portion of the SSN ontology representing observations, measurements, sensors, and sensor outputs[4].

which is a positive statement and is composed of a *subject*, a *predicate*, and an *object*, where:

- A *Subject* represents a resource described using *predicate* and *object*, and is a URI or a blank node;
- *Predicate* specifies a property or binary relation that relates the *subject* to the *object* of a triple;
- *Object* denotes a value of the *predicate*.

Formally, an RDF triple is defined a follows [10]:

Definition 2.2.1 (RDF triple [10])**.** *Let* $\mathbf{I}$, $\mathbf{B}$, $\mathbf{L}$ *be disjoint infinite sets of IRIs, blank nodes, and literals, respectively. A tuple* $(s\ p\ o) \in (\mathbf{I} \cup \mathbf{B}) \times \mathbf{I} \times (\mathbf{I} \cup \mathbf{B} \cup \mathbf{L})$ *is an RDF triple, where* s *is the subject,* p *is the property, and* o *is the object.*

Example 2.2.1. *A set of RDF triples is called RDF dataset (or knowledge graph) and can also be viewed as a graph. Thus, in Figure 2.2a, the edge (*`:obs1 rdf:type :TempObs`*) represents an RDF triple, where entity* `:obs1` *corresponds to subject,* `rdf:type` *and* `:TempObs` *represent a property and an object, respectively; there are twenty-seven more RDF triples.*

Definition 2.2.2 (RDF Graph)**.** *An RDF graph* $G = (V, E, L)$ *is a labeled directed graph where nodes represent entities or objects, while labels stand for properties:*

- *An RDF triple* $(s\ p\ o) \in E$*, corresponds to an edge in* E *from node* s *to node* o*;* p *is the label of the edge and denote the property that relates both nodes;*
- $s, o \in V$*,* s *corresponds to a subject and* o *corresponds to an object; and*
- $p \in L$*, is an edge label corresponding to a property.*

Example 2.2.2. *Figure 2.2a presents an RDF graph that corresponds to a portion of the RDF dataset from the storm season in year 2004. Nodes correspond to resources representing observations, measurements, and timestamps. Further, literals are also represented as nodes in the RDF graph and properties typically stem from a variety of RDF vocabularies that include the Semantic Sensor Network (SSN) Ontology. Edges in RDF graphs represent RDF triples and connect the nodes in RDF graphs using properties from the SSN ontology. We ignore prefixes, and replace long URLs by short identifiers for clarity.*

The RDF graph that includes properties from the SSN ontology and the instances correspond to the classes of the SSN ontology, we refer to such an RDF graph in this thesis as an SSN RDF graph. RDF graphs are usually composed of entity description sub-graphs, sometimes also referred to as Concise Bounded Descriptions (CBD)[6]. These subgraphs are named *RDF subject molecules* defined as follows:

Definition 2.2.3 (RDF Molecule [36])**.** *An RDF molecule RM is a set of RDF triples that share the same subject, i.e.,* $RM = (s\ p_1\ o_1),(s\ p_2\ o_2),\ldots,(s\ p_n\ o_n)$.

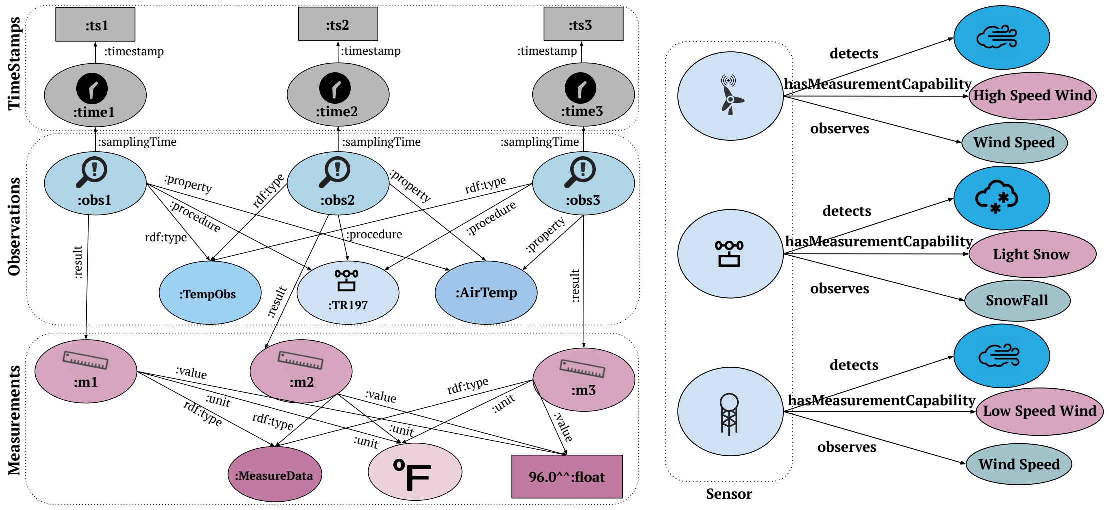
Example 2.2.3. *Figure 2.2b presents an RDF graph with three RDF subject molecules. Each RDF molecule consists of three RDF triples connected to the same subject, which represents an instance of the sensor class. Each instance of the sensor class is described by the RDF triples in terms of input, observed property, and measurement capability. For simplicity, we omit the URIs in the figure, and will refer to RDF subject molecules as molecules in the rest of the thesis.*

2.2.3 RDF Schema

RDF Schema (RDFS) [89] is part of W3C Recommendation and provides a data-modelling vocabulary for RDF data to define semantics of user-defined vocabularies. RDFS is an extension of RDF with additional modeling constructs to define classes (*rdfs:Class*), properties domain (*rdfs:domain*) and range (*rdfs:range*) restrictions, associations (*rdf:type*) of entities in classes, and hierarchies of classes (*rdfs:subClassOf*) and properties (*rdfs:subPropertyOf*). A class represents a set of entities that express a real-world concept. An association of an entity to a class is defined using property *rdf:type*. Class hierarchies are specified using the *rdfs:subClassOf* construct. Entities of classes are associated with RDF triples using properties defined by *rdf:Property*. The types of a subject and an object in a triple, representing the association of classes using a property, are defined with *rdfs:domain* and *rdfs:range* constructs. Property hierarchies are defined using *rdfs:subPropertyOf* construct. In addition, RDFS presents annotation

---

[6] `https://www.w3.org/Submission/CBD/`



(a) RDF Graph G (b) RDF Molecule

Figure 2.2: **Examples of RDF Graph.** The RDF graphs are described using the SSN ontology. (a) RDF graph G has measurements, `:m1`, `:m2`, and `:m3` of `rdf:type` `:MeasureData`, which are linked to corresponding objects using properties `:unit` and `:value`. Observations, `:obs1`,`:obs2` and `:obs3` of `rdf:type` `:TempObs` are linked to related values using properties `:property`, `:procedure`, `:result`, and `:samplingTime`.(b) An RDF graph with three subject molecules in the class Sensor; for simplicity URIs are not presented, and each sensor is associated with only three RDF triples describing sensor input, measurement capability, and observed property.

properties, *rdfs:label* and *rdfs:comment*, to enrich human-readability of a resource or entity. RDFS provides foundations to build RDF graphs with typed hierarchies of concepts and relationships among concepts as specified by *rdfs:subClassOf* and *rdfs:subPropertyOf*, as well as the restrictions defined by *rdfs:domain* and *rdfs:range*. Besides, RDFS offers a set of entailment rules to infer implicit RDF statements from explicit ones[7].

2.2.4 The SPARQL Query Language and SPARQL Protocol

SPARQL[8] query language is recommended by W3C [86] for querying RDF data. SPARQL adopts a graph pattern matching approach to query RDF datasets represented using RDF that is a directed graph data model. SPARQL queries comprise three parts [10]; *pattern matching*, *solution modifiers*, and *output type*.

---

[7] `https://www.w3.org/TR/rdf11-mt/#rdfs-entailment`

[8] SPARQL is a recursive acronym that stands for The SPARQL Protocol and RDF Query Language

The *pattern matching* part contains several features of graphs pattern matching, i.e., optional, union, nesting, filtering values, and choosing data sources to be matched by the graph pattern. The *solution modifier* allows the modification of values, obtained by the *pattern matching* part, using projection, distinct, group, order, and limit operators. The *output type* can be yes/no, values of the variables matching the patterns, construction of new RDF data from matched values, and descriptions of resources. A SPARQL contains a body and a head of form *head* $\longleftarrow$ *body*. The *body* of the query is an RDF graph pattern expression that contains triple patterns, i.e., RDF triples with variables, optional parts, constraints over variables values, and conjunctions. The *head* of the query determines how to construct query answers. SPARQL query evaluation involves two steps. In the first step, the triple patterns in the *body* of the query are matched against the RDF graph to retrieve the bindings for the variables in the *body*. In the second step, the information in the *head* of the query are used and classical relational operators are applied on the bindings to compute final query answers. The SPARQL query language implements *OPT*, *UNION*, *FILTER*, and *AND* operators. To construct graph pattern expressions. SPARQL uses *AND* operator via a point symbol (.). The syntax of SPARQL graph pattern is defined as:

Definition 2.2.4 (SPARQL Graph Pattern Expression [10])**.** *Let Y be an infinite set of variables disjoint from $I \cup B \cup L$. A SPARQL graph pattern expression is defined recursively as follows:*

1. *A triple pattern $t \in (I \cup B \cup Y) \times (I \cup Y) \times (I \cup B \cup L \cup Y)$ is a graph expression,*
2. *If Q_1 and Q_2 are graph patterns, then expressions $(Q_1\ AND\ Q_2)$, $(Q_1\ OPT\ Q_2)$, and $(Q_1\ UNION\ Q_2)$ are graph patterns,*
3. *If Q is a graph pattern and R is a SPARQL built-in filter condition, then the expression $(Q\ FILTER\ R)$ is a graph pattern.*

Example 2.2.4. *The following is a SPARQL graph pattern composed of a set of RDF triples patterns, FILTER, UNION, and AND (.) operators.*

```
{ ?observation om:procedure ?sensor .
  ?observation rdf:type      weather:VisibilityObservation .
  ?observation om:result     ?result.
  ?result      om:value      ?val .
  FILTER ( ?val < "11"^^xsd:float ) # centimeters
}
UNION
{ ?observation om:procedure ?sensor .
  ?observation rdf:type      weather:RainfallObservation .
  ?observation om:result     ?result .
```

```
    ?result         om:value        ?val  .
    FILTER ( ?val > "35"^^xsd:float ) # centimeters
  }
```

The SPARQL query language provides four query forms: SELECT, ASK, CONSTRUCT, and DESCRIBE. These SPARQL query forms use graph pattern bindings to construct result sets or RDF graphs. In this thesis, we focus on SPARQL SELECT query form that is formally defined as follows:

Definition 2.2.5 (SPARQL SELECT Query [95]). *Let Q be a SPARQL expression and let $Z \subset Y$ a finite set of variables. A SPARQL select query is an expression of the form $SELECT_Z(Q)$.*

Example 2.2.5. *The following query represents a SPARQL SELECT query composed of a graph pattern expression, AND, FILTER, and UNION operators, and solution modifier DISTINCT. The query retrieves the data about the observations that measure low visibility values, i.e., $<$ 11 centimeters, and low rainfall values, i.e., $<$ 35 centimeters. The SPARQL query projects unique values matching to the variable ?sensor that correspond to the sensors observing these values.*

```
PREFIX om:<http://knoesis.wright.edu/ssw/sensor-observation.owl#>
PREFIX weather: <http://knoesis.wright.edu/ssw/ont/weather.owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT DISTINCT ?sensor
WHERE {
  { ?observation om:procedure ?sensor .
    ?observation rdf:type       weather:VisibilityObservation  .
    ?observation om:result      ?result.
    ?result         om:value        ?val  .
    FILTER ( ?val < "11"^^xsd:float ) # centimeters
  }
  UNION
  { ?observation om:procedure ?sensor .
    ?observation rdf:type       weather:RainfallObservation  .
    ?observation om:result      ?result  .
    ?result         om:value        ?val  .
    FILTER ( ?val > "35"^^xsd:float ) # centimeters
  }
}
```

SPARQL SELECT queries are evaluated over RDF datasets to find mappings, where each mapping contributes a possible answer to a query. A mapping, μ, is a partial function $\mu : Y \rightarrow (I \cup B \cup L)$ representing the bindings from a set of variables in the query to RDF terms in the RDF data. The domain of μ, $dom(\mu)$ is the subset of Y where mapping μ is defined. Two mappings μ_1 and

μ_2 are compatible, represented as $\mu_1 \sim \mu_2$, if $\mu_1 \cup \mu_2$ is a mapping too, i.e., for all $?x \in dom(\mu_1) \cup dom(\mu_2)$, it is the case that $\mu_1(?x) = \mu_2(?x)$. $\mu(t)$ represents a triple obtained by replacing the variables in a triple t by the values according to the mapping μ [95]. The semantics of SPARQL graph pattern expression are defined as a function $[[.]]_D$; it takes a pattern expression, translates the pattern expression into algebraic operations, and returns a set of mappings. The semantics of SPARQL graph pattern expressions are formally defined as:

Definition 2.2.6 (SPARQL Set Semantics [10, 95]). *Let D be an RDF dataset, t a triple pattern, Q, Q_1, and Q_2 SPARQL expressions, R a filter condition, and $Z \subset Y$ a set of variables. Let $[[.]]_D$ be a function that translates SPARQL expressions into SPARQL algebraic operations as follows:*

$$[[t]]_D = \{\mu | dom(\mu) = vars(t) \wedge \mu(t) \in D\}$$

$$[[Q_1\ AND\ Q_2]]_D = [[Q_1]]_D \bowtie [[Q_2]]_D$$
$$[[Q_1\ OPT\ Q_2]]_D = [[Q_1]]_D ⟕ [[Q_2]]_D$$
$$[[Q_1\ UNION\ Q_2]]_D = [[Q_1]]_D \cup [[Q_2]]_D$$
$$[[Q\ FILTER\ R]]_D = \sigma_R([[Q]]_D)$$
$$[[SELECT_Z(Q)]]_D = \pi_Z([[Q]]_D)$$

The semantics of SPARQL query evaluation are defined as follows:

Definition 2.2.7 (SPARQL Set Algebra [10, 95]). *Let Ω, Ω_1, and Ω_2 be a set of mappings, R denote a filter condition, and $Z \subset Y$ be a finite set of variables. SPARQL algebraic operations join ($\bowtie$), union ($\cup$), minus ($\setminus$), left outer join (⟕), projection (π), and selection (σ) are defined as:*

$$\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 | \mu_1 \in \Omega_1, \mu_2 \in \Omega_2 : \mu_1 \sim \mu_2\}$$

$$\Omega_1 \cup \Omega_2 = \{\mu | \mu \in \Omega_1\ or\ \mu \in \Omega_2\}$$
$$\Omega_1 \setminus \Omega_2 = \{\mu_1 \in \Omega_1 |\ for\ all\ \mu_2 \in \Omega_2 : \mu_1 \not\sim \mu_2\}$$
$$\Omega_1 ⟕ \Omega_2 = (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \setminus \Omega_2)$$
$$\pi_Z(\Omega) = \{\mu_1 | \exists \mu_2 : \mu_1 \cup \mu_2 \in \Omega \wedge dom(\mu_1) \subseteq Z \wedge dom(\mu_2) \cap Z = \emptyset\}$$
$$\sigma_R(\Omega) = \{\mu \in \Omega | \mu \models R\}$$

where $\mu \models R$ iff μ satisfies SPARQL built-in filter condition R.

A set of RDF triples and filters that are related through conjunction is called a SPARQL basic graph pattern (BGP) formally defined as:

Definition 2.2.8 (BGP)**.** *Let I be the set of all IRIs, B be the set of blank nodes, L be the set of literals and Y be the set of variables. A SPARQL basic graph pattern (BGP) expression (an AND-only SPARQL expression) is defined recursively as follows:*

- *A triple pattern $t \in (I \cup B \cup Y) \times (I \cup Y) \times (I \cup B \cup L \cup Y)$ is a BGP;*
- *The expression $(Q_1\ AND\ Q_2)$ is a BGP, where Q_1 and Q_2 are BGPs;*
- *The expression $(Q\ FILTER\ R)$ is a BGP, where Q is a BGP and R is a filter expression that evaluates to Boolean value.*

A SPARQL BGP contains at least one star-shaped subquery (SSQ). An SSQ is a non-empty set of SPARQL triple patterns sharing the same subject variable(constant).

Definition 2.2.9 (Star-shaped Subquery (SSQ) [102])**.** *A star-shaped subquery $star(SQ, ?X)$ on a variable (constant) $?X$ is defined as:*

- *$star(SQ, ?X)$ is a triple pattern $t = (?X\ p\ o)$, where p and o are different to $?X$.*
- *$star(SQ, ?X)$ is the union of two stars, $star(SQ_1, ?X)$ and $star(SQ_2, ?X)$, where triple patterns in SQ_1 and SQ_2 only share the variable (constant) $?X$.*

Example 2.2.6. *The following SPARQL query is composed of a basic graph pattern (BGP) containing five triple patterns, one filter expression, and two star-shaped subqueries (SSQ). The star-shaped subqueries are over the variables ?result and ?observation at the subject positions of the triple patterns in the query.*

```
PREFIX om:<http://knoesis.wright.edu/ssw/sensor-observation.owl#>
PREFIX weather: <http://knoesis.wright.edu/ssw/ont/weather.owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT DISTINCT ?sensor
WHERE {
  ?observation om:procedure ?sensor .
  ?observation rdf:type     weather:TemperatureObservation .
  ?observation om:result    ?result.
  ?result      om:value     ?val .
  ?result      om:unit      ?uom .
  FILTER ( ?val < "20"^^xsd:float ) # fahrenheit:
}
```

Complexity of SPARQL

The evaluation of SPARQL graph patterns can be considered as a decision problem and defined as [78]:

INPUT: An RDF dataset D, a graph pattern Q, and a mapping μ.
QUESTION: Is $\mu \in [[Q]]_D$?

The complexity of SPARQL query evaluation is influenced by the type operators, i.e., AND, FILTER, UNION, and OPTIONAL used in the graph pattern. A SPARQL query is evaluated in *PTIME*, i.e., $O(|Q|.|D|)$ time, if the graph pattern expression Q contains triple patterns associated with AND (and optionally FILTER) operator. Likewise, if a SPARQL graph pattern expression Q, is composed of only UNION (and optionally FILTER) operator, the query evaluation problem is in *PTIME*. However, if the graph pattern Q combines AND, UNION, and optionally FILTER operators, the evaluation problem is *NP-complete*. The SPARQL queries with OPTIONAL operator, in combination to any of the above operators, becomes *PSPACE-complete* [95].

2.2.5 C-SPARQL - A Continuous SPARQL Query Language

C-SPARQL [16] is an extension of SPARQL to support continuous queries that are continuously executed over RDF data streams. C-SPARQL extends SPARQL in such a ways that a regular SPARQL query is also a C-SPARQL query. C-SPARQL is presented with full specifications of the syntax and formal semantics along with RDF streams as a data type.

RDF Stream data type

C-SPARQL provides RDF streams as new data types supported by SPARQL. An RDF stream is an ordered sequence of pairs, where each pair is composed of an RDF triple and its timestamp τ:

$$\ldots$$

$$((s_i\ p_i\ o_i), \tau_i)$$

$$((s_{i+1}\ p_{i+1}\ o_{i+1}), \tau_{i+1})$$

$$\ldots$$

RDF triples are annotated with timestamps. These timestamps are monotonically non-decreasing ($\tau_i \leq \tau_{i+1}$) in the data stream. Timestamps should not be unique, therefore, they are not strictly increasing. Any number of RDF triples in the stream can occur at the same time, although their sequence can be different.

Windows

RDF streams carrying RDF data need to be able to identify the data sources and also select criteria over them. To identify the data sources, each data stream is associated with a unique IRI, that serves as a locator of the data stream source. An IRI represents an IP address and a port to access streaming data. Since data streams are infinite, the notion of windows over data streams is defined. C-SPARQL uses FROM STREAM clause to express identification and windowing over stream [16]:

```
FromStrClause → FROM [ NAMED ] STREAM StreamIRI [ RANGE Window ]
Window → LogicalWindow | PhysicalWindow
LogicalWindow → Number TimeUnit WindowOverlap
TimeUnit → ms | s | m | h | d
WindowOverlap → STEP Number TimeUnit | TUMBLING
PhysicalWindow → TRIPLES Number
```

A window retrieves the last data stream elements from a stream that are processed by the query. The extraction of data elements can be physical or logical. Physical extraction involves a given number of triples to be extracted, whereas logical extraction retrieves a variable number of triple within a give time interval. Logical windows are sliding [41] if they progressively advance by a given STEP that is smaller than the window's time interval. Logical windows are non-overlapping or TUMBLING when they advance equal to their time interval. The optional NAMED clause is equivalent to SPARQL FROM clause, and binds the stream IRI to a variable that is used to access the stream through the GRAPH clause.

Query Registration

Continuous queries in C-SPARQL include at least one FROM STREAM clause, and are registered to produce continuous outputs as variable bindings tables or graphs. C-SPARQL queries are registered using the statement as follows [16]:

```
Registration → REGISTER QUERY QueryName
                    [ COMPUTED EVERY Number TimeUnit] AS Query
```

The COMPUTE EVERY clause defines the frequency to compute the query over the stream, otherwise, the system determines the frequency automatically. Moreover, streams are registered for CONSTRUCT and DESCRIBE queries to generate RDF triples associated with timestamps. Stream are registered using the following clause [16]:

```
Registration → REGISTER STREAM QueryName
               [ COMPUTED EVERY Number TimeUnit ] AS Query
```

Example 2.2.7. *The following C-SPARQL query is registered as query Q and is composed of a basic graph pattern (BGP) containing five triple patterns and one filter expression. The query retrieves temperature observations from the given data stream within a tumbling winding of size one hour.*

```
REGISTER QUERY Q AS
PREFIX om:<http://knoesis.wright.edu/ssw/sensor-observation.owl#>
PREFIX weather: <http://knoesis.wright.edu/ssw/ont/weather.owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT DISTINCT ?sensor ?val
FROM NAMED STREAM <http://www.cwi.nl/SRBench/observations>
                  [RANGE 1h TUMBLING]
WHERE {
   ?observation om:procedure ?sensor .
   ?observation rdf:type     weather:TemperatureObservation .
   ?observation om:result    ?result.
   ?result      om:value     ?val .
   ?result      om:unit      ?uom .
  FILTER ( ?val < "20"^^xsd:float ) # fahrenheit:
}
```

Aggregation and Timestamp function

C-SPARQL provides multiple independent aggregation functions within the same query, and the syntax is as follows [16]:

```
AggregateClause →( AGGREGATE { ( var , Function , Group ) [Filter] } )*
Function → COUNT | SUM | AVG | MIN | MAX
Group → var | { var (, var )* }
```

An aggregation clause consists of three parts; a new variable (not occurring in the WHERE or aggregation clauses), an aggregation function, and a set of one or more variables to define grouping criteria. The timestamp function returns the timestamp of the RDF stream element that produces the bindings. Timestamp function takes two arguments, the first is the name of a variable in the WHERE clause, and second is the stream IRI. Based on these constructs of C-SPARQL query, given below is an example query:

Example 2.2.8. *The following C-SPARQL query computes average wind speed and temperature values observed by the sensors in variables ?averageWindSpeed and ?averageTemperature, respectively. These values are computed within a sliding window of size one hour and the sliding interval is ten minutes.*

```
PREFIX om:<http://knoesis.wright.edu/ssw/sensor-observation.owl#>
PREFIX weather: <http://knoesis.wright.edu/ssw/ont/weather.owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT ?sensor (AVG(?windSpeed) AS ?averageWindSpeed)
               (AVG(?temperature) AS ?averageTemperature)
FROM NAMED STREAM <http://www.cwi.nl/SRBench/observations>
              [RANGE 1h STEP 10m]
WHERE {
   ?tempObservation        om:procedure ?sensor .
   ?tempObservation        rdf:type     weather:TemperatureObservation.
   ?tempObservation        om:result    ?temperatureResult .
   ?temperatureResult      om:value     ?temperature .
   ?temperatureResult      om:unit      ?uom .
   FILTER(?temperature >   "32"^^xsd:float &&
                              REGEX(STR(?uom), "fahrenheit", "i"))
   ?windSpeedObservation om:procedure ?sensor .
   ?windSpeedObservation rdf:type     weather:WindSpeedObservation .
   ?windSpeedObservation om:result    ?windresult .
   ?windresult           om:value     ?windSpeed .
}
GROUP BY ?sensor
```

The formal semantics of C-SPARQL are build by extending the semantics of SPARQL [78]. C-SPARQL provides formal semantics of aggregate, windows, and the timestamp function. C-SPARQL introduces a new binary operator AGG to compute aggregates. An aggregate pattern is defined as $A(y, f, b, J)$, where y is new variable, f represents an aggregate function, b is parameter of f, and J is a set of grouping variables. SPARQL set semantics given in Definition 2.2.6 are extended to add semantics for aggregate evaluation as follows [16]:

$$[[Q\ AGG\ A]]_D = [[Q]]_D \mathbin{⟕} [[A]]_D, where\ A(y_a, f_a, b_a, J_a)\ is\ an\ aggregate\ pattern.$$

The evaluation of aggregation pattern $[[A]]_D$ is computed from mappings $\mu_a : Y \to (I \cup B \cup L)$, where domain of μ_a is $dom(\mu_a) = y_a \cup J_a$, and $deg(\mu_a) = deg(J_a) + deg(y_a) = deg(J_a) + 1$, where $deg(\mu_a)$, $deg(J_a)$, and $deg(y_a)$ represent cardinalities of μ_a, J_a, and y_a, respectively. An RDF stream is defined as [16]:

$R = \{((s\ p\ o), \tau) | (s\ p\ o) \in (I \cup B) \times I \times (I \cup B \cup L), \tau \in \mathbb{T}\}$ where $\mathbb{T}$ is the infinite set of timestamps. A logical window ω_l over the RDF stream R is defined as:

$$\omega_l(R, ts_i, ts_f) = \{((s\ p\ o), \tau) \in R | ts_i < \tau \leq ts_f\}.$$

If $c(R, ts_i, ts_f) = |\{((s\ p\ o), \tau) \in R | ts_i < \tau \leq ts_f\}|$ is a function that counts the items in R having timestamps in the range $(ts_i, ts_f]$, then a physical window ω_p is defined as [16]:

$$\omega_l(R, n) = \{((s\ p\ o), \tau) \in \omega_l(R, ts_i, ts_f) | c(R, ts_i, ts_f) = n\}.$$

A sliding window ω can have range ρ and step σ. A logical window will have ρ and σ as time intervals, whereas a physical window will have ρ and σ as integers values. A variable y in C-SPARQL can have bindings from static and streaming RDF data. The bindings of y from data stream contains a timestamp of the RDF triple that matches to one of the triple patterns $t \in Q$ such that $y \in dom(t)$. The set of timestamps associated with a variable y in a triple pattern t is represented as $TS_{set}(y, t)$, and the set of timestamps related with the variable y in a graph pattern Q as [16]:

$$TS_{set}(y, Q) = \{\tau | t \in Q \wedge y \in dom(t) \wedge \tau \in TS_{set}(y, t)\}.$$

Based on these values of timestamps for a variable, the timestamp function returns the highest timestamp among all the bindings of y, and is defined as:

$$ts(y, Q) = max(TS_{set}(y, Q)))$$

2.2.6 SPARQL Query Processing

To support the execution of complex SPARQL queries Lampo et al. [60] implement optimized query execution techniques that enhance the performance of the state-of-the-art RDF-3X [72] SPARQL query engine. The compressed indexed structures, caching, optimization and execution techniques in RDF-3X are able to efficiently process the real-world complex SPARQL queries. Furthermore, RDF-3X provides techniques for efficient usage of previously loaded intermediate results in cache. Lampo et al. [60] implement execution techniques to fully exploit the RDF-3X caching features for small-sized star-shaped groups of SPARQL graph patterns, when these star-shaped queries are run in both cold and warm caches, i.e., when intermediate results are stored or not in cache. A star-shaped query joins multiple basic graph patterns that share exactly one variable. Lampo et al. implement four different operators to retrieve and combine intermediate RDF triples generated for small-sized star shaped groups. **1)** *Index Nested-Loop Join* (**njoin**) finds mappings for the first triple pattern, and uses these mappings to instantiate the variables in the second triple pattern. njoin operator uses data indices and the instantiations of the second triple pattern to speed up execution task. This operator is useful when the number of intermediate results is small. **2)** *Group Join* (**gjoin**) evaluates each of the two groups independently, and combines the results to find the compatible mappings. Since, the intermediate results previously loaded in cache can be reused without executing the operation to compute them, therefore, gjoin operator

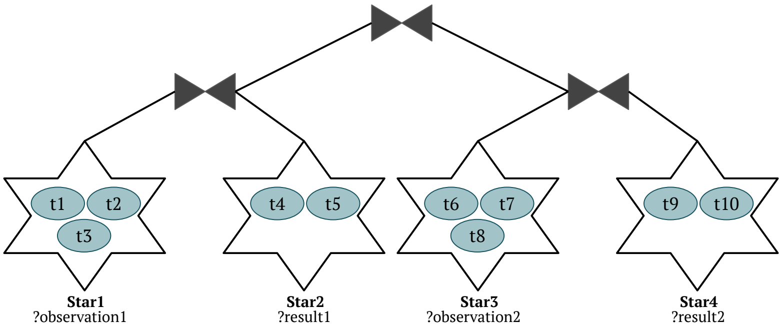


Figure 2.3: **An Execution Plan for Star-Shaped Groups**. An execution plan for the SPARQL query in Listings 2.1. The stars `Star1`, `Star2`, `Star3`, and `Star4` represent star-shaped groups over the variables `?observation1`, `?result1`, `?observation2`, and `?result2`, respectively.

takes advantage of cache. To avoid page faults, the size of the intermediate results should be small. **3)** *Star-Shaped Group Evaluation* (**sgroup**) evaluates the first pattern in the star-shaped group and identifies the instantiations of the shared variable. The rest of the patterns in the group are bind using these instantiations. If the first pattern is very selective, then this operator can be very efficient. **4)** *Index Star-Shaped Group Evaluation* (**isgroup**) builds indices for all the patterns in a group. The instantiations of each pattern are obtained independently and are merged to produce the answers. This operator benefits running in warm cache if the number of instantiations are small that are used to compute join between two basic graph patterns, and cached results can be reused to compute join with the third graph pattern.

Listing 2.1: A SPARQL Query

```
PREFIX om:<http://knoesis.wright.edu/ssw/sensor-observation.owl#>
PREFIX weather: <http://knoesis.wright.edu/ssw/ont/weather.owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT DISTINCT ?sensor
WHERE {
   ?observation1 om:procedure ?sensor .                          #t1
   ?observation1 rdf:type     weather:VisibilityObservation .    #t2
   ?observation1 om:result    ?result1.                          #t3
   ?result1      om:value     ?val1 .                            #t4
   ?result1      om:unit      ?uom1 .                            #t5
   ?observation2 om:procedure ?sensor .                          #t6
   ?observation2 rdf:type     weather:RainfallObservation .      #t7
   ?observation2 om:result    ?result2 .                         #t8
   ?result2      om:value     ?val2 .                            #t9
   ?result2      om:unit      ?uom2 .                            #t10
}
```

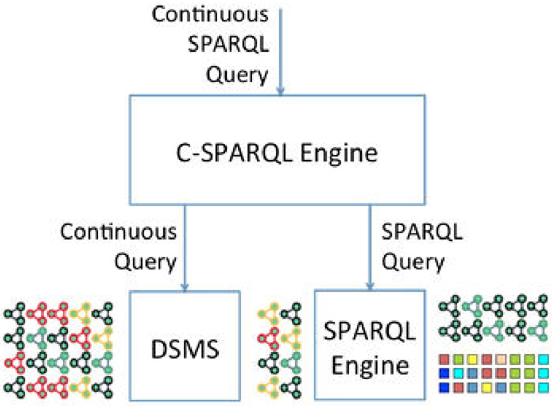


Figure 2.4: **C-SPARQL Engine Architecture**. The C-SPARQL engine architecture[9] relies on existing DSMS and SPARQL processing frameworks. The C-SPARQL engine receives a continuous SPARQL query and translates the query into a static and a continuous query. The static query is passed to the SPARQL engine, whereas the continuous query is executed by DSMS over the data stream.

A SPARQL query composed of ten RDF triple patterns is presented in Listings 2.1. The query contains four star-shaped basic graph patterns over the variables `?observation1`, `?result1`, `?observation2`, and `?result2`. Figure 2.3 presents an execution plan for the SPARQL query around the star-shaped groups. `Star1` encompasses the triple patterns in the star-shaped group around the variable `?observation1`. Similarly, `Star2`, `Star3`, and `Star4` represent star-shaped groups around the variables `?result1`, `?observation2`, and `?result2`, respectively. The stars `Star1` and `Star2` are joined on variable `?result1`, and the stars `Star3` and `Star4` are joined on variable `?result2`. The final results of the query are obtained by computing the join on variable `?sensor` over the results of the previously computed joins.

2.2.7 C-SPARQL Query Processing

The execution framework [15] for C-SPARQL queries integrates the existing Data Stream Management Systems (DSMS) [39] and SPARQL systems to run continuous queries over RDF data streams, as well as over historical RDF data that contain static knowledge. The existing stream processing frameworks, such as Aurora/Borealis [1], STREAM [9], and Stream Mill [12] which are highly optimized for processing relational streams, are used to provide a plug-in architecture for C-SPARQL processing. Figure 2.4 shows the C-SPARQL engine architecture that relies on existing technologies. SPARQL engine evaluates the static part of the

---

[9] `https://www.w3.org/community/rsp/wiki/File:C-sparql-engine.jpg`

C-SPARQL query, and the stream management system (DSMS) evaluates streams and aggregates. A query parser parses C-SPARQL query and forwards to the orchestrator that translates the C-SPARQL query into a static and dynamic part. The static query is executed over the RDF data using SPARQL reasoner, whereas dynamic is passed to DSMS for continuous evaluation. The aggregate clause in C-SPARQL query have different semantics than SQL aggregate semantics, therefore, it is not directly translated into an SQL aggregate function. The aggregates are computed in a separate view and then an outer join is performed with the retrieved result to generate final answers.

2.3 Summary

Integration of IoT data, containing an enormous number of observations continuously generated from IoT devices, and the particular research challenges presented in Chapter 1 require comprehensive solutions from different angles. The concepts and the existing technologies presented in this chapter establish a solid foundation to address the posed challenges. Data integration system and the related concepts presented in Section 2.1 define the foundations for integrating large volumes of IoT data, containing plenty of repeatedly observed measurements, from heterogeneous data sources. The Resource Description Framework (RDF) described in Section 2.2 provides formalism for representing the knowledge that is encoded in observations produced by diverse IoT devices. RDF is a machine-readable language with explicit semantics and exploits a semi-structured graph-based data model that is more flexible than the rigid relational data models. We leverage these characteristics of RDF data model and provide techniques for efficient representations of IoT observational data and the meaning encoded in the data, for answering the **RQ1**. The observational data from diverse IoT devices can be processed as streaming data in real time or can be collected and stored as historical data and processed later. SPARQL and C-SPARQL are the query languages to process and query historical and streaming RDF data, respectively. RDF-3X is an RDF engine that exploits the query processing techniques to benefit from caching the data during SPARQL query execution. We exploit the proposed RDF representations to scale up to large RDF sensor datasets, these RDF representations are processed using RDF-3X engine to address the **RQ2**. C-SPARQL is an extension of SPARQL query language to process the streaming RDF data generated in the form of RDF data streams. The C-SPARQL query engine exploits the existing Data Stream Management Systems (DSMS) and SPARQL query engines to process continuous SPARQL queries over RDF data streams. We present on-demand knowledge graph generation techniques that rely on C-SPARQL engines, to answer the research questions **RQ3** and **RQ4**.

Chapter 3

Related Work

In this chapter, we present a detailed analysis of the state-of-the-art approaches related to the main research problems and research questions defined in Chapter 1. We initially present the topics, shown in Figure 3.1, identified for the review of existing approaches. These topics include state-of-the-art approaches proposed by research communities to solve the issues related to the data representation during data integration and semantic description. Section 3.1 presents state-of-the-art approaches for frequent pattern mining. Although these approaches are able to extract frequent isomorphic graph patterns from a graph, however, they are not able to identify frequent star patterns shared among several entities. Section 3.2 presents the existing approaches for efficient representations of data using different data models. Furthermore, query optimization techniques, exploiting efficient representations of data, are discussed. In addition, existing approaches that utilize Big Data frameworks for efficient processing of large-scale data are explored. In section 3.3, the state-of-the-art techniques for semantic description and processing of streaming data are elaborated.

3.1 Frequent Pattern Mining Approaches

The problem of frequent pattern mining involves finding subgraphs, from a graph, that have frequency above a given threshold. gSpan [105] exploits the depth first search (DFS) to mine frequent patterns. gSpan maps a graph to a DFS code representing the edges sequence. Several DFS codes can be generated for a single graph. These DFS codes are ordered lexicographically based on the edge labels and the order of nodes being visited. From these ordered DFS codes the minimum DFS codes are selected to build the DFS tree. DFS over a code tree discovers all the minimum DFS codes of frequent patterns. GRAMI [33] performs frequent pattern mining and finds only the minimal set of instances that satisfy the given

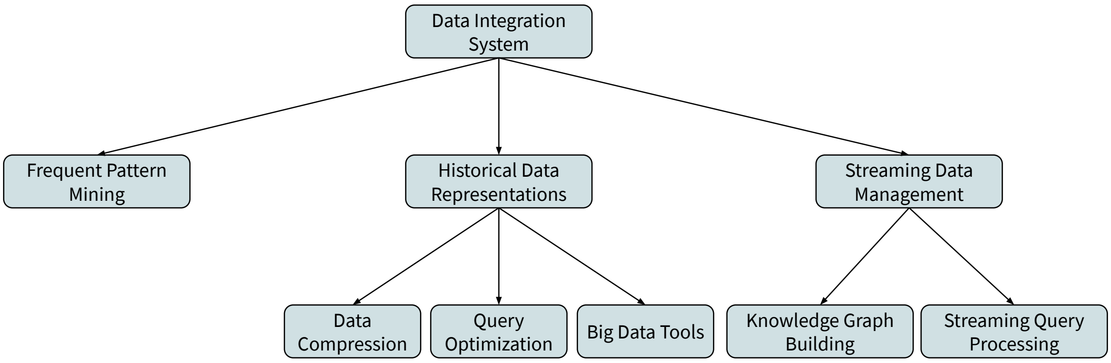


Figure 3.1: **Categories of the state-of-the-art Approaches**. The related work presented in this thesis is categorized under three topics; Frequent Pattern Mining, Historical Data Representations, and Streaming Data Management. The frequent pattern mining category covers the existing approaches that detect frequent patterns in data. The historical data representation topic, encompasses existing techniques related to data compression and query optimization, and utilizing Big Data frameworks for efficiently processing large-scale historical data. The streaming data management topic describes existing approaches related to the knowledge graph building from streaming data, and query processing over streaming data.

frequency threshold. GRAMI stores the templates of frequent patterns instead of storing their appearances. This avoids the creation and storage of all appearances of patterns. For frequency evaluation, GRAMI maps the frequent patterns mining problem to constraint satisfaction problem (CSP), which is represented by a tuple; (a) an ordered set of variables representing nodes, (b) a set of domains of variables in (a), and (c) a set of constrains between these variables. Two subgraphs patterns are isomorphic if the variables in corresponding CSP tuple have different values from the domains, however, nodes and edge labels are the same. Notwithstanding these frequent pattern mining approaches are able to identify the frequent isomorphic graph patterns, extracting frequent star patterns, which involve different subject nodes related with same objects nodes using same set of edge labels, requires an exhaustive search over the identified frequent patterns. It is important to highlight that although these approaches effectively mine subgraph patterns, they are not able to identify graph patterns where a node shared among several edges is a variable. In this thesis, we present an approach that searches for star patterns and is able to detect the ones with highest instantiations.

3.2 Historical Data Representations

3.2.1 Data Compression Approaches

Database and Semantic Web communities have proposed several efficient representations to speed up processing over the large amounts of data represented using relational and graph data models [2, 6, 8, 20, 37, 42, 51, 55, 63, 69, 71, 83, 108]. These compression approaches can be categorized into compression techniques for relational, graph and RDF graph data models. Relational data model approaches [2, 108] efficiently store huge datasets in column-oriented stores. For efficient storage and processing of graph data, approaches such as Neo4j, Sparksee [69], and technique by Lehmann et al. [63], are proposed. Approaches [7, 8, 20, 37, 51, 55, 71, 75, 83] target the efficient storage of RDF graph data.

Data Compression for Relational Data Models

Column-oriented databases [98, 108] store each attribute in a separate column such that successive values of the attribute are accumulated consecutively on the disk. This improves the query processing when the values of some of the columns are required to process the query. The column oriented data storage opens a number of opportunities to apply compression techniques more naturally over the multiple values of the same type. Compression approach proposed by Abadi et al. [2] compress each column in C-Store [98] using one of the methods like Null Suppression, Dictionary Encoding, Run-length Encoding, Bit-Vector Encoding or Lempel-Ziv [90, 104]. Zukowski et al. [108] focus on improving bad CPU/cache performance caused by the compression techniques involving if-then-else statements in the code, e.g., Null Suppression, Run-length Encoding, and does not take advantage of the super-scalar properties, e.g., pipe-lining the processes, in the modern CPUs. Zukowski et al. propose three compression methods, i.e., PFOR, PFOR-DELTA, and PDICT. PFOR expresses values as positive offsets from a base value, PFOR-DELTA represents values as differences from some frame of reference, and PDICT creates a dictionary for the array position of values. These compression solutions are exploited by column-oriented stores using the decomposition storage model [29], where *n-array* relations are decomposed into n binary relations. Each binary relation consists of one attribute values and the corresponding surrogate keys. In this model, two copies of data are stored increasing the data storage requirements. Further, for each attribute a copy of the corresponding duplicated surrogate key is required resulting in an increase of the storage by a factor of two. Moreover, various compression techniques for a large number of unique values, i.e., subject entities, are hard to implement. In this thesis, we present an approach that generates a factorized graph where entities matching a frequent star pattern are represented by a surrogate entity of the corresponding compact RDF molecule. These compact graph representations replace repeated properties and corresponding objects with the properties and objects in the compact RDF molecules, hence, improve the storage space requirements for the decomposition storage model [29].

Data Compression for Graph Data Models

A compact representation of graph data to perform two-way regular-path queries is presented by Lehmann et al. [63]. The approach exploit k^2-tree [23] structures to compress graph data. k^2-tree is a tree shaped structure to represent a graph by utilizing the sparseness and clustering features of the adjacency matrix associated with the graph. k^2-tree divides the adjacency matrix into k^2 submatrices of the same size. Each submatrix, which is child of the root, with at least one 1, from the adjacency matrix, is recursively divided into submatrices until the last level of the tree, having the element corresponding to actual value of the matrix cell, is reached. Lehmann et al. simplify the mappings between the graph nodes and the adjacency matrix by exploiting dictionary encoding to map node ID and edge in the graph to integers. Thus, whole graph is represented in a compact way as an array of k^2-tree. k^2-tree structures resort to the sparseness and clustering features of adjacency matrix associated with a graph to generate compact binary representation of the graph. The binary representations of graphs require a customized engine to process queries. In this thesis, we present an approach that generates a factorized knowledge graph where number of connections between nodes are reduced resulting in a faster graph navigation required in two-way regular path queries. Moreover, the factorized knowledge graphs have simplified structures where the number of node neighbours are reduced by replacing edges between the entities and the objects by the edges connecting the the entities to the surrogate entities in the compact RDF molecules, this ultimately can help improve the retrieval of the neighbours of a node in k^2-tree structures exploited by Lehmann et al. Neo4j has been used to efficiently store data graphs [42]. Neo4j stores nodes and edges in graphs as file records of fixed size represented with unique IDs corresponding to the offset of the associated file position. Furthermore, the edges are stored in doubly-linked lists, and a node record denotes the first edge in the doubly-linked list. In order to retrieve other edges the doubly-linked lists has to be traversed. Hence, visiting graph nodes in Neo4j is costly since it requires all the edges to be traversed, also visiting the neighbour of a node in a graph is expensive especially when the node degree is soaring [66]. This thesis presents an approach that implements compact RDF molecules and reduces the edges incident on a node involved in the frequent graph patterns in a graph, therefore, can help in reducing the traversal time in Neo4j. Sparksee [69] represents graphs using small structures to cache the significant part of data in minimum memory space to speed-up graph operations, efficient edge traversal, and query processing. These data structures utilize multiple indexes for the edges of the nodes and use key-value maps and bitmaps to reduce memory requirements. Although Sparksee promises to efficiently process operations over graphs by utilizing small memory and making the use of data caching techniques, the expected performance is not achieved when the graphs are huge in size. Similarly, Neo4j do not exhibit good performance over huge graphs due to the large volume of intermediate results [44]. The approach, proposed in this thesis, exploits compact RDF molecules to store the frequent graph patterns in RDF graphs only once, which can help to reduce the size of intermediate results and improve cache buffer.

Data Compression for the RDF Data Model

A user specific approach to minimize RDF graphs by defining Datalog rules to remove the irrelevant RDF data is proposed by Meier [71]. The approach uses constraints to maintain data consistency before and after RDF graph minimization. Instead of generating new RDF triples, Datalog rules defined by the user are used to remove RDF data from RDF graphs that is not required for the application. These rules are used to reconstruct the deleted data. Similarly, Pichler et al. [83] study the redundancy elimination on RDF graphs in the presence of rules, constraints, and queries specified by users. These two approaches are user specific and require human input for compressing the ever growing RDF graphs. A scalable lossless RDF compression technique, proposed by Joshi et al. [51], automatically generates decompression rules. The compression approach uses the rules to split the RDF datasets into smaller disjoint datasets; an active and a dormant dataset. A dormant dataset contains uncompressed RDF triples, whereas an active dataset contains compressed triples. To decompress the compressed datasets, decompression rules are applied to active datasets for inferring new triples. This technique requires the overhead of decompression over the compressed data to access the information initially represented in datasets. A factorized representation of RDF graphs is presented by Karim et al. [55], where repeated observation values are represented only once. This approach reduces the number of RDF triples that correspond to observational data described using the Semantic Sensor Network (SSN) Ontology [28]. In this thesis, we propose frequent graph pattern detection approach to automatically identify the frequent star patterns in RDF graphs described using any ontology. Further, we devise factorized graphical representations of RDF graphs which do not require data decompression to perform data management tasks. Moreover, the RDF graph factorization techniques, proposed in this thesis, are independent of the ontology describing the data and able to generate factorized representations of RDF graphs describing datasets using any ontology.

A binary RDF representation format consisting of a Header, a Dictionary and a Triple component is presented by Fernández et al. [37]. The header contains metadata describing the RDF datasets, the dictionary provides a catalog of the RDF terms (URIs, literals, blank nodes) in the RDF datasets and assigns a unique short ID to each RDF term, and the triple component compactly encodes the RDF triples by replacing long repeated RDF terms with the short IDs. Pan et al. [75] propose RDF compression based on graph patterns, which reduces the number of RDF triples and then generates compact binary representations of the reduced triples. Graph pattern-based logical compression replaces the instances of the bigger graph patterns by the instances of the smaller graph patterns. For each graph pattern, the Graph pattern-based serialization generates a sequence of bits containing the graph pattern itself and a sequence of graph pattern instances, where each instance is a list of resource IDs. The compression technique k^2-triples presented by Álvarez-García et al. [8] exploits the two dimensional k^2-trees structure, proposed by Barisaboa et al. [23], to distribute the compact triples obtained by Header-Dictionary-Triples partitioning [37]. A native RDF engine runs queries over the k^2-triples. Bok et al. [20] present RDF provenance compression technique using PROV model based on graph patterns. The proposed approach reduces the space by converting strings in provenance data into numeric data using dictionary encoding. The numeric version of RDF triples are used to extract the frequently repeated subgraphs based on the activities in the model. These frequent subgraphs are referred as compressed subgraphs and are stored as a reference pattern. These approaches are able to effectively reduce redundancies in RDF graphs, and provide effective techniques for RDF graph compression. However, customized engines are required to perform query processing over the compressed RDF graphs, and decompression techniques are needed during data management. In this thesis, we devise factorization techniques that utilize the semantics encoded in the RDF data. The proposed techniques are able to compactly represent RDF triples, reduce redundancy, and facilitate data management tasks without requiring any decompression and a need for a customized engine.

3.2.2 Data Compression based Query Optimization

Factorization techniques have been utilized for optimization of relational data and SQL query processing [14, 13]. Existing approaches proposed compact representations of relational data, obtained by applying logical axioms of relational algebra, e.g., distributivity of product over union, and commutativity of product and union. Bakibayev et al. [13] present an in-memory query engine to run select-project-join queries over factorized data. The query results are expressed using factorized representations in terms of singleton relations, product, and union. The compact representations are obtained by algebraic factorization using distributivity of product over union and the commutativity of product and union. These factorized representations form a nested structure containing attributes from the schema, and are referred as factorization tree. A set of operators for selection and projection are proposed that map the factorized representations and generate efficient query plans. Similarly, Bakibayev et al. [14] improve the performance of relational processing for aggregate and ordering queries using the distributivity of product over union to factorize relations as in the factorization of logic functions [22]. Factorized representations reduce the number of computations required for the evaluation of aggregation functions, i.e., sum, count, avg, min, max, likewise evaluation of aggregation functions as sequences of partial aggregations over the factorized representation speedup the processing. To evaluate order-by-queries, factorized tables are restructured with a constant delay enumeration. Therefore, queries can be executed in factorized relational data, and efficient execution plans can be found to speed up execution time. In this thesis, we build on these experimental results and proposed factorization technique tailored for semantically described sensor data. The *CSSD* approach exploits the semantics encoded in RDF and sensor data, such that, is able to compactly represent RDF triples, reduce redundancy, and facilitate query execution.

3.2.3 Big Data Tools and RDF

With the tremendous growth of semantic data, the problem of storing and processing large-scale semantic data has become of paramount importance. The Semantic Web researchers are exploiting Big Data frameworks to efficiently store and process the continuously growing RDF data [32, 57, 68, 74, 76, 87, 92, 93, 94]. These approaches offer RDF storage and indexing schemes for efficient RDF data processing over parallel processing frameworks. Mami et al. [68] presents relational representation of RDF data over Big Data storage technologies, i.e., Parquet and MongoDB. Mami et al. [68] create a table for each class in RDF data, where all class properties are represented as attributes in the table, additionally, one more table for each class is created and a type column containing type of each attribute is added. Du et al.[32] combine distributed file storage framework Hadoop and RDF triple store Sesame to achieve scalable data analysis for RDF. The Sesame triple store is installed on each Hadoop node and RDF datasets are partitioned in such a way that all the triples with the same predicate are allocated to the same partition. Jena-HBase [57] provides a variety of RDF data storage layouts for HBase and all operations over RDF graph are converted into underlying layout operations. The RDF storage layouts include the simple (three tables each with an index over subject, predicate, and object), vertical partitioned, indexed, vertical partitioned and indexed, hybrid, and hash layouts. Schätzle et al. [94] utilize Parquet columnar storage format for RDF data storage over Hadoop, and SPARQL queries are compiled as SQL queries to be executed using Impala. A single table containing all RDF properties as the attributes of the table is used to store RDF data. Schätzle et al. [93] present PigSPARQL, a SPARQL query processing framework using Hadoop MapReduce over large RDF graphs. The proposed data model stores an RDF graph as a collection of RDF triples, where each triple is represented as a tuple, i.e., subject, predicate, and object, with schema (s, p, o). RDF triples are converted into the proposed data model on the fly, and the framework is particularly suited for the ad-hoc query processing.

A scalable RDF data management system that utilizes Accumulo, a Google Bigtable variant is developed by Punnoose et al. [87]. Punnoose et al. present storage methods, indexing, and query processing techniques while providing fast access to the data using conventional query languages, i.e., SPARQL. A pair of a key and a corresponding value are used to store RDF data. The key consists of RowID, column and Timestamp. To store RDF data, RDF triples are indexed in three tables using the permutations, SPO, POS, and OSP, of the triple pattern. The triples are stored in these three tables in RowID. The storage is compact because RDF triples are stored only on RowID without storing the data in column, timestamp, and value fields. Papailiou et al. [76] present an RDF store to efficiently perform distributed Merge and Sort-Merge joins using multiple indexing over HBase. Indexes are created for all possible combinations of the triple patterns and are maintained in the key part of the HBase table. Furthermore, indexes are compressed by using dictionary encoding to map long URIs with the short strings. Nie et al. [74] study the efficient RDF partitioning and indexing schemes to process RDF data in distributed way using MapReduce. Horizontal partitioning partitions RDF data in such a way that all the triples with the common hash value of subject are contained in the same file. Vertical partitioning combines all the triples with the same predicate in the same file. Clustered property partitioning organizes RDF triples with the same subjects and then partitions RDF triples into clusters using the same property sets. Schätzle et al. [92] present RDF storage schema for HBase to efficiently process joins using MapReduce. Two tables are used to store RDF data; one with subject as row key and the other with object as the row key. HBase filter API filters the data on server side to avoid unnecessary data transfer.

In this thesis, we propose factorization techniques for the RDF sensor data where the RDF triples related to the redundant values are factorized. The proposed tabular representations of factorized RDF graphs, i.e., factorized tables and RDF-MT based tables, scale up to large datasets by leveraging the column-oriented Parquet storage format for HDFS. The tabular representation of the factorized RDF graphs remove the data redundancies in the tabular representations and help in improving the storage and query processing using Big data tools.

3.3 Streaming Data Management

3.3.1 Building Knowledge Graph On-Demand

Rapidly growing amount of data being produced by heterogeneous data sources, has gained the attention of Semantic Web researchers who focus on adding semantics and providing an integrated view of this gigantic and diverse data. Approaches such as GoT [82] and FuhSen [26] provide an integrated view of heterogeneous data sources; they exploit semantics within the data to create an integrated knowledge graph using Semantic Web technologies. Graph of Things (GoT) [82] addresses various challenges in data integration posed by dynamic raw IoT data in order to provide a deeper understanding of data generated by IoT streaming sources. GoT proposes an architecture to provide an integrated view of heterogeneous data from multiple disparate data sources, and allows for querying and exploration of dynamic IoT data sources along with static data. Integrated knowledge graph includes semantic descriptions of sensor data and data from social media using the Semantic Sensor Network (SSN) ontology [28] and an extended version of SSN, respectively. GoT utilizes NLP tools to recognize different data entities and extract metadata about them to provide their contextual information within the unified knowledge graph. GoT exploits Big Data distributed technologies to handle large amounts of IoT stream data. A large number of sensors and other IoT devices continuously produce huge volume of IoT stream data that is many folds larger than the textual and spatial data. Further, duplicated measurements of sensor data make IoT stream data more cumbersome when integrated within a knowledge graph. GoT needs more hardware to store and process these duplicated and continuously growing streaming data, relevant spatial, and textual data. DESERT, a framework presented in this thesis, also provides an integrated view of heterogeneous data from disparate streaming IoT sources. However, to tackle rapidly growing IoT stream data DESERT proposes several techniques. Firstly, answering a user input query might not need all the data produced by the IoT devices, therefore, DESERT only retrieves and semantifies the data required to answer user input queries. Secondly, IoT stream data produced by the variety of IoT devices over time contain duplicated values, e.g., sensors have same measurements observed over several time stamps. Semantic description of IoT stream data with duplicated values increases the size of the integrated knowledge graph. DESERT proposes a factorization approach based on RDF molecules—set of RDF triples that share the same subject[27]—to reduce the number of duplicated RDF data in an integrated knowledge graph.

An approach for on-demand knowledge graph generation from heterogeneous data sources, in response to keyword-based queries, is presented by FuhSen [26]. FuhSen exploits REST APIs provided by the data sources to retrieve entity information corresponding to the input search query. Wrappers over several data sources run the search query over the data source and retrieve the information about the entity in form of RDF molecules, where an RDF molecule is a set of RDF triples describing the entity. FuhSen interlinks RDF molecules using RDF molecules based clustering approach to create a fused and universal representation of the entities whose information is retrieved from more than one data sources. A weighted bipartite graph of RDF molecules is created, where weights correspond to the semantic similarity between the RDF molecules. Based on the sum of the values of edges weights the maximum value is taken to identify the matched RDF molecules. Once the similar RDF molecules are identified, they are integrated into a knowledge graph. FuhSen generates an on-demand knowledge graph and resorts to RDF molecules based semantic similarity measures for semantic data integration. However, a large number of data sources might have duplicated information among RDF molecules, impacting thus the size of the knowledge graph. Similarly, DESERT creates on-demand knowledge graph from heterogeneous data (IoT) sources; nevertheless, DESERT exploits factorization techniques to reduce duplicated data in the knowledge graph. DESERT relies on transformation rules to generate factorized data from IoT stream data, producing all query answers.

3.3.2 Query Processing for Streaming Data

Large number of ubiquitous IoT devices, e.g., sensors, mobile phones, and location tracking devices, are generating stream data continuously. Processing of this rapidly and continuously arriving IoT stream data is crucial in order to extract useful information generated by these devices. Several RDF stream processing engines, C-SPARQL [15] and CQELS [81], have been developed to provide query execution over Linked Stream Data. C-SPARQL executes C-SPARQL queries [17] over streams of RDF data; it utilizes already existing Data Stream Management Systems (DSMS) [39] and a SPARQL-based reasoner. To effectively integrate DSMS and SPARQL technologies, C-SPARQL decomposes and transforms C-SPARQL queries into inputs corresponding to each of these technologies. C-SPARQL registers an RDF stream and C-SPARQL query to start stream data processing. The input C-SPARQL query is decomposed and translated into a dynamic and a static query, against a DSMS and SPARQL query engine, respectively. The static query retrieves the data from the SPARQL reasoner, while the dynamic query is registered and executed in the DSMS. The knowledge extracted from both systems is joined continuously within the streaming window to generate the results. CQELS provides a query execution framework for continuous RDF streaming data. CQELS utilizes data encoding techniques, where RDF elements are mapped to integers, to store more data in-memory. However, RDF stream data arriving at high rate might cause problems while encoding. Although C-SPARQL and CQELS engines provide RDF stream data processing, the rapidly growing amount of IoT stream data demands novel optimization techniques to process large amount of data within the streaming window. In this thesis we implement a continuous SPARQL query engine, DESERT, that implements on-demand semantification techniques, which reduce the amount of stream data being fed to the continuous query engine. The semantics of the data required to answer the input continuous query are described in the knowledge graph. More importantly, DESERT resorts to factorization techniques to minimize a knowledge graph size within the streaming window, by reducing the number of duplicated values in IoT stream data. Therefore, the continuous streaming engine receives relatively smaller knowledge graph size to execute an input continuous query.

3.4 Summary

Subject to the above-mentioned analysis of the existing approaches, this thesis focuses on techniques for the detection of frequent star patterns that are shared among several entities in knowledge graphs. These techniques should be able to efficiently identify frequent star patterns with highest instantiations. Furthermore, we want to exploit the frequent star patterns to generate efficient representations of knowledge graphs while the semantics encoded in the data are preserved. The representations of observational data and their meaning should be able to improve the data storage and processing over different data models and Big Data tools. More importantly, efficient semantic description techniques for the rapidly generated streaming observational data are required. These techniques should be able to answer continuous queries over data streams in an effective and efficient way.

Chapter 4

Compact Representations

Nowadays, there is a rapid increase in the number of observational data produced by a wide variety of IoT sensors and devices. Observational data contain observations that are characterized by certain properties and corresponding values. In real-world applications, several observations are redundant, i.e., they share the same values for a certain set of properties. Representations of such observations and their meaning provide actionable insights; however, these representations tremendously increase the size of data. Knowledge graphs have become a popular formalism for representing entities and their properties using a graph data model, e.g., the Resource Description Framework (RDF). An RDF graph comprises entities of the same type connected to objects or other entities using labeled edges annotated with properties. RDF graphs usually contain entities that share the same objects in a certain group of properties, i.e., they match star patterns composed of these properties and objects. In case the number of these entities or properties in these star patterns is large, the size of the RDF graph and query processing are negatively impacted; we refer these star patterns as *frequent star patterns*. In this chapter, we address the problem of identifying redundant observations by detecting *frequent star patterns* in RDF knowledge graphs describing historical data. Figure 4.1 shows the challenge we tackle in this chapter and the contribution to address the challenge. The content of this chapter is based on the publication [53] and the research work that is under review. The results of this chapter provide an answer to the following research question:

RQ1: What are the criteria to identify frequent star patterns?

To answer this research question, we devise the concept of *factorized RDF graphs*, which denote compact representations of RDF graphs where the number of frequent star patterns is minimized. We also develop computational methods to identify *frequent star patterns* and generate a *factorized RDF graph*, where *compact*

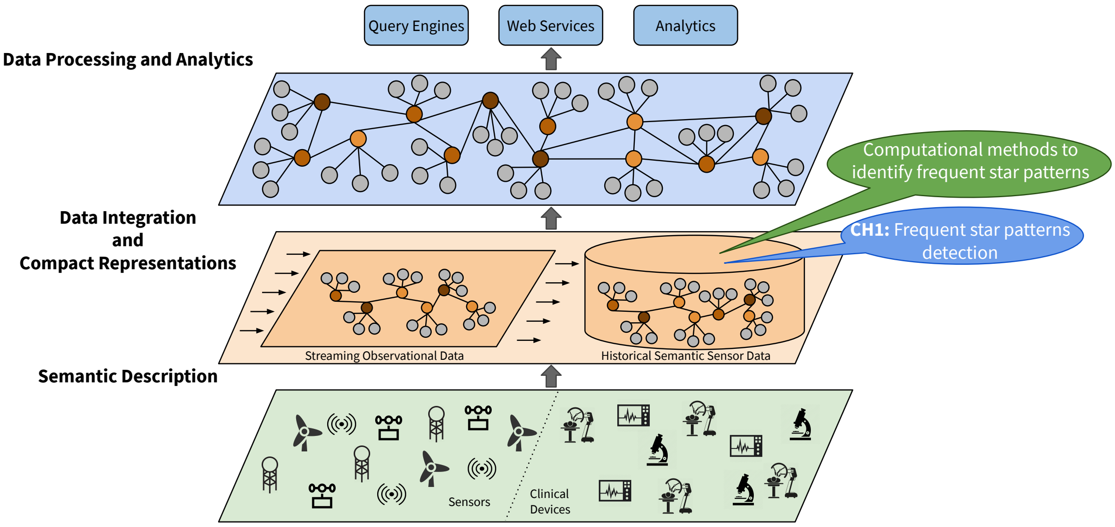

Query Engines
Web Services
Analytics
Data Processing and Analytics
Computational methods to identify frequent star patterns
Data Integration and Compact Representations
CH1: Frequent star patterns detection
Streaming Observational Data
Historical Semantic Sensor Data
Semantic Description
Sensors
Clinical Devices

Figure 4.1: **Challenges and Contributions.** This chapter addresses the problem of frequent star patterns detection in RDF knowledge graphs representing historical data, and provides computational methods to identify frequent star patterns.

RDF molecules replace frequent star patterns. A compact RDF molecule of a frequent star pattern denotes an RDF subgraph that instantiates the corresponding star pattern. Instead of having all the entities matching the original frequent star pattern, a *surrogate* entity is added and related to the properties of the frequent star pattern; it is linked to the entities that originally match the frequent star pattern. Since the edges between the entities and the objects in the frequent star pattern are replaced by edges between these entities and the surrogate entity of the compact RDF molecule, the size of the RDF graph is reduced. We devise computational methods for factorizing RDF graphs. The specific contributions of this chapter are as follows:

- Criteria for detecting frequent star patterns;
- Factorization techniques compacting frequent star patterns in RDF graphs. We have presented two algorithms: An exhaustive approach (named E.FSP) searches the space of frequent patterns produced by an algorithm like gSpan, to identify frequent star patterns. Further, G.FSP implements a Greedy meta-heuristics that is able to traverse the space of star patterns and identify the ones that are frequent. Star patterns are traversed in iterations, starting with the star patterns with the largest number of properties. The criteria of frequent star patterns correspond the stop criteria of the algorithm.
- An empirical study of both the frequent star patterns detection and factoriza-

> tion techniques using existing benchmarks. Experimental results show that both E.FSP and G.FSP identify frequent star patterns. Moreover, G.FSP overcomes E.FSP by reducing execution time in at least three orders of magnitude. More importantly, the experiments indicate that factorizing frequent star patterns by using surrogate keys enable for the creation of compact RDF graphs that reduce size while preserving the information in RDF graphs.

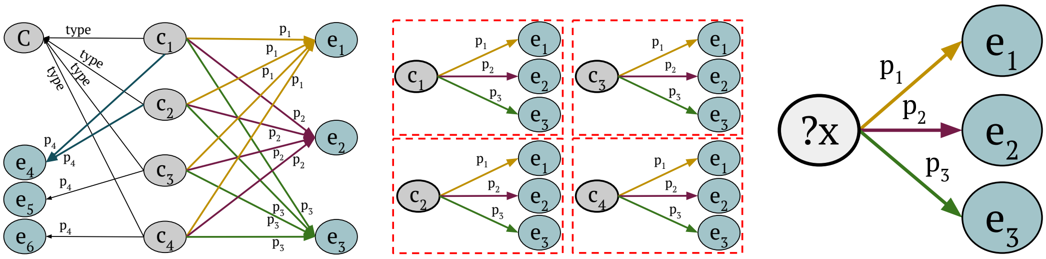
This chapter is structured as follows: We motivate our research in Section 4.1 by illustrating an RDF knowledge graph where entities of a class are associated with objects using certain properties of the class. Some of these properties and the relevant objects are redundantly shared by these entities, generating redundancies in the RDF graph. In Section 4.2 we present RDF graph factorization approach. To address the research question **RQ1**, we develop computational methods to identify frequent star patterns. Furthermore, we devise the concept of *factorized RDF graphs* which are compact representations of RDF graphs with a minimized number of frequent star patterns. Section 4.3 reports on the results of the experimental study. We use existing benchmarks to evaluate the efficiency and effectiveness of the proposed techniques. The results suggest that the proposed techniques for detecting frequent star patterns are able to effectively and efficiently identify frequent star patterns in RDF graphs. Moreover, the proposed factorization techniques are able to considerably reduce the size of an RDF graph while preserving all the information initially represented in the RDF graph. Finally, we present the closing remarks of this chapter in Section 4.4.

4.1 Motivating Example

We motivate the problem addressed in this chapter with an RDF graph where entities of the same type – or resources – match the same star pattern. In an RDF graph, matching the same star pattern means that the properties and objects are the same, whereas the entities are different. When the number of entities matching a star pattern is soaring, the size of the RDF graph increases and the query processing over the RDF graph is affected negatively. A star pattern with a high number of matching entities is a frequent star pattern. Figure 4.2a depicts an RDF graph composed by a class C, the entities c_1, c_2, c_3, c_4, e_1, e_2, e_3, e_4, e_5, and e_6, and the properties p_1, p_2, p_3, and p_4. A directed edge $(s\ p\ o)$ in the RDF graph stands for an RDF triple where p is a label that represents an RDF predicate, while s and o are subject and object nodes, respectively. Edges labeled with the predicate *type*[1], indicate that c_1, c_2, c_3 and c_4 are of the same type, i.e., the class C.

---

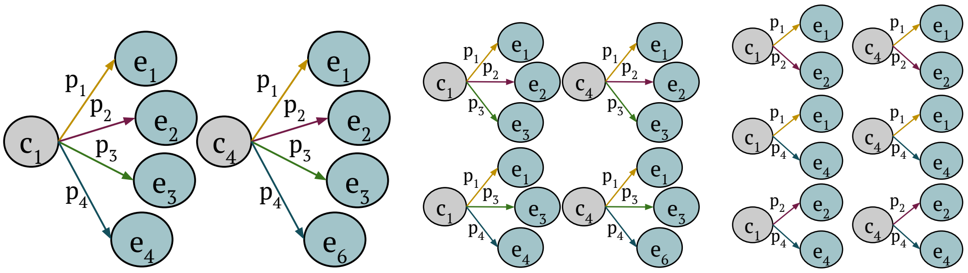
[1] property *type* refers to *rdf:type*



(a) An RDF Graph G (b) Entities in the Graph Pattern (c) A Star Pattern

Figure 4.2: **Motivating Example**. Frequent star pattern. (a) RDF graph with classes, entities, and properties; (b) Entities c_1, c_2, c_3, and c_4 are related to e_1, e_2, and e_3 with properties p_1, p_2, and p_3, respectively; (c) A star pattern with subject variable $?x$, respectively, relates e_1, e_2, and e_3 with properties p_1, p_2, and p_3.

The directed edge (c_1 p_1 e_1) expresses that the entity c_1 is related to object e_1 with the property p_1. Similarly, entities c_2, c_3, and c_4 are related to object e_1 with the property p_1, i.e., the indegree of e_1 is four. Similarly, entities c_1, c_2, c_3 and c_4 are related to e_2 and e_3 with the properties p_2 and p_3, respectively. Note that entities c_1, c_2, c_3, and c_4 are associated with the same objects, i.e., e_1, e_2 and e_3 through the edges annotated with same properties p_1, p_2, and p_3. Albeit sound, these redundant labeled edges generate frequent star patterns because entities of the same type are described using the same properties and objects. Figure 4.2b illustrates the RDF subgraphs that map to the same star pattern, shown in Figure 4.2c, extracted from the RDF graph in Figure 4.2a; note that *?x* is a variable whose instantiations correspond to constants in the RDF graph. In these RDF subgraphs, the properties p_1, p_2, and p_3, and the corresponding objects e_1, e_2, and e_3, respectively, are the same, whereas the entities c_1, c_2, c_3, and c_4 are different. This indicates that the star pattern is a frequent star pattern, i.e., several entities c_1, c_2, c_3, and c_4 instantiate the star pattern. Thus, several entities are related to the same objects, even not all the properties of the class are involved in frequent star patterns. A frequent star pattern comprising the entities c_1, c_2, c_3, and c_4 is illustrated in Figure 4.2c, where the node $?x$ represents the entities c_1, c_2, c_3, and c_4 of class C in the RDF graph in Figure 4.2a. gSpan [105] solves the problem of identifying the frequent subgraphs that involve same subject entities related to the same object values using a set of properties. However, the approach proposed in this chapter requires the identification of frequent star patterns, where each star pattern– with a subject variable–involves different subject entities related to the same object values using a set of properties. Figures 4.3a ,4.3b, and 4.3c show some of the subgraphs extracted by gSpan involving entities c_1 and c_4 , and the sets of properties containing four, three, and two properties, respectively,



(a) Subgraphs per 4 Properties (b) Subgraphs involving three Properties (c) Subgraphs involving two Properties

Figure 4.3: **Graph Patterns Identified by gSpan**. Subgraphs, involving entities c_1 and c_4, extracted by gSpan from the RDF graph in Figure 4.2a.(a) Subgraphs per set $\{p_1, p_2, p_3, p_4\}$ of properties; (b) Subgraphs over three properties from p_1, p_2, p_3, and p_4; (c) Subgraphs over two properties from p_1, p_2, p_3, and p_4.

from the RDF graphs in Figure 4.2a. gSpan exhaustively enumerates the frequent subgraphs; thus, finding frequent star patterns requires an exhaustive search over the generated frequent subgraphs. In this chapter, we exploit the RDF model and propose a technique that allows for transforming an RDF graph G into another RDF graph G' where the number of frequent star patterns is minimized. The graph G' includes all the nodes from G but additionally, G' comprises nodes that represent *factorized entities*– like the one in Figure 4.5c.

4.2 RDF Graph Factorization Approach

4.2.1 Problem Statement

Star patterns denote graph patterns covering RDF molecules:

Definition 4.2.1 (Star Pattern)**.** *Given is an RDF graph $G = (V, E, L)$, a class C in V and a set of properties $SP = \{p_1, p_2, \ldots, p_n\}$ such that C is the domain of all the properties in SP. Let entities $o_1, o_2, \ldots, o_n$ be the objects of the properties $p_1, p_2, \ldots, p_n$, respectively. Let $?s$ be a variable. A star pattern of C over the properties $p_1, p_2, \ldots, p_n$ and objects $o_1, o_2, \ldots, o_n$ corresponds to a graph pattern composed of the conjunction of triple patterns: $(?s\ p_1\ o_1)$,$(?s\ p_2\ o_2), \ldots, (?s\ p_n\ o_n)$.*

Example 4.2.1. *Figure 4.2c shows a star pattern composed of three triple patterns containing properties p_1, p_2, and p_3 and the corresponding objects e_1, e_2, and e_3, respectively. The entities c_1, c_2, c_3, and c_4 of class C in the RDF graph in Figure 4.2a match the star pattern. The variable $?x$ is the subject of the triple patterns referring to the entities matching the star pattern.*

Definition 4.2.2 (Class Multiplicity). *Given an RDF graph* $G = (V, E, L)$*, a class* C *in* E *and a set of properties* $SP = \{p_1, p_2, \ldots, p_n\}$ *such that* C *is the domain of all the properties in set* SP *of properties. Let entities* $o_1, o_2, \ldots, o_n$ *be objects of the properties* $p_1, p_2, \ldots, p_n$*, respectively. The multiplicity of* o_1*,* o_2*,...,* o_n *in* G*,* $M(o_1, o_2, \ldots, o_n|G)$ *is defined as the number of different entities in* C *that match a star pattern having the same objects* $o_1, o_2, \ldots, o_n$ *in the properties* $p_1, p_2, \ldots, p_n$*. Entities* s *correspond to instantiations of the subject variable in the star pattern.*

$$M(o_1, o_2, \ldots, o_n|G) = |\{s| \, (s \ \mathtt{:type}\ C) \in G, (s\ p_1\ o_1) \in G, (s\ p_2\ o_2) \in G, \ldots, \\ (s\ p_n\ o_n) \in G\}|$$

Example 4.2.2. *In the RDF graph in Figure 4.2a, the multiplicity of the objects* e_1*,* e_2 *and* e_3*, given the set* $\{p_1, p_2, p_3\}$ *of properties, is* 4*, because there are four instantiations of the subject variable. Similarly, the multiplicity of objects* e_4*,* e_5 *and* e_6*, in the set* $\{p_4\}$ *of properties is* 1 *and* 2*.*

Definition 4.2.3 (Class Multiplicity Inverse). *Given class* C*, a set* $SP = \{p_1, p_2, \ldots, p_n\}$ *of properties and corresponding objects* o_1*,* o_2*,...,* o_n*, the multiplicity inverse of* o_1*,* o_2*,...,* o_n *in* G*, denoted* $MI(o_1, o_2, \ldots, o_n|G)$*, is:*

$$MI(o_1, o_2, \ldots, o_n|G) = 1/M(o_1, o_2, \ldots, o_n|G)$$

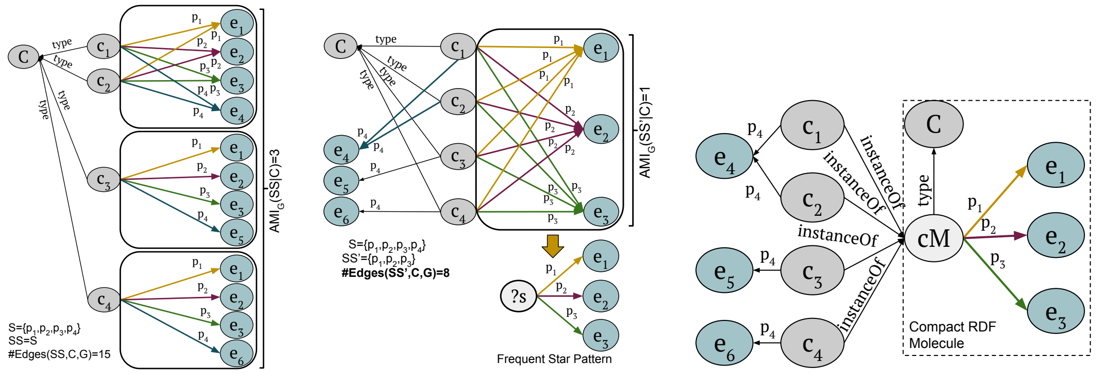
Example 4.2.3. *In the RDF graph in Figure 4.2a, the class multiplicity inverse of the objects* e_1*,* e_2*, and* e_3*, given the set* $\{p_1, p_2, p_3\}$ *of properties, is* $\frac{1}{4}$*. The multiplicity inverse of objects* e_4*,* e_5*, and* e_6 *in the set* $\{p_4\}$ *is* $\frac{1}{1}$ *and* $\frac{1}{2}$*.*

Definition 4.2.4 (Multiplicity of Star Patterns). *Given a class* C *in an RDF graph* G *with properties* $SP = \{p_1, p_2 \ldots, p_n\}$*. The multiplicity of the star patterns in* C *over* SP*,* $AMI_G(p_1, p_2, \ldots, p_n|C)$*, is defined as follows:*

$$AMI_G(p_1, p_2, \ldots, p_n|C) = \lceil f'_{\forall s \in C}(\{MI(o_1, o_2, \ldots, o_n|G)|(s\ \mathtt{type}\ C) \in G, \\ (s\ p_1\ o_1) \in G, (s\ p_2\ o_2) \in G, \ldots, (s\ p_n\ o_n) \in G\})\rceil$$

where $f'(.)$ *is an aggregation (e.g., summation) function.*

Example 4.2.4. *In the RDF graph in Figure 4.2a, the multiplicity of the star patterns of* C *over the set* $\{p_1, p_2, p_3\}$ *of properties is* $\frac{1}{4} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4} = 1$*, which is obtained by summing up the class multiplicity inverse of the objects* e_1*,* e_2*, and* e_3 *given the set* $\{p_1, p_2, p_3\}$ *of properties, for each entity* c_1*,* c_2*,* c_3*, and* c_4 *of class* C *matching the star pattern. Similarly, the multiplicity of the star patterns of class* C *over the set* $\{p_4\}$ *is* $\frac{1}{2} + \frac{1}{2} + \frac{1}{1} + \frac{1}{1} = 3$ *in the RDF graph, and is obtained by summing up the individual class multiplicity inverse of objects* e_4*,* e_5*, and* e_6 *given the set* $\{p_4\}$*, for each of the entities* c_1*,* c_2*,* c_3*, and* c_4 *of class* C *that map the corresponding star patterns. The multiplicity of the star patterns over a set of properties corresponds to the number of star patterns composed of the set of properties and the corresponding objects.*



(a) $\#Edges(SP, C, G)$ over p_1,p_2,p_3, and p_4

(b) $\#Edges(SP, C, G)$ over properties p_1,p_2, and p_3

(c) **Factorized Graph** G' from G

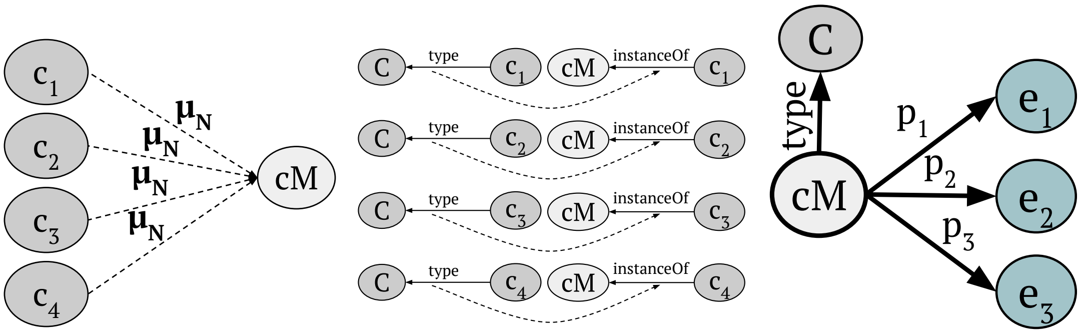
Figure 4.4: **The Frequent Star Patterns Detection Problem**. Properties involved in frequent star patterns. (a) Star patterns over the set $SS = \{p_1, p_2, p_3, p_4\}$ of properties in class C require three surrogate entities and $\#Edges(SS, C, G)$ are 15; (b) Star patterns over the set $SS' = \{p_1, p_2, p_3\}$ of properties in class C require one surrogate entitiy and $\#Edges(SS', C, G)$ are eight; (c) A factorized RDF graph G' of G composed of compact RDF molecule with a surrogate entity cM.

The problem of frequent star patterns detection is defined next, the solutions correspond to frequent star patterns. We define the frequent star patterns detection problem as the minimization of connections between a class instances and values linked through the properties. To find the minimum number of edges over the properties in a class, the sum of the number of edges in the star patterns over a set of properties and the number of edges between the class entities and the properties that are not involved in the star patterns is computed.

Definition 4.2.5 (FSP Detection Problem)**.** *Given an RDF graph $G = (V, E, L)$ and a class C in G with set of properties S and number of instances $AM_G(C)$. The problem of Frequent Star Patterns Detection (FSP Detection) is to find a subset SP of S such that the star patterns SGP of C over SP corresponds to* frequent star patterns*, i.e., $\#Edges(SP, C, G)$ is minimized:*

$$\underset{SP \subseteq S}{\arg\min} \left\{ \underbrace{AMI_G(SP|C) * (|SP| + 1) + AM_G(C) * (|S - SP|)}_{\#Edges(SP,C,G)} \right\} \tag{1}$$

Example 4.2.5. *Figure 4.4 depicts the problem of detecting frequent star patterns from the RDF graph in Figure 4.2a. Figure 4.4a presents three star patterns $AMI_G(SS|C)$ over p_1, p_2, p_3, and p_4, and 15 edges in $\#Edges(SS, C, G)$. However, only one star pattern $AMI_G(SS'|C)$ over the properties p_1, p_2, and p_3 exists*



(a) $\mu_\mathbf{N}$ from G into G' (b) *type* G into *instanceOf* G' (c) A Compact RDF Molecule

Figure 4.5: **The RDF Graph Factorization Problem**. Factorization of RDF graph G into G'. (a) Entity mappings μ_N from the RDF graph G in 4.2a to the surrogate entity cM in G'; (b) Transformation of property *type* from G to G'; (c) A compact RDF molecule for the frequent star pattern over p_1, p_2, and p_3.

in Figure 4.4b. A small value of $\#Edges(SS', C, G)$ i.e., eight, shows a subgraph over SS' that is represented by only one star pattern with more instantiations than the star patterns for SS, i.e., it is a frequent star pattern. Thus, the set SP of properties where $\#Edges(SP, C, G)$ is minimal, encloses a subgraph with the minimal number of star patterns which have the maximal number of instantiations; additionally, these star patterns are the ones with the greater number of properties. Figure 4.4c shows the factorized RDF graph where this frequent star pattern has been replaced with a compact RDF molecule on a surrogate entity cM; this factorization reduces the size of the original RDF graph.

Theorem 4.2.1. *Given an RDF graph G, a class C in G, and non-empty sets of properties S, SP, and SP' of C such that $SP' \subset SP \subset S$. If $\#Edges(SP', C, G) > \#Edges(SP, C, G)$, then $\forall SP'' \subset SP'$, $\#Edges(SP'', C, G) \geq \#Edges(SP, C, G)$.*

Proof. By contradiction. Suppose $\#Edges(SP'', C, G) < \#Edges(SP, C, G)$. From $\#Edges(SP', C, G) > \#Edges(SP, C, G)$ and $SP' \subset SP \subset S$, it can be inferred that $AMI_G(SP|C) < AM_G(C)$, $AMI_G(SP'|C) < AM_G(C)$, $|SP''| < |SP'| < |SP| < |S|$, $|SP - SP''| \geq 2$, and $AMI_G(SP''|C) < AM_G(C)$. Considering these inequalities in $\#Edges(SP'', C, G)$ and $\#Edges(SP, C, G)$, we can demonstrate that $\#Edges(SP'', C, G)$ is at least greater than $\#Edges(SP, C, G)$ in $2 * AM_G(C)$, contradicting, thus, $\#Edges(SP'', C, G) < \#Edges(SP, C, G)$. □

Definition 4.2.6 (A Compact RDF Molecule). *Given a star pattern SGP of a class C over the properties $p_1, p_2, \ldots, p_n$ and objects $o_1, o_2, \ldots, o_n$. Given a surrogate entity sg of type C. A compact RDF molecule for SGP is an RDF molecule composed of RDF triples $(sg\ p_1\ o_1)$,$(sg\ p_2\ o_2)$,…,$(sg\ p_n\ o_n)$.*

Example 4.2.6. *Figure 4.5c shows a compact RDF molecule that instantiates the star pattern presented in Figure 4.2c, which is composed of the properties* p_1*,* p_2*, and* p_3 *and the corresponding objects* e_1*,* e_2*, and* e_3*, respectively. The surrogate entity* cM *in the compact RDF molecule, represents the entities* c_1*,* c_2*,* c_3*, and* c_4 *of type* C *matching the star pattern, as shown in Figure 4.2b.*

Definition 4.2.7 (The RDF-F Problem)**.** *Given an RDF graph* $G = (V, E, L)$ *and a set of properties* SP*, the problem of RDF factorization (RDF-F) corresponds to finding a factorized RDF graph of* G*,* $G' = (V', E', L')$*, where the following hold:*

- *Entities in* G *are preserved in* G'*, i.e.,* $V \subseteq V'$*.*
- *For each entity* s_i *in* V *that corresponds to an instantiation of the variable of a frequent star pattern* SGP *of a class* C *over the set* SP *in* G*, there is an entity* s_{SGP} *in* V' *that corresponds to the surrogate entity of the compact RDF molecule of* SGP*. Formally, there is a partial mapping* μ_N*:* $V \rightarrow V'$*:*
  - *Instances of the frequent star pattern* SGP *are mapped to the surrogate entity of the star pattern, i.e.,* $\mu_N(s_i){=}s_{SGP}$*.*
  - *The mapping* μ_N *is not defined for the rest of the entities that do not instantiate a frequent star pattern in* G*.*
- *For each RDF triple t in (s p o) in* E*:*
  - *If* $\mu_N(s)$ *is defined and* C_s *is the type of* s*, and* p *is type, then the triples (s instanceOf* $\mu_N(s)$*), (*$\mu_N(s)$ *type* C_s*) belong to* E'*.*
  - *If* $\mu_N(s)$ *is defined and* C_s *is the type of* s*, and* $p \in SP$*, then the triples (*$\mu_N(s)$ *p o) belong to* E'*.*
  - *Otherwise, the RDF triple t is preserved in* E'*.*

Example 4.2.7. *Consider RDF graphs* G *and* G' *shown in Figures 4.2a and 4.4c, respectively. Figure 4.5a depicts a map* μ_N *that assigns entities* c_1*,* c_2*,* c_3*, and* c_4 *of class* C *in* G *to the surrogate entity* cM *in* G'*. Further, entities* c_1*,* c_2*,* c_3*,* c_4*,* C*,* e_1*,* e_2*,* e_3*,* e_4*,* e_5*, and* e_6 *are preserved in* G'*. Moreover, the edge labeled with property* p_1 *in* G*, i.e., (*c_1 p_1 e_1*) is presented with edges (*c_1 $instanceOf$ cM*), (*cM p_1 e_1*) and (*cM *type* C*) in* G'*; similarly, edges labeled with properties* p_2 *and* p_3 *in* G *are represented in* G'*. Figure 4.5b shows the transformations of the connections between entities* c_1*,* c_2*,* c_3*, and* c_4 *and the class* C *using labeled edges annotated with property type, with the connections relating the entities* c_1*,* c_2*,* c_3*, and* c_4 *to the corresponding surrogate entity* cM *using the property* $instanceOf$*.*

Definition 4.2.8 (Axioms for InstanceOf)**.** *The property* instanceOf *is a functional property defined as follows:*

- *If (s_i instanceOf sg) and (sg type C) then (s_i type C).*
- *If (s_i instanceOf sg) and (sg p_j o_k) then (s_i p_j o_k).*

These two axioms enable to represent implicitly, all the knowledge encoded in the edges from an original RDF graph that are removed during the factorization process. They are utilized during query processing to rewrite queries over the original RDF graph into queries against the factorized RDF graph.

4.2.2 FSP Detection Approach

Algorithm 1 E.FSP Algorithm

```
Input: A dictionary subgraphsDict of subgraphs over the subsets of properties in
    $S$, A set $S$ of properties of class $C$.
Output: Frequent star patterns fsp, A set $SP$ of properties
fsp ← [], $SP$ ← ∅, minEdges ← 0, subsetCard ← $|S|$
while subsetCard ≥ 2 do
    propSets ← getSubsetsOf($S$, subsetCard)
    for $SP$ ∈ propSets do
        subgraphs ← subgraphsDict[$SP$]
        totalEdges ← countEdges(subgraphs)
        if minEdges == 0 then
            minEdges ← totalEdges
            fsp ← subgraphs
            bestSP ← $SP$
        else if totalEdges < minEdges then
            minEdges ← totalEdges
            fsp ← subgraphs
            bestSP ← $SP$
        end if
    end for
    subsetCard ← subsetCard − 1
end while
$SP$ ← bestSP
return fsp, $SP$
```

To solve the *FSP detection* problem, we propose two algorithms that perform iterations over frequent patterns involving different sets of properties sets of a class C in an RDF graph G, and the class entities. *E.FSP*, presented in Algorithm 1, resorts to a frequent pattern mining algorithm like gSpan. *E.FSP* exploits breadth first search technique to exhaustively traverse the search space of frequent patterns generated by the frequent pattern mining algorithm, and always finds the best frequent star patterns. Figure 4.6a illustrates the iterations performed by *E.FSP* to find the frequent star patterns in the RDF graph in Figure 4.2a. *E.FSP* receives a dictionary *subgraphsDict* of all the subgraphs over the subsets of the set S of properties in the class C in an RDF graph G. The keys of the dictionary *subgraphsDict* are the combination of properties in the subsets of S, and the dictionary values are the subgraphs involving the properties from the corresponding keys. *E.FSP* generates frequent star patterns and a set of properties involved in the frequent star patterns. *E.FSP* initializes the variables *fsp*, SP, *minEdges*, and *subsetCard* in line 1. The variables *minEdges* and *subsetCard* are initialized with values 0 and cardinality of S, respectively. From lines 2-18, *E.FSP* iterates over all the subgraphs involving two or more properties to find the frequent star patterns. In Figure 4.6a, *E.FSP* starts iterations with the set of properties $SP = \{p_1, p_2, p_3, p_4\}$, and the subgraphs involving the properties in subsets of SP, where the cardinality of subsets is equal to the cardinality of S, i.e., four (line 3). The generated subset contains all the properties in SP, i.e., $\{p_1, p_2, p_3, p_4\}$, and the total number of edges *totalEdges* in SP is computed, i.e., 16 (line 5-6). Since *minEdges* are 0, therefore, the value 16 of *totalEdges* is assigned to *minEdges*, subgraphs over $SP = \{p_1, p_2, p_3, p_4\}$ and SP are assigned to fsp and *bestSP*, respectively (line 7-10). At line 17, the subset size *subsetSize* is reduced by one in order to generate the subsets of properties of S with the cardinality one less the cardinality of S, i.e., three. The subsets $\{p_1, p_2, p_4\}$, $\{p_1, p_3, p_4\}$, and $\{p_2, p_3, p_4\}$, of cardinality three, generate more number of edges, i.e., value of *totalEdges* is 17, than the minimum number of edges *minEdges*, i.e., 16, and are not selected as the best sets of properties. However, the subgraphs over the subset $\{p_1, p_2, p_3\}$ contain 11 number of triples, which is less than 16 the value of *minEdges*. Therefore, *E.FSP* selects $\{p_1, p_2, p_3\}$ as the best set of properties and the corresponding subgraphs as the frequent star patterns (line 11-15). Once all the subsets SP of S with cardinality three, are evaluated, the value of *subsetCard* is reduced by one, i.e., two, and the subsets of cardinality two are evaluated in the next iteration. Figure 4.6a presents that all the subsets of cardinality two generate larger values, i.e., 14 and 18, for *totalEdges* than the value 11 for *minEdges*. Therefore, none of the subsets of properties of cardinality two contains the frequent star patterns. Further, all the subsets of cardinality greater or equal to two have been evaluated, *E.FSP* stops and returns $\{p_1, p_2, p_3\}$ as the best set of properties and the corresponding subgraphs as the frequent star patterns (line 19-20).

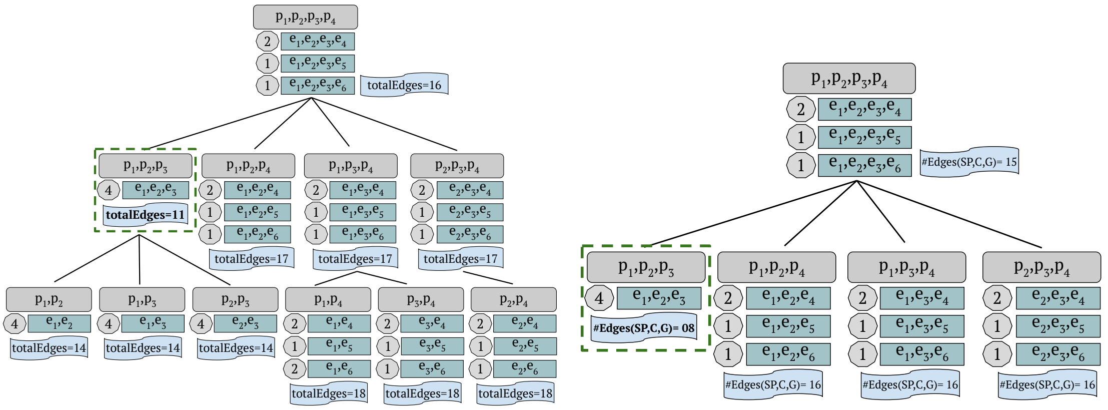
Algorithm 2 G.FSP Algorithm

Input: A set S of properties of class C in G, and a list *starList* of star patterns over properties in S.

Output: Frequent star patterns *fsp*, A set SP of properties.

```
$fsp \leftarrow []$, $starList' \leftarrow []$, $SP \leftarrow S$, $SP' \leftarrow \emptyset$, $fValue \leftarrow fValue' \leftarrow 0$
repeat
  if $|SP| \geq 2$ then
    if $AMI_G(SP|C) == 1$ then
      $fsp \leftarrow starList$
      return $fsp$, $SP$
    else
      $fValue \leftarrow \#Edges(SP, C, G)$
      for $p \in SP$ do
        $SP' \leftarrow SP - \{p\}$
        if $|SP'| \geq 2$ then
          Create $starList'$ over $SP'$ using $starList$
          $value \leftarrow \#Edges(SP', C, G)$
          if $AMI_G(SP'|C) == 1$ then
            $fValue' \leftarrow value$
            $bestSP \leftarrow SP'$
            $bestSList \leftarrow starList'$
            break
          else if $value < fValue'$ then
            $fValue' \leftarrow value$
            $bestSP \leftarrow SP'$
            $bestSList \leftarrow starList'$
          end if
        end if
      end for
    end if
  end if
  $starList \leftarrow bestSList$, $SP \leftarrow bestSP$
until $fValue' > fValue$
$fsp \leftarrow starList$
return $fsp$, $SP$
```

G.FSP, presented in Algorithm 2, adopts a greedy approach to traverse the search space without generating all the frequent patterns. *G.FSP* starts iterations using a set SP of properties containing all the properties in S of a class C in



(a) Exhaustive FSP Approach (E.FSP) (b) Greedy FSP Approach (G.FSP)

Figure 4.6: **Frequent Star Patterns Detection**. *E.FSP* and *G.FSP* iterate over the star patterns in the RDF graph in Figure 4.2a to detect the frequent star patterns. (a) *E.FSP* exhaustively iterates the whole search space of frequent patterns; (c) *G.FSP* iterates the search space without generating all the star patterns.

an RDF graph G. *G.FSP* computes the Formula 1 value for SP and iterates over the subsets SP' of cardinality one less the cardinality of SP and computes Formula 1 for each of subsets SP'. A set SP' with a smaller formula value than the formula value of SP, is selected as the best set of properties in that iteration, and is used in the next iteration to check the subsets of cardinality one less the cardinality of the selected set. The iterations are performed until the cardinality of the selected subset of properties is less than two. Based on the property presented in Theorem 4.2.1, *G.FSP* stops, if none of the subsets SP' generates less value for formula than the formula value of SP. In addition, *G.FSP* stops whenever the cardinality of the set of properties is less than two, or the multiplicity of star patterns $AMI_G(SP|C)$ is one. *G.FSP* receives a set S of properties in class C in an RDF graph G, and a list $starList$ of star patterns involving properties in S. *G.FSP* returns frequent star patterns *fsp* and a set of properties SP involved in the frequent star patterns. Figure 4.6b shows the iterations performed by *G.FSP* to detect the frequent star patterns in the RDF graph in Figure 4.2a. *G.FSP* initializes all the variables at line 1, where SP is assigned the set S of properties for the first iteration, i.e., $SP = \{p_1, p_2, p_3, p_4\}$. In lines 2-29, *G.FSP* iterates over the subsets of SP to find the frequent star patterns based on the criteria in Formula 1. The cardinality value four of SP is greater than two (line 3), and $AMI_G(SP|C)$ is not equal to one (line 4-7), therefore, *G.FSP* computes the value of $\#Edges(SP, C, G)$ of SP, i.e., 15 (line 8). In lines 9-25, *G.FSP* iterates over the subsets of SP of cardinality one less the cardinality of SP to find the best set of properties for the next iteration. At line 10, a property p is removed from

SP to generate a subset SP', e.g., by removing p_1 a subset $SP' = \{p_2, p_3, p_4\}$ is generated. Since the cardinality of SP' is more than two, therefore, a star list $starList'$, representing the star patterns over SP', is created using $starList$ (line 12-13). The value of $\#Edges(SP', C, G)$ for SP' is computed, i.e., 16 (line 13). For SP', $AMI_G(SP'|C)$ is not one, and the value 16 of $\#Edges(SP', C, G)$ for SP' is not less than the value 15 of $\#Edges(SP, C, G)$ for SP, therefore, the star patterns over $SP' = \{p_2, p_3, p_4\}$ do not involve frequent star patterns and SP' is not a best candidate for the next iteration. Similarly, the subsets $\{p_1, p_3, p_4\}$ and $\{p_1, p_2, p_4\}$, of SP by removing p_2 and p_3, respectively, give a higher value 16 for $\#Edges(SP', C, G)$ and the star patterns over these set of properties are not better than the star patterns over SP. However, $SP' = \{p_1, p_2, p_3\}$, generated from SP by removing p_4, gives one star pattern, therefore, the star pattern involving properties in SP' is returned as the frequent star pattern without performing more iteration (line 14-18). In case, the set SP' is involved in more than star patterns and the formula value of SP' smaller than the value of SP, then SP' is selected for the next iteration (line 19-23). *G.FSP* stops and no further iterations are performed if none of the subsets SP' of SP generates a smaller value for $\#Edges(SP', C, G)$ than $\#Edges(SP, C, G)$. *G.FSP* returns the star patterns involving SP, with a minimum value for $\#Edges(SP, C, G)$, as the frequent star patterns, and SP as the best set of properties. *E.FSP* and *G.FSP* work under the following assumptions: (a) all RDF molecules are complete, i.e., all class entities have values for all the properties, (b) all the properties are functional. In addition to these assumptions, *G.FSP* has one more assumption: (c) if there are ties while deciding between the sets of properties, only one will be selected. Complexity of *E.FSP* is exponential, i.e., 2^n. *G.FSP* adopts a Greedy approach and prunes the search space by selecting only the best set of properties during each iteration until the stop condition is met, i.e., no better set of properties with a minimum formula value can be found. In the worst case, the computational complexity of *G.FSP* is $\sum_{i=0}^{n}(n-i)=\frac{n(n+1)}{2}$, where n is the cardinality of the input set of properties. The complexity of *G.FSP* grows linearly with the increase in the size of the input set of properties.

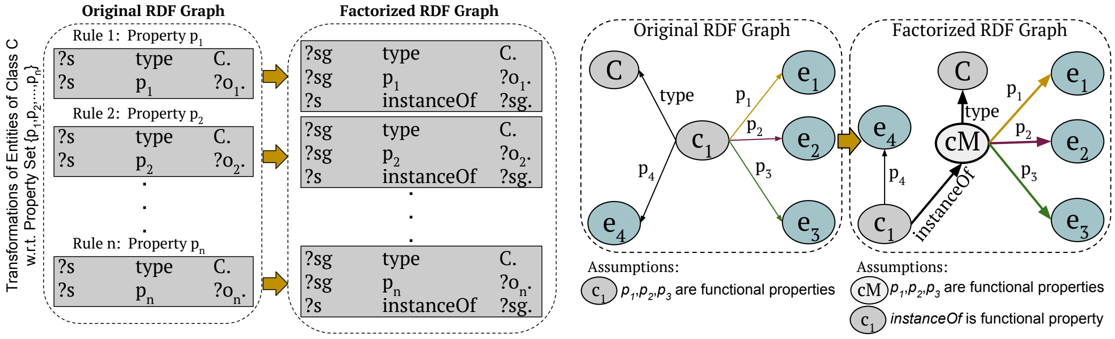
4.2.3 A Factorization Approach

Algorithm 3 The Factorization Algorithm

```
Input: An RDF graph $G(V, E, L)$, A class $C$, A set $SP$ of properties from E.FSP
    Algorithm 1 or G.FSP Algorithm 2
Output: Factorized RDF Graph $G'(V', E', L')$ and entity mappings $\mu_N$
1: $\mu_N \longleftarrow \emptyset, V' \longleftarrow \emptyset, E' \longleftarrow \emptyset, L' \longleftarrow \emptyset$
2: for all $o_1, o_2, \dots, o_n \in V such\ that\ SS = \{s | p_1, p_2, \dots, p_n \in SP AND$
   $(s$ type C$) \in G, (s\ p_1\ o_1) \in G, (s\ p_2\ o_2) \in G \dots, (s\ p_n\ o_n) \in G\}$ do
```

```
3:     $sg \leftarrow SurrogateEntity()$
4:     for $ss \in SS$ do
5:         $\mu_N \leftarrow \mu_N \cup \{(ss, sg)\}$
6:     end for
7: end for
8: for $(s\ p\ o) \in E \wedge s, o \in V$ do
9:     if $\mu_N(s) \neq \emptyset$ then
10:        {Create compact RDF molecule}
11:        if $p == type$ then
12:            $E' \leftarrow E' \cup \{(s\ instanceOf\ \mu_N(s)),\ (\mu_N(s)\ p\ o)\}$
13:            $V' \leftarrow V' \cup \{s, \mu_N(s), o\}$
14:            $L' \leftarrow L' \cup \{p, instanceOf\}$
15:        else if $p \in SP$ then
16:            $E' \leftarrow E' \cup \{(\mu_N(s)\ p\ o)\}$
17:            $V' \leftarrow V' \cup \{\mu_N(s), o\}$
18:            $L' \leftarrow L' \cup \{p\}$
19:        else
20:            $E' \leftarrow E' \cup \{(s\ p\ o)\}$
21:            $V' \leftarrow V' \cup \{s, o\}$
22:            $L' \leftarrow L' \cup \{p\}$
23:        end if
24:    else
25:        $E' \leftarrow E' \cup \{(s\ p\ o)\}$
26:        $V' \leftarrow V' \cup \{s, o\}$
27:        $L' \leftarrow L' \cup \{p\}$
28:    end if
29: end for
30: return $G'(V', E', L'), \mu_N$
```

We present a solution to the problem of factorizing RDF graphs describing data using ontologies. A sketch of the proposed method is presented in Algorithm 3. The algorithm receives an RDF graph $G = (V, E, L)$, a class C, and a set SP' of properties from *E.FSP* or *G.FSP*, and generates a factorized RDF graph $G' = (V', E', L')$, and the entity mappings μ_N from the entities of class C in V in RDF graph G to the surrogate entities in V' in RDF graph G'. The algorithm initializes the set of mappings μ_N, the set of nodes V', the set of labeled edges E' and the set of edge labels (properties) L' of the factorized RDF graph G' (line 1). For all the entities of C related to the same objects $o_1, o_2, \ldots, o_n$ using edges annotated with properties $p_1, p_2, \ldots, p_n$ in SP', the algorithm creates a surrogate entity sg



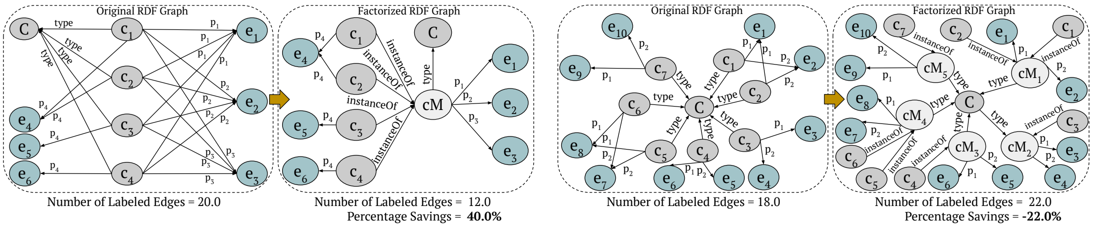
(a) Transformation Rules for Class C

(b) Original and Factorized RDF Graphs

Figure 4.7: **Transformations in RDF Graph**. Transformation rules preserved between original and factorized graphs. (a) Transformation rules over the properties $p_1, p_2, \ldots, p_n$; (b) Portions of RDF graphs (original and factorized). Nodes and edges added to create the factorized RDF graph, are highlighted in bold.

for the corresponding compact RDF molecule in G' (lines 2-3). In lines 4-6, the algorithm maps all the entities, that are related to $o_1, o_2, \ldots, o_n$ using properties $p_1, p_2, \ldots, p_n$ in G, to the surrogate entity in μ_N. Once all the mappings of the entities of C in G to the corresponding surrogate entities in G' are in μ_N, the factorized RDF graph G' is created using μ_N (lines 8-29). For each RDF triple $(s\ p\ o)$ in E, if entity mapping $\mu_N(s)$ is defined, then a compact RDF molecule is created. If p is *type*, then the new edges (s *instance* $\mu_N(s)$), and $(\mu_N(s)\ p\ o)$ are added to G' along with entities s, o and the mapped surrogate entity of s, and the edge labels p and $instanceOf$ (lines 11-14). If p is in SP, the new edge $(\mu_N(s)\ p\ o)$, and entities s, o and the edge label p are added to G' (lines 15-18). If entity mapping $\mu_N(s)$ are defined, however, p is not in SP, or p is not *type*, then the edge $(s\ p\ o)$ is added to G' along with the corresponding nodes and the edge label (lines 19-23). If entity mapping $\mu_N(s)$ is not defined, then the edge $(s\ p\ o)$ and the nodes s and o, and edge label p are added to G' (lines 24-28).

Figure 4.7 depicts the transformations for the set $\{p_1, p_2, \ldots, p_n\}$ of properties performed in an RDF graph whenever a corresponding factorized RDF graph is created. Figure 4.7a presents transformation rules; one rule for each property in $\{p_1, p_2, \ldots, p_n\}$ of class C. Each rule states how the labeled edges associated with a C in an original RDF graph are transformed into the edges in the factorized graph. Rule 1 asserts that the relation between an entity s of C with an object o_1 is not explicitly represented by one property in the factorized RDF graph. In order to retrieve o_1, a path across the labeled edges between entities s and the corresponding surrogate entity sg have to be traversed. Similarly, the rest of the transformation rules establish how explicit associations between entities of C and


Original RDF Graph
Number of Labeled Edges = 20.0
Factorized RDF Graph
Number of Labeled Edges = 12.0
Percentage Savings = 40.0%
Original RDF Graph
Number of Labeled Edges = 18.0
Factorized RDF Graph
Number of Labeled Edges = 22.0
Percentage Savings = -22.0%

(a) %age Decrease in Edges after Factorization (b) %age Increase in Edges after Factorization

Figure 4.8: **RDF Graph Factorization Overhead**. Factorization of RDF graphs is not worthy in all cases. (a) Entities of class C in the original RDF graph match the frequent star pattern over the properties p_1, p_2, and p_3; (b) few entities match each star pattern over p_1 and p_2 causing factorization overhead.

the objects using properties $p_2, \ldots, p_n$ in the original RDF graphs are represented by the path of labeled edges annotated with properties in the factorized RDF graphs. Algorithm 3 adds the corresponding labeled edges of these paths in lines 7-16. Furthermore, Figure 4.7b presents a portion of the RDF graph in Figure 4.2a and corresponding transformation in the factorized RDF graph in Figure 4.4c. The surrogate entity and the new labeled edges are highlighted in bold in the factorized RDF graph. The Algorithm 3 creates the surrogate entity in line 4; new edges are created in line 10. Additionally, assumptions about the characteristics of the entity associations in the graph are presented. Some edges existing between the entities in RDF graph in Figure 4.2a are not present in the factorized RDF graph in Figure 4.4c, these entity associations can be obtained by traversing the factorized RDF graph as indicated by the corresponding transformation rules in Figure 4.7a.

Figure 4.8 illustrates an example of factorization overhead, i.e., a case when it is not worthy to factorize a class given a set of properties in an RDF graph. Figure 4.8a presents an example where savings are observed in the number of edges after factorization. The factorization of RDF graph in Figure 4.8a for the class C using the properties p_1, p_2, and p_3, reduces the number of edges from 20.0 to 12.0 and the positive value 40.0% for percentage savings indicates a percentage decrease in the number of edges. Furthermore, the edge savings gained after factorization are high enough to compensate the addition of the surrogate entity cM in the factorized RDF graph. In contrast, factorization of the RDF graph over the class C using the properties p_1 and p_2 introduces an overhead, as shown in Figure 4.8b, by increasing the number of nodes and edges in the factorized RDF graph. The number of edges is increased from 18.0 to 22.0, shown in Figure 4.8b, after factorization and a negative value -22.0% for the percentage savings indicates an increase in the number of edges. The star patterns, detected in the original RDF graph, in Figure 4.8b, are replaced by the corresponding compact RDF molecules with the corresponding surrogate entity and new labeled edges (presented in Algorithm 3). Due to the high number of star patterns, the addition of the surrogate entities and new labeled edges increases the size of the factorized RDF graph.

4.3 Experimental Study

We study the effectiveness and efficiency of the proposed techniques for detecting frequent star patterns. Moreover, given a class, we evaluate the impact of the factorization techniques on the RDF graphs size by selecting several combinations of the properties in the class. We empirically assessed the following research questions: **ResearchQ1)** Are the proposed frequent star patterns detection techniques able to efficiently detect the frequent star patterns in RDF graphs? **ResearchQ2)** Are the proposed frequent star patterns detection techniques able to detect the frequent star patterns in RDF graphs? **ResearchQ3)** What is the impact of different combinations of properties of a class on the size of factorized RDF graphs? **ResearchQ4)** Are the proposed factorization techniques able to reduce the number of labeled edges in RDF graphs? Our experimental configuration is as follows:
Datasets. Evaluation is conducted on three *LinkedSensorData* datasets [77] semantically described using the Semantic Sensor Network (SSN) Ontology. These

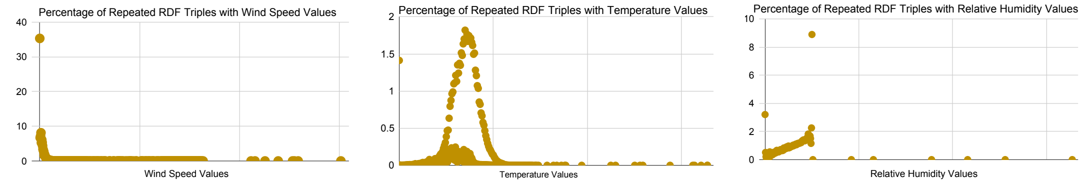
Table 4.1: **Datasets**. (a) Statistics of the datasets with observations about several weather phenomena, collected from around 20,000 weather stations in the United States; (b) The number of labeled edges *NLE(G)*, in the datasets obtained after gradually integrating the RDF datasets D1, D2, and D3 describing observations.

(a) Statistics of datasets collected from around 20,000 weather stations in the US.

Dataset ID	**Climate Event**	**Date**	**#RDF Triples**	**# Obs**
D1	Blizzard	April, 2003	38,054,493	4,092,492
D2	Hurricane Charley	August, 2004	108,644,568	11,648,607
D3	Hurricane Katrina	August, 2005	179,128,407	19,233,458

(b) Number of Labeled Edges *NLE(G)* in datasets.

Dataset ID	**Observation NLE(G)**	**Measurement NLE(G)**
D1	24,142,314	12,071,157
D1D2	93,286,824	46,643,412
D1D2D3	207,630,306	103,815,153



(a) %age of Windspeed Repeated Triples in $D1D2D3$ (b) %age of Temperature Repeated Triples in $D1D2D3$ (c) %age of Relative Humidity Repeated Triples in $D1D2D3$

Figure 4.9: **Percentage of Repeated RDF Triples with Observation Values**. Few of the large number of values are highly repeated. (a) Percentage of repeated RDF triples with windspeed values; (b) Percentage of repeated triples with temperature values; (c) Percentage of repeated triples with humidity values.

RDF datasets comprise observations and measurements of several climate phenomena, e.g., temperature, visibility, precipitation, rainfall, and humidity, collected during the hurricane and blizzard seasons in the United States in the years 2003, 2004, and 2005[2]. Table 4.1a describes the main characteristics of these RDF datasets. Moreover, Figure 4.9 shows percentage of repeated RDF triples with wind speed, temperature, and relative humidity values in dataset $D1D2D3$. The unit of measurement is same for each type of observation. These plots show that some of the large number of observed values are highly repeated in the datasets. Further, values are not discretized to produce the same query answers.

Table 4.2: **Observation and Measurement Classes**. Sets of properties containing properties of the *Observation* and *Measurement* (Meas.) classes in the SSN ontology, each set of properties is assigned a unique ID, e.g., *A1* and *A8*.

Class	Set of Properties	SID
Observation	{property}	A1
	{samplingTime}	A2
	{procedure, generatedBy}	A3
	{property, procedure, generatedBy, samplingTime}	A4
	{property, procedure, generatedBy}	A5
	{property, samplingTime}	A6
	{procedure, samplingTime, generatedBy}	A7
Meas.	{value, unit}	A8
	{value}	A9
	{unit}	A10

---

[2] Available at: `http://wiki.knoesis.org/index.php/LinkedSensorData`

Metrics. We measure the results of our empirical evaluation in terms of number of nodes and edges in an RDF graph. The size of an RDF graph is presented as the sum of nodes and edges in the graph, where the nodes correspond to the entities and objects, whereas the edges are labeled edges annotated with the properties of a class in an RDF graph. In our empirical evaluation, we report on the following metrics: **a) Execution Time (Exec.Time(ms))** is the time required to find the frequent star patterns in RDF graphs. **b) Number of Nodes (NN)** is the number of *Observation* and *Measurement* entities and objects in RDF graphs. **c) Number of Labeled Edges (NLE)** represents the number of labeled edges annotated with the properties in *Observation* and *Measurement* classes in RDF graphs. **d) Percentage Savings in the Number of Labeled Edges (%Savings)** stands for the percentage increase or decrease in the number of labeled edges using a positive or a negative value, respectively. The interpretation of the metric **%Savings** is, higher is better.
Implementation. The experiments were performed on a Linux Debian 8 machine with a CPU Intel Xeon(R) Platinum 8160 2.10GHz and 754GB RAM. The datasets are factorized for *Observation* and *Measurement* classes using all possible combinations of the properties in each class. Table 4.2 shows the set of properties for *Observation* and *Measurement* (Meas.), respectively, in the SSN ontology. Each set of properties is assigned a set identification string *SID*, and are referred with the corresponding identification string in the paper. *Observation* contains *property*, *procedure*, *generatedBy*, and *samplingTime* property. *procedure* and *generatedBy* are inverse of each other and are considered together in the sets. Similarly, in *Measurement*, sets of properties contain the properties *value* and *unit*. Further, for experiments, datasets are gradually merged to increase datasets size. The source code is available at `https://github.com/SDM-TIB/Graph-Factorization`.

4.3.1 Efficiency of FSP Detection Approach

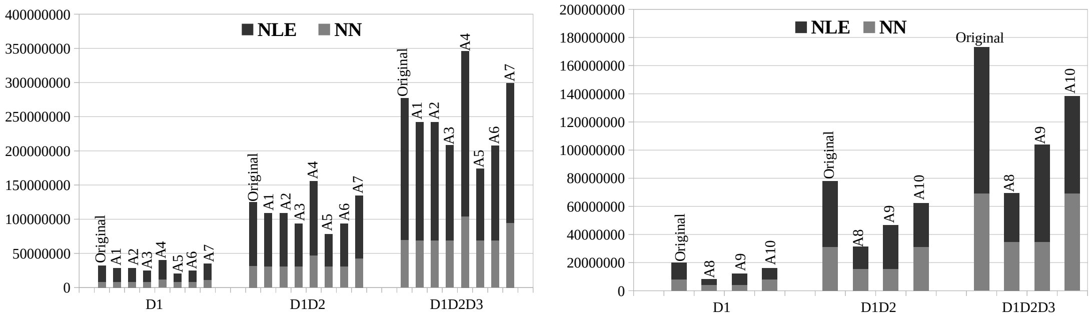
For evaluating the efficiency of the proposed frequent star patterns techniques and to answer the research question **ResearchQ1**, we execute *E.FSP* and *G.FSP* over five percent of RDF triples from dataset $D1$. The dataset of the selected RDF triples describe the *Measurement* and *Observation* classes, where several different types of observations from the *Observation* class are included in the dataset. gSpan [105] is used to generate the frequent patterns space for *E.FSP*, which iterates over all the generated frequent patterns. To evaluate the efficiency of two approaches, we selected five percent of RDF triples from the dataset $D1$; this number was chosen as a timeout because gSpan was able to generate the frequent patterns within thirty minutes. Efficiency comparison in terms of execution time of *E.FSP* and *G.FSP* is reported in Table 4.3. *G.FSP* finds the frequent star patterns without generating all the star patterns involving all the possible subsets of properties. Table 4.3 shows for *E.FSP* and *G.FSP*, the number of iterations over sets of properties *PSIterations*, the number of frequent star patterns detected *#FSP*, and the execution time in milliseconds *Exec.Time(ms)* required to detect the frequent star patterns. The results indicate that *E.FSP* and *G.FSP* detect the same frequent star patterns. The frequent star patterns, detected by *E.FSP* and *G.FSP*, are over the set of properties $A5$ and $A8$ for all the different observations in the *Observation* class, and the *Measurement* class, respectively. Execution time of *G.FSP* to detect frequent star patterns is less by at least three orders of magnitude than the execution time of *E.FSP*, e.g., *G.FSP* detects frequent star patterns in measurement class in 1.9×10^2 milliseconds, whereas 5.3×10^5 milliseconds are required using *E.FSP*.

4.3.2 Effectiveness of FSP Detection Approach

To answer the research questions **ResearchQ2** and **ResearchQ3**, we compute the values of Formula 1 for all the sets of properties given in Table 4.2 for the *Observation* and *Measurement* classes, respectively, in the three RDF datasets. The computed formula values for the *Observation* and *Measurement* classes are shown in Table 4.4. Moreover, we compute the size of the original and factorized RDF graphs, in terms of nodes and edges in the RDF graphs. The formula values

Table 4.3: **Efficiency of Frequent Star Patterns Detection.** *E.FSP* and *G.FSP* are used to detect the frequent star patterns for the *Observation* and *Measurement* classes in the five percent of RDF triples from the dataset $D1$. *E.FSP* and *G.FSP* detect the same frequent star patterns involving the sets $A5$ and $A8$ of properties from the *Observation* and *Measurement* classes, respectively. *G.FSP* takes less time to identify the same frequent star patterns than *E.FSP*.

		PSIterations		**#FSP**		**Exec.Time(ms)**	
	Class	**E.FSP**	**G.FSP**	**E.FSP**	**G.FSP**	**E.FSP**	**G.FSP**
Observation	Precipitation	8	4	23	23	2.1×10^4	1.5×10^1
	Pressure	5	4	183	183	1.3×10^6	7.1×10^2
	Rainfall	5	4	533	533	1.3×10^6	8.0×10^2
	RelativeHumidity	5	4	341	341	1.3×10^6	7.5×10^2
	Snowfall	8	4	382	382	9.2×10^5	3.1×10^2
	Temperature	5	4	395	395	1.3×10^6	7.8×10^2
	Visibility	5	4	395	395	1.3×10^6	7.3×10^2
	WindDirection	5	4	350	350	1.3×10^6	7.5×10^2
	WindSpeed	5	4	410	410	1.3×10^6	7.6×10^2
Measurement		1	1	1,907	1,907	5.3×10^5	1.9×10^2



(a) # of Observation Nodes NN and Edges NLE (b) # of Measurement nodes NN and edges NLE

Figure 4.10: **Nodes and Labeled Edges**. The number of nodes NN and labeled edges NLE before and after factorization of the RDF datasets. (a) The number of nodes NN and labeled edges NLE representing observations in the RDF datasets; (b) The number of nodes NN and labeled edges NLE representing measurements.

are computed for the sets of properties that contain only one property in the set, as well as the factorization is performed using these sets of properties to illustrate the association between the formula values and the savings obtained in the factorized graphs. Table 4.4 shows that the set $A5$ of properties in the *Observation* class generates the smaller values $D1 = 4,142,727$, $D1D2 = 15,756,888$, and $D1D2D3 = 334,898,603$ for the Formula 1, than all the other sets $A1$, $A2$, $A3$, $A4$, $A6$, and $A7$. A smaller formula value for $A5$ indicates that the RDF graphs encapsulate a minimum number of star patterns, over the properties in the set $A5$ such that a large number of entities of the *Observation* class match these star patterns. Therefore, replacing these star patterns with the compact RDF molecules during the factorization reduces the size of the RDF graphs. Figure 4.10a presents the number of *Observation* nodes NN and the labeled edges NLE in the original and factorized RDF datasets $D1$, $D1D2$, and $D1D2D3$. The results show that factorization of the *Observation* class over the set $A5$ of properties reduces the sum of the number of observation nodes and the labeled edges in the factorized RDF graphs by up to 37%. On contrary, a large formula value for $A4$ in datasets $D1 = 4,142,727$, $D1D2 = 15,756,888$, and $D1D2D3 = 334,898,603$, than the other sets $A1$, $A2$, $A3$, $A5$, $A6$, and $A7$ indicates that a large number of star patterns over the properties in $A4$ exist in the RDF graphs and a small number of entities of the *Observation* class match these star patterns. Figure 4.10a depicts an increase in the number of *Observation* nodes NN and the labeled edges NLE in the factorized RDF datasets $D1$, $D1D2$, and $D1D2D3$ after factorizing over the properties in $A4$. Similarly, the results for $A1$, $A2$, $A3$, $A6$, and $A7$ in Table 4.4 and Figure 4.10a clearly show that the higher the formula value for a set of properties increases the number of nodes and edges in the factorized RDF graphs by factorizing using the properties in the corresponding set. In case of the *Measurement* class Table 4.4 shows smaller formula values for the set $A8$ of properties, i.e., $D1 = 28,491$, $D1D2 = 34,554$, and $D1D2D3 = 40,302$, than the other sets $A9$ and $A10$. Figure 4.10b reports the sum of the nodes and the labeled edges representing measurements in the original and factorized RDF datasets $D1$, $D1D2$, and $D1D2D3$. The sum of the nodes and the labled edges of the measurements are reduced up to 60% in all the factorized RDF graphs by factorizing over the properties in $A8$. Furthermore, the higher formula values for the sets $A9$ and $A10$ indicate less savings after factorization compared to the set $A8$. The number of nodes and edges in the factorized RDF graphs by factorizing over the properties in sets $A9$ and $A10$ in Figure 4.10b are higher than $A8$. These results show that the different combinations of class properties impact on the factorization of RDF graphs and the proposed frequent star patterns detection techniques are able to detect the set of properties involved in the generation of frequent star patterns. Moreover, our techniques are able to anticipate the best set of properties, answering thus, research questions **ResearchQ2** and **ResearchQ3**.

Table 4.4: **Values Computed for Formula 1.** The sets of *Observation* and *Measurement* (Meas.) properties in Table 4.2 are used to compute the Formula 1 values over the RDF datasets $D1$, $D1D2$, and $D1D2D3$. The minimum formula values for the *Observation* and *Measurement* classes and the corresponding sets $A5$ and $A8$, respectively, of properties are highlighted in bold. The smaller formula values for $A5$ and $A8$ in the *Observation* and *Measurement* classes, respectively, indicate the maximum savings after factorizing the RDF graphs over the properties in $A5$ and $A8$, as shown in Figure 4.10 and Table 4.5.

		#Edges(SP, C, G)		
	SID	**D1**	**D1D2**	**D1D2D3**
Observation	A1	12,071,185	46,643,440	103,815,183
	A2	12,090,195	46,687,690	103,891,717
	A3	8,111,623	31,205,888	69,358,875
	A4	20,118,595	78,698,580	174,865,870
	A5	**4,142,727**	**15,756,888**	**34,898,603**
	A6	8,097,964	31,245,605	69,474,786
	A7	15,784,707	61,406,644	135,902,747
Meas.	**A8**	**28,491**	**34,554**	**40,302**
	A9	4,037,067	15,563,838	34,623,579
	A10	4,023,731	15,547,816	34,605,063

4.3.3 Effectiveness of RDF Graph Factorization

We factorize the gradually increasing RDF datasets $D1$, $D1D2$, and $D1D2D3$ over the *Observation* and *Measurement* classes using the properties in the sets of properties given in Table 4.2. The percentage savings are computed in terms of labeled edges for the observations and measurements in the RDF datasets after factorization. Table 4.1b presents the number of edges $NLE(G)$ in the *Observation* and *Measurement* classes in the original RDF datasets $D1$, $D1D2$, and $D1D2D3$. Table 4.5 presents the number of labeled edges $NLE(G')$ and the percentage savings $\%savings$ after factorization of the *Observation* and *Measurement* classes. The highest savings 49.14%, 49.43%, and 49.53% in $NLE(G')$ for observations after factorizing $D1$, $D1D2$, and $D1D2D3$ over the properties in $A5$, shows that the number of frequent star patterns over the properties in $A5$ are reduce by replacing them with the corresponding compact RDF molecules. On the other hand, the set $A4$ of properties gives negative values of percentage savings $\%Savings$, -16.68%, for the RDF dataset $D1$, and -16.67%, for the RDF datasets $D1D2$ and $D1D2D3$, indicating an increase in the number of labeled edges after the factorization of the RDF datasets. Similarly, for measurements, the positive values 66.37% of percentage savings after factorizing $D1$, and 66.56% for $D1D2$ and $D1D2D3$ over $A8$ indicate a decrease in the number of labeled edges after factorization. Furthermore, the percentage savings in the set $A8$ of properties are higher than in $A9$ and $A10$. These results allow us to positively answer research question **ResearchQ4**.

Table 4.5: **Percentage Savings in Labeled Edges after Factorization.** Savings *%Savings* in the number of Labeled Edges *NLE(G′)* after factorization of the RDF datasets using the sets of properties in Observation and Measurement classes.

		D1		**D1D2**		**D1D2D3**	
	SID	**NLE(G')**	**%Savings**	**NLE(G')**	**%Savings**	**NLE(G')**	**%Savings**
Observation	A1	20,125,493	16.64	77,745,918	16.66	173,032,155	16.66
	A2	20,144,503	16.56	77,790,168	16.61	173,108,689	16.63
	A3	16,226,021	32.79	62,546,938	32.95	139,064,503	33.02
	A4	28,170,155	-16.68	108,838,750	-16.67	242,239,479	-16.67
	A5	**12,277,576**	**49.14**	**47,175,356**	**49.43**	**104,786,128**	**49.53**
	A6	16,150,898	33.10	62,317,489	33.20	138,639,234	33.23
	A7	23,837,352	1.26	92,088,523	1.28	204,304,156	1.60
Meas.	**A8**	**4,059,738**	**66.37**	**15,599,469**	**66.56**	**34,716,176**	**66.56**
	A9	8,069,688	33.15	31,130,127	33.26	69,300,827	33.25
	A10	8,056,352	33.26	31,114,105	33.29	69,282,311	33.26

4.4 Summary

Representing observational data directly in RDF knowledge graphs results in a substantial inflation of the data volume due to redundancies. We devise techniques to detect redundancies in historical data and propose factorized representations of RDF knowledge graphs where these redundancies are reduced. The proposed techniques are analysed using existing benchmarks. The results suggest that the proposed techniques are able to identify data redundancies in RDF knowledge graphs, and the factorized representations are able to reduce size of RDF knowledge graphs while the information initially encoded in the data is preserved.

Chapter 5

Integration of Historical Semantic Sensor Data

Sensors are frequently used in a large spectrum of demanding application domains, including, eHealth, precision agriculture, as well as smart environments, etc. An enormous amount of sensor data is produced by these sensors. Data semantics facilitate information exchange, adaptability, and interoperability among several devices and sensors. As a result, there have been proposed a large number of diverse RDF implementations to store and process RDF data. Collections of sensor data can be semantically described using ontologies, e.g., the Semantic Sensor Network (SSN) Ontology. Albeit semantically enriched, the volume of semantic sensor data is considerably larger than raw sensor data, hence negatively impact on the storage and processing of semantic sensor data. Moreover, some measurement values can be observed several times, and a large number of repeated facts can be generated. Chapter 4 presents computational methods to detect redundancies in RDF knowledge graphs and identifies properties of a class that are involved in generating these redundancies. Moreover, a general factorization algorithm for factorizing RDF knowledge graph is proposed. The factorization algorithm receives a class and a set of properties in the class to generate factorized RDF representations of a knowledge graph. In this chapter, we exploit the properties identified by the approach proposed in Chapter 4 and the domain knowledge of the SSN ontology to factorize the RDF knowledge graph describing sensor data. Furthermore, based on the factorized semantic sensor data, we present tabular-based representations of sensor data in order to scale up to large datasets. In addition, we analyze the impact of the proposed factorized representations on query execution. In this chapter, we address the problem of efficient query processing over historical semantic sensor data. Figure 5.1 presents the challenge we tackle to solve the problem and the contribution to address the challenge. The research work presented in this chapter is based on the publications [52, 55]. The following research question is

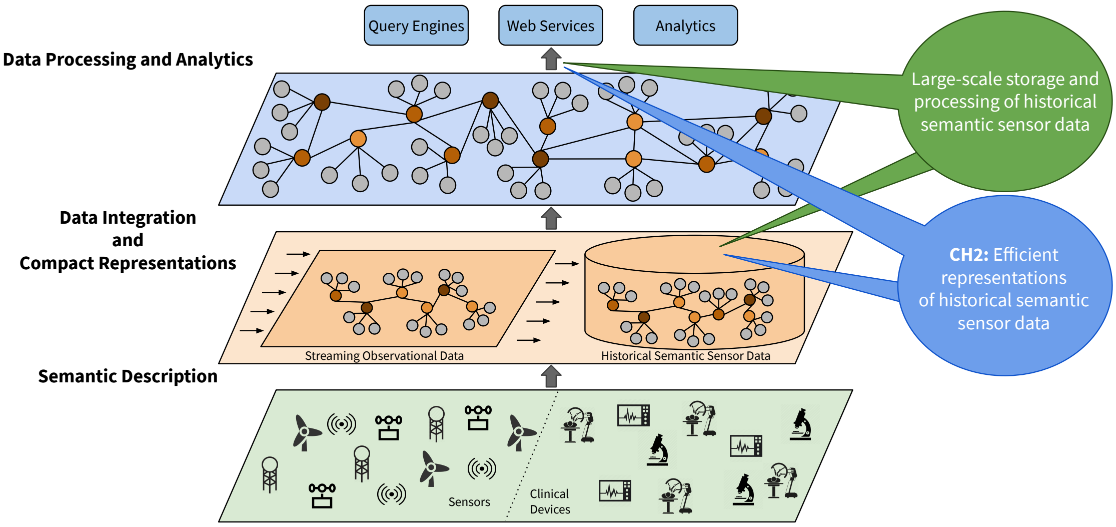

Query Engines
Web Services
Analytics
Data Processing and Analytics
Large-scale storage and processing of historical semantic sensor data
Data Integration and Compact Representations
CH2: Efficient representations of historical semantic sensor data
Streaming Observational Data
Historical Semantic Sensor Data
Semantic Description
Sensors
Clinical Devices

Figure 5.1: **Challenges and Contributions.** This chapter focuses on the problem of efficient representations of historical semantic sensor data, and implements query processing techniques to scale up to large historical semantic sensor datasets.

addressed in this chapter:

RQ2: How can efficient representations be exploited to manage historical semantic sensor data?

To answer this research question, we propose *Compacting Semantic Sensor Data (CSSD)* approach that generates a *compact* or *factorized* representations of historical semantic sensor data, where repeated values are represented only once. Furthermore, these compact representations are able to enhance the storage and processing of historical semantic sensor data. To scale up to large datasets, tabular representations are utilized to store and manage factorized semantic sensor data using Big Data technologies. In essence, this chapter makes the following contributions to the problem of processing historical semantic sensor data:

- The *CSSD* approach using factorization techniques for a compact representation of semantic sensor data described using the SSN ontology;
- Tabular representations of semantic sensor data to scale-up to large datasets;
- SPARQL query rewriting techniques against factorized semantic sensor data; and
- An empirical evaluation of the proposed compact representations along several dimensions demonstrating the effectiveness and efficiency of the *CSSD*

approach over the diverse RDF implementations.

The chapter is structured as follows: We motivate the research problem in Section 5.1. We present an RDF graph describing more than five thousand weather observations collected from sensors. The RDF graph visualizations and statistics show that the graph is composed of a large number of RDF triples, where several RDF triples are shared among different observations. Furthermore, we illustrate that a high number of RDF triples are related to a unique observation value. A formal description of the proposed approach is discussed in Section 5.2, where we introduce the factorization techniques for semantic sensor data to scale-up to large datasets. We devise tabular-based representations in Section 5.3 for efficient storage and processing of semantic sensor data using Big Data frameworks. We present evaluation results of an experimental study in Section 5.4. We empirically study the effectiveness of the proposed compact representations of semantic sensor data, and the impact of these compact representations on query processing. Additionally, we evaluate the effects of storing the proposed representations of semantic sensor data on several RDF implementations. Results suggest that the proposed compact representations of semantic sensor data empower the storage and query processing of sensor data over diverse RDF implementations. We provide a summary of the achieved results in Section 5.5.

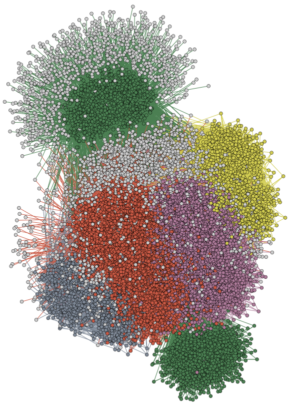
5.1 Motivating Example

The *MesoWest LinkedObservation*[2] datasets comprise sensor data describing hurricane and blizzard observations in the United States. Observations include measurements of several weather phenomena, e.g., snowfall, rainfall, wind direction, wind speed, humidity, temperature, and precipitation. These sensor observations are semantically described using the Semantic Sensor Network (SSN) ontology. Together, these *LinkedObservations* contain almost two billion RDF triples describing major active storms in the United States since 2002. The RDF observational data on the storm season in the year 2004 contain 108,644,568 RDF triples about different weather phenomena, and 11,648,607 observations about rainfall, precipitation, wind direction, relative humidity, and temperature. Figure 5.2a depicts the graph of semantic sensor observations about pressure, rainfall, wind direction, temperature, and visibility, as well as observation timestamps from year 2004 MesoWest dataset. The RDF graph illustrates 46,341 RDF triples describing 5,149 linked observations. The same color nodes and edges in the graph represent

---

[1] `http://www.cytoscape.org/`

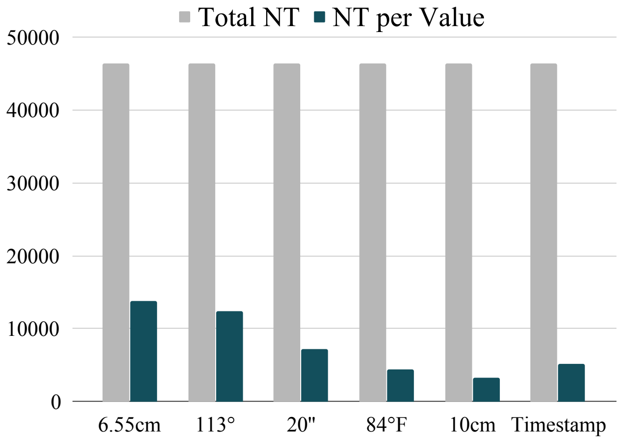
[2] `http://wiki.knoesis.org/index.php/LinkedSensorData`



(a) Original **RDF Graph**

S#	Parameter	Value
1	Connected Components	1.000
2	Network Centralization	0.328
3	**Avg. # of Neighbors**	**6.361**
4	Network Density	0.000
5	Multi-edge Node Pairs	5,149.000
6	Network Heterogeneity	11.106

(b) **Statistics** of Original RDF Graph



(c) **NT per Value** vs Total Triples

Figure 5.2: **Motivating Example.** Several RDF triples are related to same values in an RDF graph. (a) An RDF Graph consists of $46,341$ RDF triples and describes $5,149$ weather observations about pressure, rainfall, wind direction, temperature, and visibility collected by the sensors deployed in $\sim 20,000$ weather stations in the United States in year 2004. The same color of nodes and edges in the RDF graph represents the RDF triples associated with same values; (b) Statistics of the RDF graph reveals the existence of several links between the nodes. The high value 6.361 of *avg. # of neighbours* indicates that on average a node is connected to more than six other nodes in the RDF graph. Similarly, a high value $5,149$ of *multi-edge node pairs* shows the existence of several node pairs, where nodes in each node pair are connected using multiple edges in the RDF graph; (c) Number of RDF triples, associated with the same value, with respect to total RDF triples in the RDF graph in Figure 5.2a. The RDF graph and statistics in Figures 5.2a and 5.2b, respectively, are generated by `Cytoscape tool`[1].

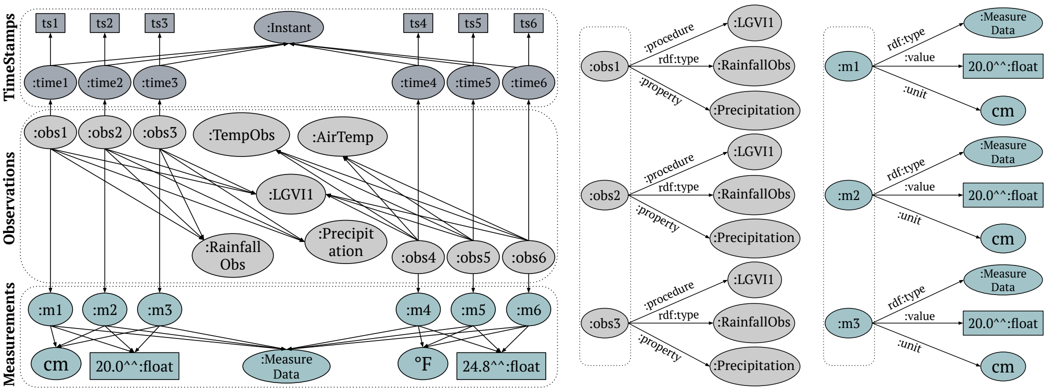
the RDF triples associated with the same measurement value. The RDF triples describing timestamps are also represented with the same color nodes and edges. Statistics of the RDF graph, shown in Figure 5.2b, reflect the existence of highly redundant inter-connectivity among the RDF nodes. The RDF graph and the statistics reveal that the RDF triples are duplicated with the repetition of measurement values. Also, each sensor observation is associated with seven neighbors in average, i.e., observations are described in seven RDF triples in average. Figure 5.2c illustrates the number of RDF triples per distinct value within the dataset. Rainfall measurement value, `6.55 cm`, is the most repeated and is associated with 15,552 RDF triples, and 113° wind direction is the second most repeated measurement value and is associated with 13,941 RDF triples. Similarly, the number of RDF triples associated with distinct measurement values can be observed for other climate phenomena, e.g., pressure, temperature, and visibility, and substantiate the *natural intuition* that the number of *distinct values* is much smaller than the number of *observations*. We exploit this natural intuition of semantic sensor data, and propose a compact representation. In these compact representations, RDF triples of repeated measurements are *factorized* and added to the dataset only once. Unlike other RDF compression techniques, semantics of observational data are utilized to factorize the semantic sensor data, where the factorized representations provide efficient storage over diverse RDF implementations, and queries can be directly executed against factorized datasets. To scale up to large datasets, factorization based tabular representation can be devised to store and manage large amount of semantic sensor data using Big Data technologies.

5.2 The Semantic Sensor Data Factorization Approach

5.2.1 Problem Statement

The concept of RDF molecule (presented in chapter 2) is utilized to devise observation and measurement molecules based on the Semantic Sensor Network Ontology. Moreover, we present the concepts of measurement and observation multiplicity as the number of times measurements and observations are repeated in an RDF graph. Building on these definitions the problem tackled in this chapter is defined.

Definition 5.2.1 (Observation Molecule). *An observation molecule OM is a set of RDF triples that share the same subject of type observation class, i.e.,*
OM= (obs `rdf:type` *:Observation), (obs* `:procedure` *proc), (obs* `:property` *pp).*



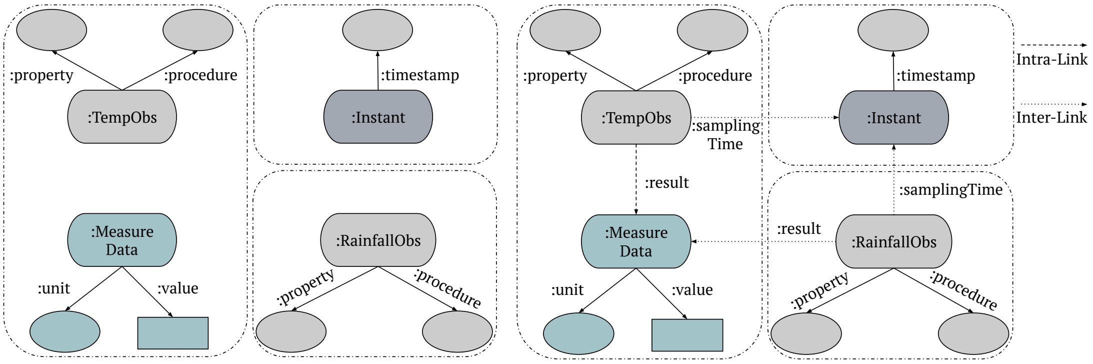
(a) RDF Graph G (b) Observation Molecules (c) Measurement Molecules

Figure 5.3: **Example of a Simplified RDF.** Several RDF triples are related to the same measurement values for simplicity URIs are not presented. (a) RDF graph G has rainfall measurements, `:m1`, `:m2`, and `:m3` and observations, `:obs1`,`:obs2`, and `:obs3`, which are related to the value 20.0 for precipitation, and are duplicated. Measurement and Observation multiplicities are 3, i.e., $M_m(val, uom|G) = 3$ and $M_o(obs, proc, pp, val, uom|G) = 3$. Similarly, temperature measurements, `:m4`, `:m5`, and `:m6` and observations, `:obs4`,`:obs5`, and `:obs6` are related to the value 24.8 for air temperature, and are duplicated with Measurement and Observation multiplicities 3, i.e., $M_m(val, uom|G) = 3$ and $M_o(obs, proc, pp, val, uom|G) = 3$. (b) An RDF graph with three observation molecules; each observation is associated with only three RDF triples describing observation type, observed property, and sensor procedure; (c) An RDF graph with three measurement molecules in class `:MeasureData`; each measurement is described using three RDF triples to express the measurement type, measured value and the unit of measurement.

Example 5.2.1. *Figure 5.3b presents three observation molecules. Each observation molecule consists of three RDF triples describing the same observation subject, i.e.,* $obs1$, $obs2$, *and* $obs3$. *Each observation subject is described in terms of observation type, observed property and the observation procedure.*

Definition 5.2.2 (Measurement Molecule)**.** *A measurement molecule* MM *is a set of RDF triples that share the same measurement subject, i.e.,* $MM =$ (m `rdf:type` *:MeasureData*), (m `:value` val), (m `:unit` uom).

Example 5.2.2. *Figure 5.3c presents three measurement molecules. Each measurement molecule consists of three RDF triples having the same measurement subject, i.e.,* `:m1`, `:m2`, *and* `:m3`. *Each measurement is described in terms of measured value and unit using properties* `:value` *and* `:unit`, *respectively.*



(a) RDF Molecule Templates (RDF-MTs) (b) RDF-MT Intra- and Inter-dataset Linking

Figure 5.4: **RDF Molecule Templates (RDF-MTs) and RDF-MT Linking**. (a) Four RDF Molecule Templates are extracted from the RDF graph in Figure 5.3a around the classes `:TempObs`, `:RainfallObs`, `:MeasureData`, and `:Instant`. The classes `:TempObs` and `:MeasureData` belong to the same dataset, and `:RainfallObs` and `:Instant` belong to two other datasets. (b) RDF-MT around `:TempObs` is linked to `:MeasureData` RDF-MT in the same dataset by `:result`. `:TempObs` RDF-MT is linked to `:Instant` RDF-MT in a different dataset using property `:samplingTime`, and `:RainfallObs` RDF-MT is linked to `:MeasureData` RDF-MT in the other dataset using property `:result`.

RDF molecule templates are the abstract descriptions of the triples in RDF graphs and are defined as follows:

Definition 5.2.3 (RDF Molecule Template (RDF-MT) [34]). *An RDF Molecule Template (RDF-MT) is a* $5-tuple =< WebI, C, DTP, IntraL, InterL >$, *where:*

- $WebI$ *is a Web service API that provides access to an RDF dataset G via SPARQL protocol;*
- C *is an RDF class such that the triple pattern* $(?s$ `rdf:type` $C)$ *is true*[3] *in* G*;*
- DTP *is a set of pairs* (p, T) *such that* p *is a property with domain* C *and range* T*, and the triple patterns* $(?s\ p\ ?o)$ *and* $(?o$ `rdf:type` $T)$ *and* $(?s$ `rdf:type` $C)$ *are true in* G*;*
- $IntraL$ *is a set of pairs* (p, C_j) *such that* p *is an object property with domain* C *and range* C_j *, and the triple patterns* $(?s\ p\ ?o)$ *and* $(?o$ `rdf:type` $C_j)$ *and* $(?s$ `rdf:type` $C)$ *are true in* G*;*

---

[3] Evaluating the triple pattern into true corresponds to the ASK query.

- *InterL is a set of triples (p, C_k, SW) such that p is an object property with domain C and range C_k ; SW is a Web service API that provides access to an RDF dataset K, and the triple patterns (?s p ?o) and (?s* `rdf:type` *C) are true in G, and the triple pattern (?o* `rdf:type` *C_k) is true in K.*

Example 5.2.3. *Figure 5.4 shows an example of RDF molecule templates extracted from Figure 5.3a and connections between these RDF-MTs. Four RDF-MTs are extracted around the classes* `:TempObs`, `:RainfallObs`, `:MeasureData`, *and* `:Instant`. *Each RDF-MT around a class contains the properties describing the class. Moreover, Figure 5.4b shows intra-linking between* `:TempObs` *and* `:MeasureData` *RDF-MTs in the same dataset using property* `:result`. `:TempObs` *and* `:Instant` *RDF-MTs are interlinked using property* `:samplingTime`. *Similarly,* `:RainfallObs` *RDF-MT is interlinked to* `:Instant` *using* `:samplingTime`.

Definition 5.2.4 (Measurement Multiplicity). *Given an RDF graph G of sensor data using the SSN ontology. Given a resource* `uom` *corresponding to a measurement unit, and a literal value* `val`, *the measurement multiplicity of* `uom` *and* `val` *in G, $M_m(val, uom|G)$, is defined as the number of measurements have same value val and measurement unit uom in G.*

$$\begin{aligned} M_m(val, uom|G) = |\{m|\ & (m\ \texttt{rdf:type :MeasureData}) \in G, \\ & (m\ \texttt{:unit}\ uom) \in G, \\ & (m\ \texttt{:value}\ val) \in G\}| \end{aligned}$$

Example 5.2.4. *In the RDF graph shown in Figure 5.3a, there are three measurements of type* `:MeasureData`, *represented by* `:m1`, `:m2`, *and* `:m3` *that are associated with the measurement unit* `cm` *and the value 20.0. Therefore, the measurement multiplicity of measurement unit* `cm` *and the value 20.0 is 3. Similarly, three measurements* `:m4`, `:m5`, *and* `:m6` *of type* `:MeasureData` *are associated with the measurement unit °*`F` *and the value 24.8 of multiplicity 3.*

Definition 5.2.5 (Observation Multiplicity). *Given an RDF graph G of sensor data described using the SSN ontology. Given resources* `proc`, `ph`, `pp`, *and* `uom` *corresponding to a procedure, an observed phenomenon, observed property, and measurement unit, and a literal value* `val`, *the multiplicity of an observation* `obs` *for* `uom` *and* `val` *in G, $M_o(proc, ph, pp, val, uom|G)$, is defined as the number of observations about the property* `pp` *of the observed phenomenon* `ph`, *sensed by* `proc`, *that have the same value* `val` *and unit of measurement* `uom` *in G.*

$$\begin{aligned} M_o(proc, ph, pp, val, uom|G) = |\{obs|\ & (obs\ \texttt{rdf:type}\, ph) \in G, \\ & (obs\ \texttt{:procedure}\, proc) \in G, \\ & (obs\ \texttt{:property}\, pp) \in G, \\ & (obs\ \texttt{:result}\, m) \in G, \\ & (m\ \texttt{rdf:type :MeasureData}) \in G, \\ & (m\ \texttt{:unit}\, uom) \in G, \\ & (m\ \texttt{:value}\, val) \in G\}| \end{aligned}$$

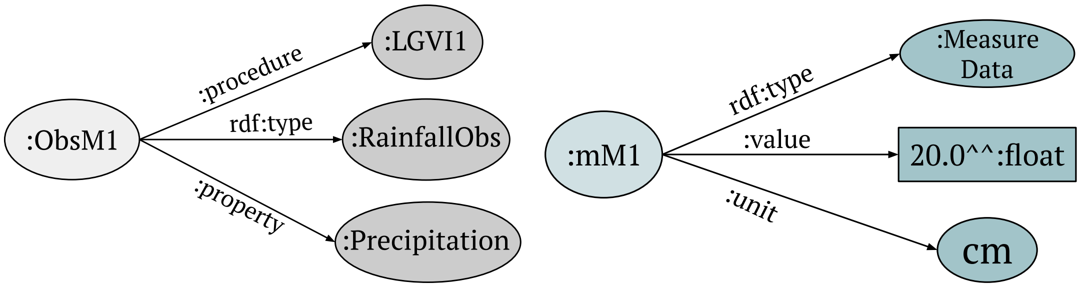
Example 5.2.5. *In Figure 5.3a, the observation multiplicity for the procedure* `:LGVI1`*, the observed phenomenon* `:RainfallObs`*,* `:Precipitation` *as observed property, measurement unit* `cm`*, and value* 20 *is* 3*, i.e., there are three different observations,* `:obs1`*,* `:obs2`*, and* `:obs3`*, associated with the same properties, and value and unit. Similarly, the observation multiplicity for the procedure* `:LGVI1`*, the observed phenomenon* `:TempObs`*, the observed property* `:AirTemp`*, the measurement unit* °`F`*, and the value* 24 *is* 3*, i.e., there are three different observations,* `:obs4`*,* `:obs5`*, and* `:obs6` *associated with the same properties, and value and unit.*

Definition 5.2.6 (Compact Observation Molecule)**.** *Given a surrogate observation oM, a compact observation molecule COM is a set of RDF triples that share the same surrogate observation oM, i.e., $COM =$ (oM* `rdf:type` *:Observation),(oM* `:procedure` *proc),(oM* `:property` *pp), such that the multiplicity of the surrogate observation oM is one.*

Example 5.2.6. *Figure 5.5a presents a compact observation molecule, with a surrogate observation* `:obsM1`*, for the observation molecules in Figure 5.3b. The compact observation molecule represents all the properties and corresponding objects as in the observation molecules in Figure 5.3b. However, the redundant edges, repeatedly connecting the similar type of observations* `:obs1`*,* `:obs2`*, and* `:obs3` *to a set of objects using the same properties, are transformed into the edges connecting these properties and corresponding objects to the surrogate observation* `:obsM1`*.*

Definition 5.2.7 (Compact Measurement Molecule)**.** *Given a surrogate measurement mM, a compact measurement molecule CMM is a set of RDF triples that share the same surrogate measurement mM, i.e., $CMM =$ (mM* `rdf:type` *:MeasureData),(mM* `:value` *val),(mM* `:unit` *uom), such that the multiplicity of the surrogate measurement mM over val and uom is one.*

Example 5.2.7. *A compact measurement molecule for the measurement molecules in Figure 5.3c is presented in Figure 5.5b with a surrogate measurement* `:mM1`*. The surrogate measurement* `:mM1` *corresponds to the measurements* `:m1`*,* `:m2`*, and* `:m3` *in Figure 5.3c and has multiplicity one.*



(a) Compact Observation Molecule (b) Compact Measurement Molecule

Figure 5.5: **Compact Observation and Measurement Molecules**. Compact molecules for the observations and measurements in Figures 5.3b and 5.3c, respectively. (a) A compact observation molecule, for the observations in Figure 5.3b, with a surrogate observation `:obsM1` and multiplicity one, corresponds to the observations `:obs1`, `:obs2`, and `:obs3`. (b) A compact measurement molecule, with surrogate measurement `:mM1` of multiplicity one, corresponds to the measurements `:m1`, `:m2`, and `:m3` in Figure 5.3c.

Definition 5.2.8 (A Factorized RDF Graph). *Given an RDF graph $G = (V_G, E_G, L_G)$ representing sensor data described using the SSN ontology, a factorized RDF graph $G' = (V_{G'}, E_{G'}, L_{G'})$ with respect to G is an RDF graph where the following conditions hold:*

- *Entities in G are preserved in G', i.e., $V_G \subseteq V_{G'}$.*
- *For each entity obs in V_G that corresponds to an entity of class `:Observation` over the properties `:procedure` and `:property` and objects proc and pp, respectively, in G, there is an entity oM in $V_{G'}$ that corresponds to the surrogate observation of the compact observation molecule over the properties `:procedure` and `:property` and objects proc and pp, respectively, in G'.*
- *For each entity m in V_G that corresponds to an entity of class `:MeasureData` over the properties `:value` and `:unit` and objects val and uom, respectively, in G, there is an entity mM in $V_{G'}$ that corresponds to the surrogate measurement of the compact measurement molecule over the properties `:value` and `:unit` and objects val and uom, respectively, in G'.*
- *There is a partial mapping μ_N: $V_G \rightarrow V_{G'}$:*
  - *Subject entities of observation molecules in G are mapped to the surrogate observations of the compact observation molecules in G', i.e., $\mu_N(obs){=}oM$.*

  - *Subject entities of measurement molecules in G are mapped to the surrogate measurements of the compact measurement molecules in G', i.e., $\mu_N(m)$=mM.*
  - *The mapping μ_N is not defined for the rest of the entities that are not instances of the* `:Observation` *or* `:MeasureData` *class in G.*

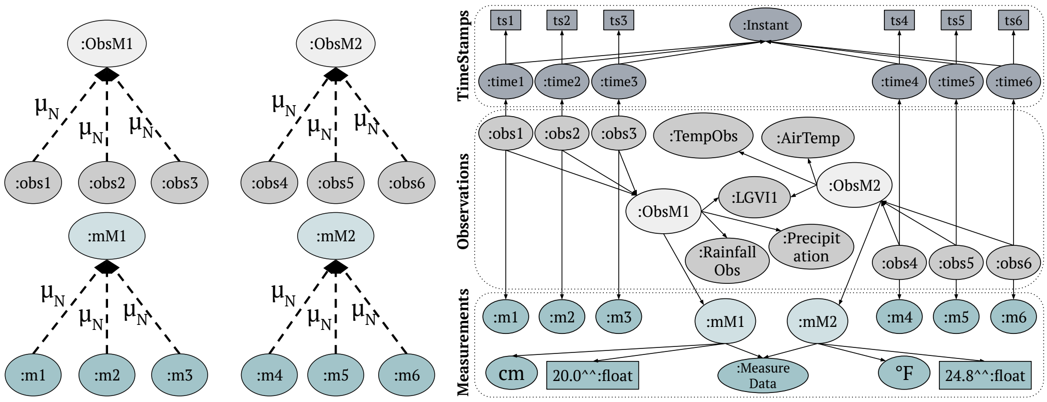
- *For each RDF triple t in (s p o) in E_G:*
  - *If $\mu_N(s)$ is defined and* `:Observation` *is the type of s, then the RDF triples*
    (s `:instanceOf` *$\mu_N(s)$) and ($\mu_N(s)$* `rdf:type :Observation`*) belong to $E_{G'}$.*
  - *If $\mu_N(s)$ is defined and* `:MeasureData` *is the type of s, then the RDF triple*
    ($\mu_N(s)$ `rdf:type :MeasureData`*) belong to $E_{G'}$.*
  - *If $\mu_N(s)$ is defined and* `:Observation` *is the type of s, and p is not* `:result` *and* `:samplingTime`*, then ($\mu_N(s)$ p o) is in $E_{G'}$.*
  - *If p is* `:samplingTime`*, then (s p o) is in $E_{G'}$.*
  - *If $\mu_N(s)$ and $\mu_N(o)$ are defined and p is* `:result`*, then ($\mu_N(s)$ p $\mu_N(o)$) and (s p o) are in $E_{G'}$.*
  - *If $\mu_N(s)$ is defined and* `:MeasureData` *is the type of s, then ($\mu_N(s)$ p o) is in $E_{G'}$.*
  - *Otherwise, the RDF triple t is preserved in $E_{G'}$.*

- *Multiplicity of measurements is reduced, i.e., for all val, uom such that $M_m(val, uom|G) \geq 1$, then $M_m(val, uom|G')$=1, and*

- *Multiplicity of observations is reduced, i.e., for all proc, ph, pp, val, uom, such that, $M_o(proc, ph, pp, val, uom|G) \geq 1$, then $M_o(proc, ph, pp, val, uom|G')$=1.*

Definition 5.2.9 (The SSDF Problem). *Given an RDF graph $G = (V_G, E_G, L_G)$ representing sensor data with the SSN ontology, the problem of semantic sensor data factorization (SSDF) in G, corresponds to finding a factorized RDF graph $G' = (V_{G'}, E_{G'}, L_{G'})$ of G.*

Example 5.2.8. *Consider RDF graphs G and G' in Figures 5.3a and 5.6b, respectively. Furthermore, Figure 5.6a presents mappings μ_N that assign measurement nodes* `:m1`*,* `:m2`*, and* `:m3` *in G to the surrogate measurement* `:mM1` *in G', and* `:m4`*,*



(a) Entity mappings μ_N from G into G' (b) Factorized RDF graph G'

Figure 5.6: **Instance of the Semantic Sensor Data Factorization Problem**. A Factorized RDF graph G' computed from the RDF graph G in Figure 5.3a. (a) Entity mappings μ_N from G in Figure 5.3a, maps measurements `:m1`, `:m2`, and `:m3` into the surrogate measurement `:mM1`, and `:m4`, `:m5`, and `:m6` into the surrogate measurement `:mM2`, and observations `:obs1`, `:obs2`, and `:obs3` into the surrogate observation `:oM1`, and `:obs4`, `:obs5`, and `:obs6` into `:oM2`; (b) Factorized RDF Graph G' with no duplicated nodes of measurements and observations, i.e., multiplicity of measurements and observations is one in G'. Mappings between the observations and the surrogate observation are explicitly stated, while the mappings between measurements and the surrogate measurements are represented through a path in the RDF graph.

`:m5`, and `:m6` in G to the surrogate measurement `:mM2` in G'. Similarly, observation nodes `:obs1`, `:obs2`, and `:obs3` in G are mapped to the surrogate observation `:obsM1` in G', and `:obs4`, `:obs5`, and `:obs6` in G are mapped to the surrogate observation `:obsM2` in G'; μ_N is the identity for the rest of the nodes. Moreover, surrogate measurement and observation multiplicities are 1 for corresponding nodes in G'. Thus, G' is the factorized graph of G where duplicated nodes of measurements and observations in G are represented with the relevant surrogate nodes in G', e.g., `:mM1` instead of `:m1`, `:m2`, and `:m3`.

Once RDF graphs are factorized, query processing is performed against the factorized graphs. SPARQL queries over the original RDF graphs need to be rewritten against the corresponding factorized RDF graphs in the way that equivalent answers are computed. We have defined seven query rewriting rules, given in Table 5.1. Each rule is given a name, i.e., *fssn1*, *fssn2*, *fssn3*, *fssn4*, *fssn5*, *fssn6*, and *fssn7* and has a head and a body. The head of a rule corresponds to the triple pattern in the query against original RDF graph, whereas the body of the rule represents the corresponding triple patterns against the factorized RDF graph. The head of the rule *fssn1* contains a triple pattern that matches to all the observation entities in original RDF graphs. On the other hand, the body of the rule *fssn1* comprises the triple patterns that match the corresponding surrogate observations and associates the original observations to the surrogate observations using the property `:instanceOf` in factorized RDF graphs, as well as maintains the variable substitutions, i.e., $?obs$ is replaceable by $?Xobs$. These variable substitutions are maintained for the query clauses such as SELECT, FILTER, GROUP BY, ORDER BY etc. The head of rule *fssn2*, matches the procedure of the observations in the original RDF graph, while the body of the rule extracts the procedure of the surrogate observations in factorized RDF graphs as well as the associations between original and surrogate observations, and keeps the observation variable

Table 5.1: **Query Rewriting Rules**. The rewriting rules for observations and measurements with respect to the relevant properties are expressed in terms of triple patterns. The variables corresponding to observations and measurements are replaced in SPARQL query clauses, i.e., SELECT, ORDER BY, GROUP BY, and FILTER.

Rule Name	**Head**	**Body**
fssn1	$?obs$ *rdf:type :Observation*	$?obs$ *rdf:type* *:Observation* $?Xobs$ *:instanceOf* $?obs$ Replace $?obs$ by $?Xobs$ in query clauses
fssn2	$?obs$ *:procedure* $?sensor$	$?obs$ *:procedure* $?sensor$ $?Xobs$ *:instanceOf* $?obs$ Replace $?obs$ by $?Xobs$ in query clauses
fssn3	$?obs$ *:property* $?property$	$?obs$ *:property* $?property$ $?Xobs$ *:instanceOf* $?obs$ Replace $?obs$ by $?Xobs$ in query clauses
fssn4	$?m$ *rdf:type :MeasureData*	$?m$ *rdf:type* *:MeasureData* $?Xobs$ *:instanceOf* $?obs$ $?Xobs$ *:result* $?Xm$ Replace $?m$ by $?Xm$ in query clauses
fssn5	$?m$ *:value* $?val$	$?m$ *:value* $?val$ $?Xobs$ *:instanceOf* $?obs$ $?Xobs$ *:result* $?Xm$ Replace $?m$ by $?Xm$ in query clauses
fssn6	$?m$ *:unit* $?uom$	$?m$ *:unit* $?uom$ $?Xobs$ *:instanceOf* $?obs$ $?Xobs$ *:result* $?Xm$ Replace $?m$ by $?Xm$ in query clauses
fssn7	$?obs$ *:result* $?m$	$?obs$ *:result* $?m$ $?Xobs$ *:instanceOf* $?obs$ $?Xobs$ *:result* $?Xm$ Replace $?obs$ by $?Xobs$ and $?m$ by $?Xm$ in query clauses

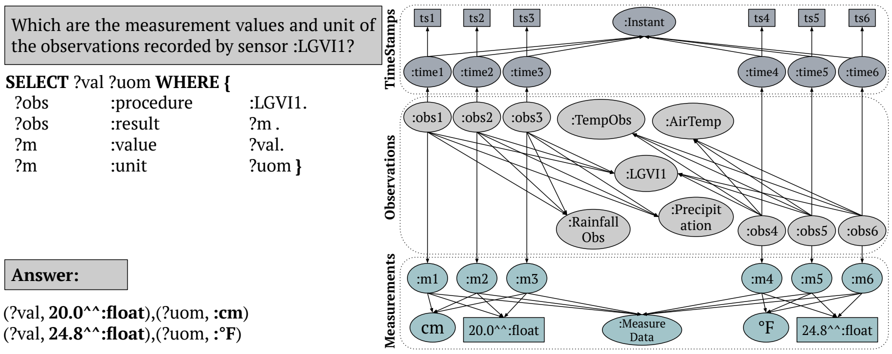
substitutions. Similarly, the head of the rule *fssn3* consists of a triple pattern that matches the observed property of the observations in the original RDF graph, and the body of the rule contains the triple patterns against factorized RDF graphs that match the observed property of the surrogate observations and associate the surrogate and the original observations. Also, the variable substitution for the observation variables are maintained in the body of the rule *fssn3*.

The rules *fssn4*, *fssn5*, and *fssn6* are used to rewrite the triple patterns involving the measurement properties. The head of the rule *fssn4* contains a triple pattern to all the entities of measurements in original RDF graphs are matched. The body of the rule *fssn4* contains three triple patterns that match to the surrogate measurements, as well as, associate the original observations with the surrogate observations, using property *:instanceOf*, and relate the original observations to the corresponding original measurements using property *:result*. Moreover, the body of the rule maintains the measurement variable substitutions, i.e., $?m$ is replaced by $?Xm$ in the query clauses SELECT, FILTER, ORDER BY etc. The head of the rule *fssn5* find matches of the values of measurements in original RDF graphs, whereas the body of the rule find values of the surrogate measurements in factorized RDF graphs. Further, the triple patterns extract associations between the original and surrogate observations, as well as between the original observations and corresponding original measurements, and maintain the measurement variable substitutions. The head of rule *fssn6* contains the triple pattern matching the measurement units in original RDF graph, whereas the body matches the unit of the surrogate measurements. Also, body maintains associations between original and surrogate observations and original observations and corresponding measurements along with measurement variable substitutions. Finally, the head of the rule *fssn7* maps the original observations and the measurements in original RDF graphs using property *fssn7*. The body of the rule *fssn7* find associations between surrogate observations and surrogate measurements in factorized RDF graphs using property *:result*. Likewise associations between the original and surrogate observations and the original observations and original measurements are maintained. Additionally, the variable substitutions for the observations and measurements are maintained.

Definition 5.2.10 (The Query Evaluation Problem). *Let G and G' be RDF graphs such that G' is a factorized graph of G. Consider a SPARQL query Q over G. The problem of evaluating SPARQL queries against a factorized RDF graph corresponds to the problem of transforming Q into a SPARQL query Q' over G' such that the results of evaluating Q over G and the results of Q' over G' are the same, i.e., the following condition is satisfied:*

$$[[Q]]_G = [[Q']]_{G'}$$

Example 5.2.9. *Figure 5.7 shows an instance of the problem of evaluating queries*



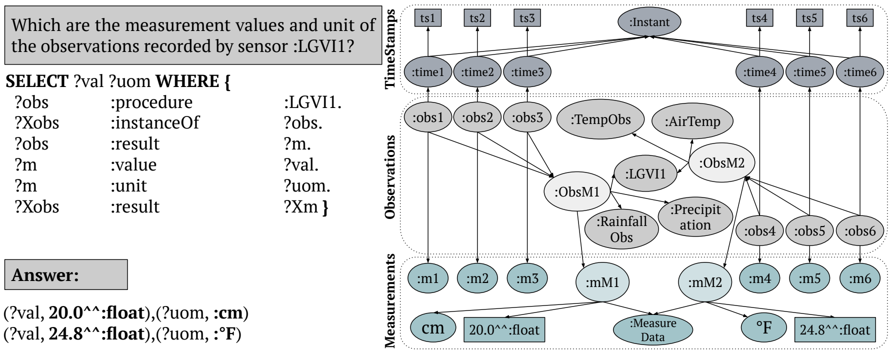
Which are the measurement values and unit of the observations recorded by sensor :LGVI1?

```
SELECT ?val ?uom WHERE {
 ?obs        :procedure        :LGVI1.
 ?obs        :result           ?m .
 ?m          :value            ?val.
 ?m          :unit             ?uom }
```

Answer:

(?val, **20.0^^:float**),(?uom, **:cm**)
(?val, **24.8^^:float**),(?uom, **:°F**)

(a) SPARQL Query over Original RDF Graph



Which are the measurement values and unit of the observations recorded by sensor :LGVI1?

```
SELECT ?val ?uom WHERE {
 ?obs        :procedure        :LGVI1.
 ?Xobs       :instanceOf       ?obs.
 ?obs        :result           ?m.
 ?m          :value            ?val.
 ?m          :unit             ?uom.
 ?Xobs       :result           ?Xm }
```

Answer:

(?val, **20.0^^:float**),(?uom, **:cm**)
(?val, **24.8^^:float**),(?uom, **:°F**)

(b) SPARQLQuery over Factorized RDF Graph

Figure 5.7: **Instance of the Query Evaluation Problem**. Evaluation of SPARQL queries over the original and factorized RDF graphs respects set semantics, i.e., answers are duplicated-free. (a) SPARQL query over the RDF graph in Figure 5.3a selects the measurement values and units, i.e., $(24.8, {}^\circ F)$ and $(20.0, : cm)$ of the temperature and rainfall observations, respectively, collected by the sensor *:LGVI1*; (b) SPARQL query over factorized graph, shown in Figure 5.6b, of the RDF graph in Figure 5.3a where multiplicity of observations and measurement nodes is one; original query is rewritten to produce equivalent results over the factorized graph.

on factorized RDF graphs. A SPARQL query Q over the RDF graph G in Figure 5.3a is presented in Figure 5.7a. The SPARQL query Q' in Figure 5.7b, corresponds to a rewriting of Q, against G' which represents factorization of G. The evaluations of Q and Q' produce the same answers. In this chapter, we present SPARQL query rewriting rules that allow for rewriting a query Q over an original RDF graph into a query Q' over a corresponding factorized RDF graph.

5.2.2 A Factorization Approach

We present a solution to the problem of factorizing RDF graphs describing semantic sensor data. A sketch of the proposed factorization approach is presented in Algorithm 4. The algorithm receives an RDF graph $G(V_G, E_G, L_G)$ and generates a factorized RDF graph $G'(V_{G'}, E_{G'}, L_{G'})$, and the entity mappings μ_N from the observations and measurements in $G(V_G, E_G, L_G)$ to the surrogate observations and measurements in $G'(V_{G'}, E_{G'}, L_{G'})$, respectively. The algorithm initializes the sets of the entity mappings μ_N, the set of nodes $V_{G'}$ and the set of edges $E_{G'}$ of the factorized graph $G'(V_{G'}, E_{G'}, L_{G'})$ (line 1). For all the measurements with value *val* and the corresponding unit of measurement *uom* in $G(V_G, E_G, L_G)$, the algorithm (lines 2-3) creates a surrogate measurement for the corresponding compact measurement molecule in $G'(V_{G'}, E_{G'}, L_{G'})$, i.e., the subject of a compact measurement molecule is created. In lines 4-5, the algorithm maps all the measurements, related to *val* and *uom* in $G(V_G, E_G, L_G)$, to the surrogate measurements in μ_N . For all the observations with observed phenomenon *ph*, sensor procedure *proc*, observed property *pp*, measurement value *val* and unit of measurement *uom* in $G(V_G, E_G, L_G)$, the algorithm creates a surrogate observation representing a corresponding compact observation molecule in $G'(V_{G'}, E_{G'}, L_{G'})$ (lines 6-7), and adds in μ_N the mappings of all the observations in $G(V_G, E_G, L_G)$ with the surrogate observations in $G'(V_{G'}, E_{G'}, L_{G'})$ in lines 8-9. Once all the mappings of the measurements and observations in $G(V_G, E_G, L_G)$ to the corresponding surrogate measurements and observations, respectively, in $G'(V_{G'}, E_{G'}, L_{G'})$ are in μ_N, the factorized graph $G'(V_{G'}, E_{G'}, L_{G'})$ is created using μ_N (lines 10-22). All nodes s and o related with the property `:result` in $G(V_G, E_G, L_G)$ are added to $G'(V_{G'}, E_{G'}, L_{G'})$ along with their associations. Moreover, a new edge relating s and $\mu_N(s)$ using the property `:instanceOf` is added to $G'(V_{G'}, E_{G'}, L_{G'})$ (lines 11-13). If s and o are linked using a property `rdf:type` and o is either `:Observation` or `:MeasureData`, then an new edge $(\mu_N(s)\ p\ o)$ is added to $G'(V_{G'}, E_{G'}, L_{G'})$ along with $\mu_N(s)$ and o (lines 14-16). If s and o are associated through a predicate p in {:procedure, :property, :value, :unit}, then a new edge $(\mu_N(s)\ p\ o)$ is added to $G'(V_{G'}, E_{G'}, L_{G'})$ in lines 17-19. Otherwise, the edge $(s\ p\ o)$ is added to the $G'(V_{G'}, E_{G'}, L_{G'})$ in lines 20-22.

Algorithm 4 The Factorization Algorithm

```
Input: An RDF graph $G(V_G, E_G, L_G)$
Output: Factorized RDF Graph $G'(V_{G'}, E_{G'}, L_{G'})$, and entity mappings $\mu_N$
1: $\mu_N \longleftarrow \emptyset, V_{G'} \longleftarrow \emptyset, E_{G'} \longleftarrow \emptyset, L_{G'} \longleftarrow \emptyset$
2: for $val, uom \in V_G$ such that $SM = \{m | (m$ rdf:type :MeasureData$) \in G, (m$ :unit $uom) \in G, (m$ :value $val) \in G\}$ do
3:     $mM \leftarrow SurrogateMeasurement()$
```

```
4: end for
5: for $m \in SM$ do
6:   $\mu_N \leftarrow \mu_N \cup \{(m, mM)\}$
7:   for $proc, ph, pp, val, uom \in V_G$ such that $SO = \{obs | (obs$ rdf:type $ph) \in G$, $(obs$ :procedure $proc) \in G$, $(obs$ :property $pp) \in G$, $(obs$ :result $m) \in G$, $(m$ rdf:type :MeasureData$) \in G$, $(m$ :unit $uom) \in G$, $(m$ :value $val) \in G\}$ do
8:     $oM \leftarrow SurrogateObservation()$
9:     for $obs \in SO$ do
10:      $\mu_N \leftarrow \mu_N \cup \{(obs, oM)\}$
11:    end for
12:  end for
13: end for
14: for $(s\ p\ o) \in E_G \wedge s, o \in V_G \wedge p \in L_G$ do
15:   if $p ==$ :result then
16:     $E_{G'} \leftarrow E_H \cup \{(s\ p\ o), (\mu_N(s)\ p\ \mu_N(o)), (s$ :instanceOf $\mu_N(s))\}$
17:     $V_{G'} \leftarrow V_{G'} \cup \{s, o, \mu_N(s), \mu_N(o)\}$
18:     $L_{G'} \leftarrow L_{G'} \cup \{p,$ :instanceOf$\}$
19:   else if $p ==$ rdf:type && $(o ==$ :Observation$||o ==$ :MeasureData$)$ then
20:     $E_{G'} \leftarrow E_{G'} \cup \{(\mu_N(s)\ p\ o)\}$
21:     $V_{G'} \leftarrow V_{G'} \cup \{\mu_N(s), o\}$
22:     $L_{G'} \leftarrow L_{G'} \cup \{p\}$
23:   else if $p ==$ :procedure$||p ==$ :property$||p ==$ :value$||p ==$ :unit then
24:     $E_{G'} \leftarrow E_{G'} \cup \{(\mu_N(s)\ p\ o)\}$
25:     $V_{G'} \leftarrow V_{G'} \cup \{\mu_N(s), o\}$
26:     $L_{G'} \leftarrow L_{G'} \cup \{p\}$
27:   else
28:     $E_{G'} \leftarrow E_{G'} \cup \{(s\ p\ o)\}$, $V_{G'} \leftarrow V_{G'} \cup \{s, o\}$, $L_{G'} \leftarrow L_{G'} \cup \{p\}$
29:   end if
30: end for
31: return $G'(V_{G'}, E_{G'}, L_{G'}), \mu_N$
```

Figure 5.9b depicts a portion of the RDF in Figure 5.3a and the corresponding transformation in the factorized RDF graph in Figure 5.6b. The surrogate measurements and observations, and the new edges are highlighted in bold. The Algorithm 4 creates the surrogate measurements and observations in line 3 and in line 7; new edges are created in line 12, 15 and 18. Additionally, assumptions about the characteristics of the associations between the nodes in the graph are presented. While some edges existing in the RDF graph in Figure 5.3a are not present in the factorized RDF graph, these associations can be obtained by traversing the graph through the surrogate observations and measurements. The implicit satisfaction of all the associations in the original RDF graph that are not included in the factorized graph is restricted under the following assumptions:

For all resources `:obs` and `:m` corresponding to an observation and a measurement, respectively, in $G(V_G, E_G, L_G)$, i.e., `:obs` is of type `:Observation` and `:m` is of type `:MeasureData`, the following properties hold.

- **Measurement**: `:m` is only associated with one value and with one unit of measurement, i.e., the properties `:value` and `:unit` that relate a measurement with a value and a measurement unit are both functional properties for any measurement `:m`. Furthermore, `:m` is related with only one observation, i.e., the property `:result` that associates an observation with a measurement has a functional inverse.
- **Observation**: `:obs` is only associated with one procedure, i.e., the property `:procedure` that associates an observation and a procedure is a functional property for any observation `:obs`.

For all resources `:obsM`, `:mM`, `:obs`, and `:m` corresponding to a surrogate observation, a surrogate measurement, an observation, and a measurement, respectively, in $G'(V_{G'}, E_{G'}, L_{G'})$, the following properties hold.

- **Surrogate Observation**: `:obsM` is only associated with one procedure and one observed property, i.e., the properties `:procedure` and `:property` that associate a surrogate observation with a procedure and a property, respectively, are functional properties for any surrogate observation `:obsM`.
- **Surrogate Measurement**: `:mM` is only associated with one value and with one unit of measurement, i.e., the properties `:value` and `:unit` that relate a surrogate measurement with a value and a measurement unit are both functional properties for any surrogate measurement `:mM`. Furthermore, `:mM` is related with only one surrogate observation, i.e., `:result` that associates a surrogate observation with a surrogate measurement has a functional inverse.
- **Observation**: `:obs` is only associated with one surrogate observation, i.e., `:instanceOf` property that associates an observation with a surrogate observation is a functional property.
- **Measurement**: `:m` is related with only one observation, i.e., `:result` associates an observation with a measurement, and has a functional inverse.

We are assuming that SPARQL queries against the original and factorized RDF graphs are evaluated under the set semantics, i.e., no duplicates are in the answers. Coming back to the motivating example, Figure 5.8 illustrates the factorized RDF graph of the graph in Figure 5.2. The factorized RDF graph in Figure 5.8a is sparse and the average number of neighbors has been reduced from 6.361 to 2.568. These statistics of the factorized RDF graph indicate that the number of RDF triples describing an observation is reduced after factorization. Figure 5.8c shows the number of RDF triples per measurement value in the factorized RDF graph with respect to the original RDF graph. For each measurement value the number of associated RDF triples in the factorized RDF graph is reduced by 74%.

5.2.3 Queries over Factorized RDF Graphs

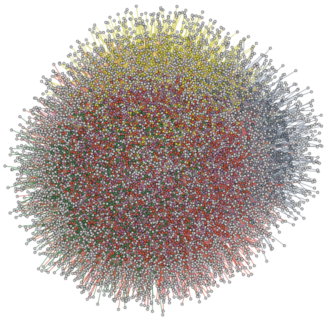
In this section, we define the algorithm that solves the problem of query evaluation on a factorized RDF graph. Table 5.1 presents the rules to rewrite a SPARQL query against an original SSN RDF graph into a query against the corresponding factorized RDF graph. The query rewriting rules are defined in terms of SPARQL triple patterns. For each property of the observation and measurement classes in the SSN ontology, a rewriting rule is defined. Furthermore, substitutions for the variables corresponding to the observations and measurements are presented. These variable substitutions are used during the query rewriting in the query clauses, i.e., SELECT, ORDER BY, GROUP BY, and FILTERS etc. Given a SPARQL query and a set R of query rewriting rules, Algorithm 5 describes the steps performed to each set of triple patterns that composes a Basic Graph Pattern (BGP). If the input query consists of several BGPs, the structure of the original query remains the same, and the algorithm is applied to each BGP using the query rewriting rules in Table 5.1.

Algorithm 5 The Query Rewriting Algorithm

```
Input: Set $ST$ of triple patterns in a BGP of $Q$ and set $SR$ of query rewriting rules
Output: $ST_{new}$ the rewriting of $ST$ under $SR$
$ST_{new} \longleftarrow \emptyset$
for $t \in ST$ do
    Select $r \in SR$ such that $t$ matches the head of $r$ and instantiate the body of $r$
    Let $SQ_t$ be the matched body of $r$ and $variableSubstitutions$ be the set of mappings between variables in $t$ into variables in $SQ_t$, add $(t, SQ_t, variableSubstitutions)$ to $ST_{new}$
```

---

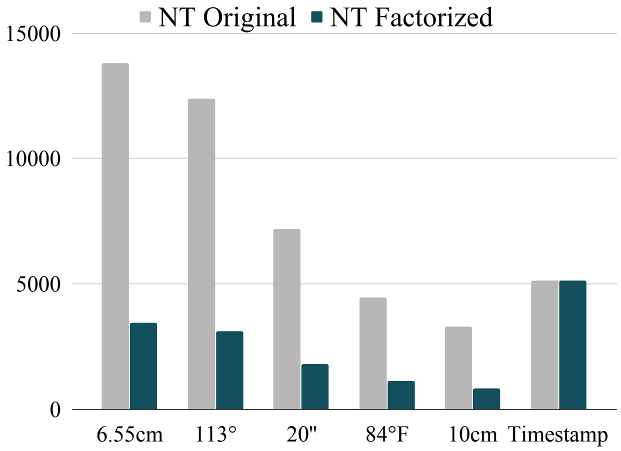
[4] `http://www.cytoscape.org/`



(a) Factorized **RDF Graph**

S#	Parameter	Value
1	Connected Components	1.000
2	Network Centralization	0.143
3	**Avg. # of Neighbors**	**2.568**
4	Network Density	0.000
5	Multi-edge Node Pairs	5.000
6	Network Heterogeneity	9.186

(b) **Statistics** of Factorized RDF Graph

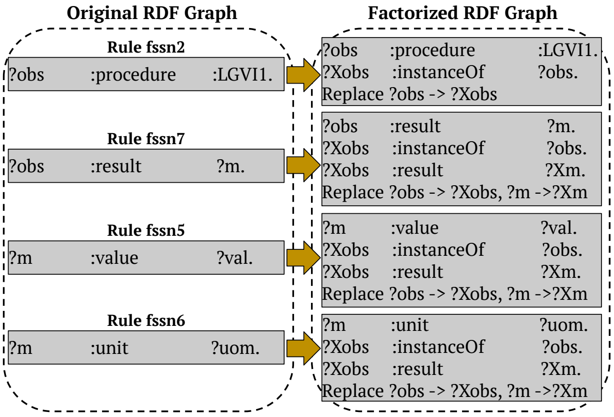


(c) **NT** Factorized vs Original

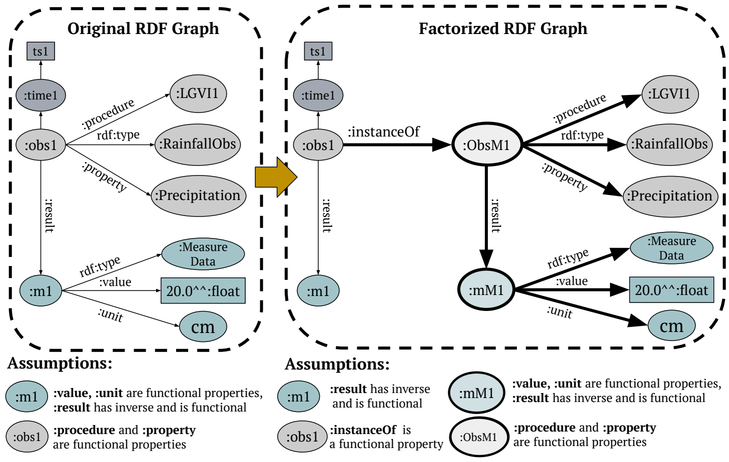
Figure 5.8: **Factorization of the Running Example.** The number of RDF triples related to the same value is reduced in an RDF graph after factorization of the RDF graph. (a) The factorized RDF Graph representing the sensor observations from Figure 5.2a contains a smaller number of edges; (b) Statistics of the factorized RDF graph show that the average number of neighbours of a node are reduced, as well as the multi-edge node pairs while keeping the sensor observations connected in the RDF graph; (c) The number of RDF triples associated with each distinct value is considerable reduced in the factorized RDF graph. Timestamps are not factorized, therefore, the number of RDF triples describing `Timestamp` remains the same in both, original and factorized, RDF graphs. The RDF graph in Figure 5.8a and statistics in Table 5.8b are generated by the `Cytoscape tool`[4].

```
end for
return $ST_{new}$
```

Figure 5.7 presents two SPARQL queries: Figures 5.7a and 5.7b present an



(a) Query Rewriting



(b) Original and Factorized RDF Graphs

Figure 5.9: **Example of Query Rewriting**. Rewriting rules for a query over the original RDF graphs are presented. (a) Query rewriting rules *fssn2*, *fssn5*, *fssn6*, and *fssn7* from Table 5.1 are used to rewrite the SPARQL query in Figure 5.7a into the SPARQL query in Figure 5.7b. The variables ?*obs* and ?*m* in the rewritten query correspond to the variables representing surrogate observation and measurement, respectively, in the factorized RDF graph. In order to retrieve the original observations and measurements, the variables ?*obs* and ?*m* are replaced with $?Xobs$ and $?Xm$ in the query clauses of the the rewritten SPARQL query. Associations explicitly represented in the original RDF graph are modeled with triple pattern paths in the factorized graph. (b) Portions of the RDF graphs (original and factorized). Nodes and edges highlighted in bold are added during the creation of the factorized graph. Assumptions enforce that the factorized graph represents all relations represented in the original graph.

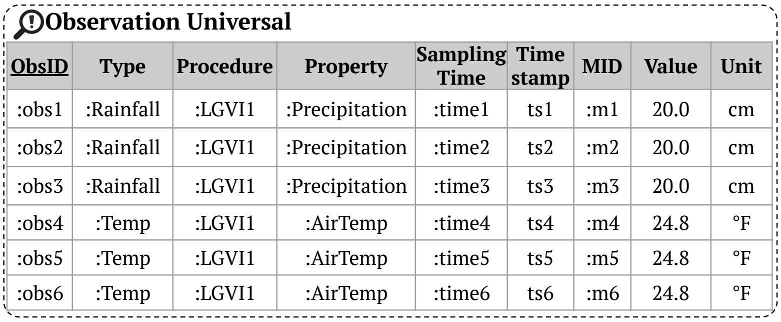
original query Q and rewriting of Q produced by Algorithm 5. Figure 5.9a presents the rewriting of SPARQL query in Figure 5.7a. Rules *fssn2*, *fssn5*, *fssn6*, and *fssn7* from Table 5.1 are used to rewrite the query. The algorithm replaces each triple pattern in a BGP that instantiates the head of a rule in SR by the body of the rule, e.g., the triple pattern (*?obs :procedure :LGVI1*) instantiates the head of rule *fssn2*, thus, the triple pattern in the BGP is replaced with the body of *fssn2*, as shown in Figure 5.9a. Moreover, the variables corresponding to the observations and measurements in the original query represent the surrogate observations and measurements in the rewritten query, consequently, these variables are replaced by the new variables in the query clauses to refer the original observations and measurements. The variable substitution for observation *?obs* by $?Xobs$ is maintained during the rewriting process using rule *fssn2* in order to retrieve the original observations, if required. Similarly, other triple patterns in the BGP each matching the head of a rule, i.e., *fssn5*, *fssn6*, and *fssn7*, are replaced by the body of the rule, and the variable substitutions of *?obs* and $?m$ by $?Xobs$ and $?Xm$, respectively, are maintained for the query clauses. The evaluation of both, original and rewritten, queries produce the same results. Another important property is that the time complexity of the original and rewritten queries is also the same.

Theorem 5.2.1. *Given G and G' such that G' is a factorized RDF graph of G. Let Q and Q' be SPARQL queries where Q' is a rewritten query of Q over G' generated by Algorithm 5. The problem of evaluating Q' against G' is in: (1) PTIME if query Q has only AND and FILTER operators; (2) NP-complete if query Q has expressions with AND, FILTER, and UNION operators; and (3) PSPACE-complete for OPTIONAL graph pattern expressions.*

Proof. We proceed with a proof by contradiction. Assume that complexity of Q' is higher than Q. Then, UNION or OPTIONAL operators not included in Q are added to Q'. However, Algorithm 5 only changes triple patterns over G by triple patterns against G'. Additionally, Algorithm 5 includes new JOINs (AND operator). However, adding AND or FILTER operators does not affect the complexity of the problem of evaluating Q' over G', and contradicting the fact that the complexity of Q' is higher than Q. $\square$

5.3 Tabular Representation of RDF Graphs

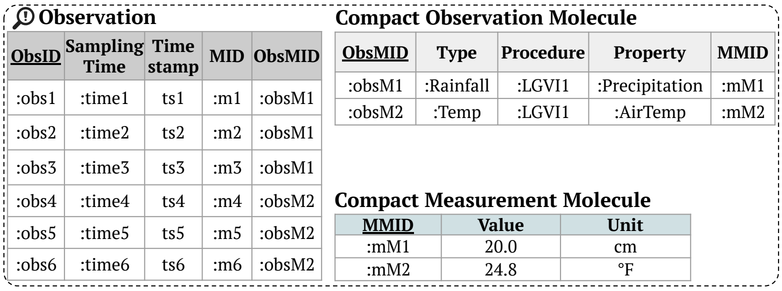
Sensor data tend to stack up quickly, scaling up to large amounts of data. In order to capture that growth, we opt for representing factorized data in tabular format, so that Big Data processing technologies can be used. For that purpose, we



Observation Universal

ObsID	Type	Procedure	Property	Sampling Time	Time stamp	MID	Value	Unit
:obs1	:Rainfall	:LGVI1	:Precipitation	:time1	ts1	:m1	20.0	cm
:obs2	:Rainfall	:LGVI1	:Precipitation	:time2	ts2	:m2	20.0	cm
:obs3	:Rainfall	:LGVI1	:Precipitation	:time3	ts3	:m3	20.0	cm
:obs4	:Temp	:LGVI1	:AirTemp	:time4	ts4	:m4	24.8	°F
:obs5	:Temp	:LGVI1	:AirTemp	:time5	ts5	:m5	24.8	°F
:obs6	:Temp	:LGVI1	:AirTemp	:time6	ts6	:m6	24.8	°F

(a) Universal Parquet Table for Observations



Observation

ObsID	Sampling Time	Time stamp	MID	ObsMID
:obs1	:time1	ts1	:m1	:obsM1
:obs2	:time2	ts2	:m2	:obsM1
:obs3	:time3	ts3	:m3	:obsM1
:obs4	:time4	ts4	:m4	:obsM2
:obs5	:time5	ts5	:m5	:obsM2
:obs6	:time6	ts6	:m6	:obsM2

Compact Observation Molecule

ObsMID	Type	Procedure	Property	MMID
:obsM1	:Rainfall	:LGVI1	:Precipitation	:mM1
:obsM2	:Temp	:LGVI1	:AirTemp	:mM2

Compact Measurement Molecule

MMID	Value	Unit
:mM1	20.0	cm
:mM2	24.8	°F

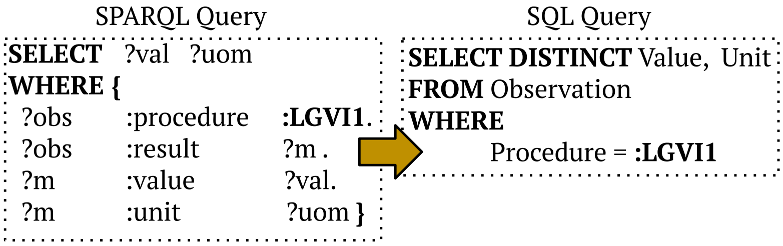
(b) Factorized Data Parquet Tables

Figure 5.10: **Factorized Tabular Representation of RDF Graphs**. Parquet tables are used to represent RDF graphs in Spark (RDF Graphs in Figures 5.3a and 5.6b). (a) A universal table stores all the data of the original graph; parquet table columnar storage efficiently stores duplicated data.(b) Three parquet tables are used to store factorized data comprising compact observation and measurement molecules. Primary keys are denoted by underlined attribute names. A foreign key is represented by repeating the name of a primary key in another table.

choose to store the data in the modern, columnar-oriented *Parquet*[5] storage format. We propose tabular representations of both the original and factorized RDF graphs (in Figure 5.3a and Figure 5.6b, respectively), shown in Figure 5.10 and Figure 5.12. Parquet uses Run-Length Encoding (RL), whereby repeated numerical values are encoded into pairs of the value and its occurrences number which allows an efficient representation of large datasets of semantic sensor data. Moreover, columnar nature of Parquet makes it best suited for scenarios where queries access only a few number of columns from a *wide* table of many columns. Parquet pulls only the requested columns, contrary to row-oriented storage, where the whole row is read even if only few columns are requested. We rely on these properties of Parquet tables, and represent RDF graphs using a *universal* table. The universal

---

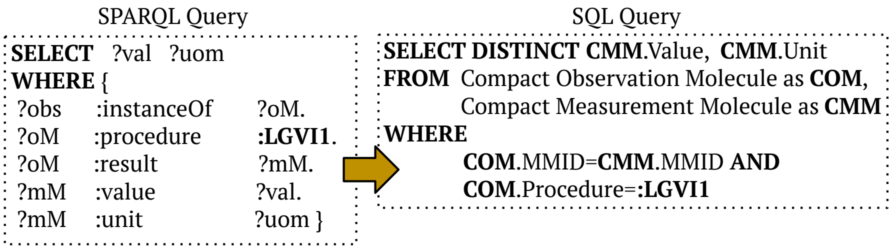
[5] `https://parquet.apache.org/`



```
SELECT  ?val  ?uom
WHERE {
 ?obs     :procedure    :LGVI1.
 ?obs     :result       ?m .
 ?m       :value        ?val.
 ?m       :unit         ?uom }
```

```
SELECT DISTINCT Value, Unit
FROM Observation
WHERE
        Procedure = :LGVI1
```

(a) Query Universal Table



```
SELECT  ?val  ?uom
WHERE {
 ?obs    :instanceOf    ?oM.
 ?oM     :procedure     :LGVI1.
 ?oM     :result        ?mM.
 ?mM     :value         ?val.
 ?mM     :unit          ?uom }
```

```
SELECT DISTINCT CMM.Value, CMM.Unit
FROM  Compact Observation Molecule as COM,
      Compact Measurement Molecule as CMM
WHERE
        COM.MMID=CMM.MMID AND
        COM.Procedure=:LGVI1
```

(b) Query Factorized Data Tables

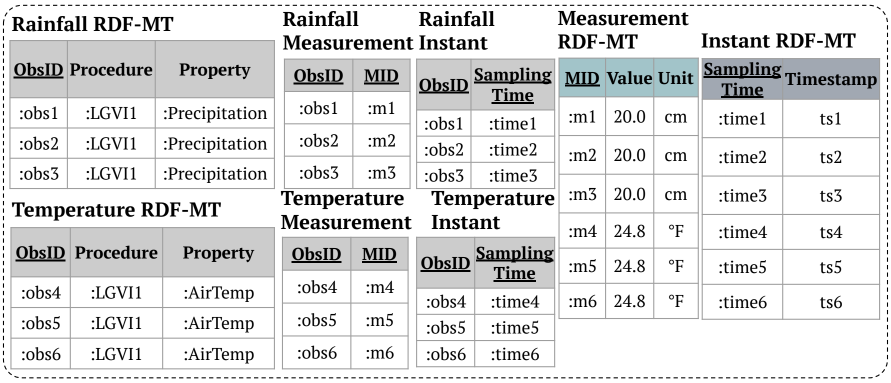
Figure 5.11: **Query Evaluation Over Universal and Factorized Tables**. SPARQL queries over original and factorized RDF graphs and their corresponding SQL queries against the universal and factorized tables are presented. (a) SQL query over the universal parquet table; only a selection and the distinct modifier are required to collect values and unit of measurement. (b) SQL query against the parquet tables representing the factorized RDF graph; one join, one filter, one selection, and the distinct modifier are needed to express the SPARQL query.

tabular representation, `Observation Universal` in Figure 5.10a, of original RDF graph in Figure 5.3a, contains the properties that directly or indirectly describe an observation. For example, in Figure 5.3a, an observation is directly described by the properties `rdf:type`, `:procedure`, `:property`, `:result`, and `:samplingTime`; while `:value`, `:unit`, and `:timestamp` indirectly describe an observation. In the universal table these predicates are modeled with the attributes: `Type`, `Procedure`, `property`, `MID`, `SamplingTime`, `Value`, `Unit`, and `Timestamp`, respectively.

The tabular representation of the factorized RDF graph in Figure 5.6b is shown in Figure 5.10b. The `Compact Observation Molecule` table contains the properties that describe a surrogate observation in the factorized RDF graph. For example, in Figure 5.6b, the properties `rdf:type`, `:procedure`, `:property`, and `:result` describe a surrogate observation and are modeled in the `Compact Observation Molecule` table with the attributes `Type`, `Procedure`, `Property`, and `MMID`, respectively. The `Compact Measurement Molecule` table contains the properties describing a surrogate measurement using value and unit in the factorized RDF graph. Thus, in Figure 5.10b, the `Compact Measurement Molecule` table is populated with the values of the properties `:value` and `:unit` of the surrogate measurements in the factorized RDF graph in Figure 5.6b. Note that the type `:MeasureData` is not included as an attribute in the table, because this is implicitly represented in the table name. The `Observation` factorized table contains the observation predicates that are not represented in the `Compact Observation Molecule` and `Compact Measurement Molecule` tables, as well as a reference to the corresponding surrogate observations, as a foreign key. For example, the predicates `:result`, `:samplingTime`, and `:timestamp` are not included in `Compact Observation Molecule` and `Compact Measurement Molecule` tables, but they are represented in the `Observation` table by the attributes `MID`, `SamplingTime`, and `Timestamp`, respectively. Moreover, an association between an observation and corresponding surrogate observation, described using the property `:instanceOf` in the factorized RDF graph, is represented by the foreign key `ObsMID`. Moreover, SPARQL queries against original and factorized graphs are translated into SQL queries over the universal and factorized tables, respectively. Figure 5.11 illustrates the SQL representations of SPARQL queries in Figure 5.7. In all cases, the results of the original and factorized SQL queries are the same as the SPARQL queries over the original and factorized RDF graphs.

Instead of using the universal tabular representations, RDF graphs can be represented using RDF molecule template (RDF-MT) based tabular representations. RDF-MTs describe RDF data sources using abstract representations of the classes in RDF graphs and their properties. Furthermore, RDF-MTs provide links between the classes belonging to the same RDF graphs or to different RDF graphs using intra- and inter-linking, respectively. For each RDF-MT around a class one table is created containing the attributes of the class as columns. Similarly, for each intra- or inter-link between the classes a binary table is created containing the identifiers from the corresponding classes as attributes. The web service APIs, i.e., SPARQL endpoints, collect RDF data around each RDF molecule template.

Figure 5.12a illustrates the RDF molecule template (RDF-MT) based tabular representations of the RDF graph in Figure 5.3a. The RDF graph contains four RDF molecule templates, i.e., `:RainfallObs`, `:TempObs`, `:Instant`, and `:MeasureData`. The rainfall and temperature observations are directly described using the properties `:procedure` and `:property`, and are modeled with the attributes `Procedure` and `Property`, respectively, in the RDF-MT based tabular representations `Rainfall RDF-MT` and `Temperature RDF-MT`, respectively. Similarly, RDF molecule template around `:MeasureData` is described using the properties `:value` and `:unit`, and are modeled with the attributes `Value` and `Unit`, respectively, in the tabular representation `Measurement RDF-MT`. The RDF molecule template of `:Instant` is described using the property `:timestamp` and is represented using the attribute `Timestamp` in the RDF-MT based table `Instant`



Rainfall RDF-MT

ObsID	Procedure	Property
:obs1	:LGVI1	:Precipitation
:obs2	:LGVI1	:Precipitation
:obs3	:LGVI1	:Precipitation

Temperature RDF-MT

ObsID	Procedure	Property
:obs4	:LGVI1	:AirTemp
:obs5	:LGVI1	:AirTemp
:obs6	:LGVI1	:AirTemp

Rainfall Measurement

ObsID	MID
:obs1	:m1
:obs2	:m2
:obs3	:m3

Temperature Measurement

ObsID	MID
:obs4	:m4
:obs5	:m5
:obs6	:m6

Rainfall Instant

ObsID	Sampling Time
:obs1	:time1
:obs2	:time2
:obs3	:time3

Temperature Instant

ObsID	Sampling Time
:obs4	:time4
:obs5	:time5
:obs6	:time6

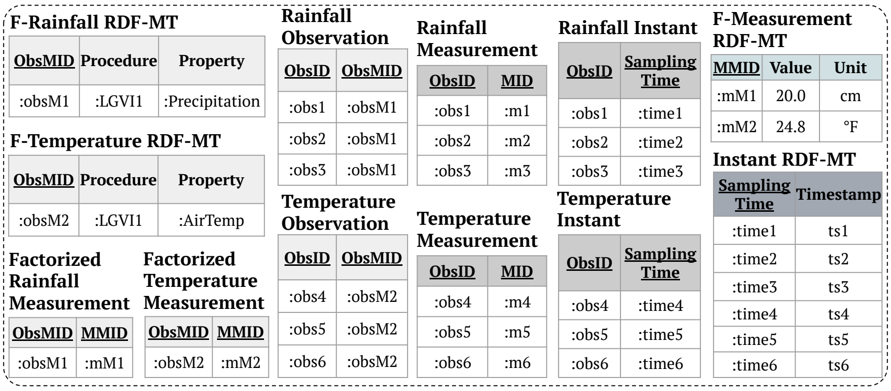
Measurement RDF-MT

MID	Value	Unit
:m1	20.0	cm
:m2	20.0	cm
:m3	20.0	cm
:m4	24.8	°F
:m5	24.8	°F
:m6	24.8	°F

Instant RDF-MT

Sampling Time	Timestamp
:time1	ts1
:time2	ts2
:time3	ts3
:time4	ts4
:time5	ts5
:time6	ts6

(a) RDF-MT based Table for Observations



F-Rainfall RDF-MT

ObsMID	Procedure	Property
:obsM1	:LGVI1	:Precipitation

F-Temperature RDF-MT

ObsMID	Procedure	Property
:obsM2	:LGVI1	:AirTemp

Factorized Rainfall Measurement

ObsMID	MMID
:obsM1	:mM1

Factorized Temperature Measurement

ObsMID	MMID
:obsM2	:mM2

Rainfall Observation

ObsID	ObsMID
:obs1	:obsM1
:obs2	:obsM1
:obs3	:obsM1

Temperature Observation

ObsID	ObsMID
:obs4	:obsM2
:obs5	:obsM2
:obs6	:obsM2

Rainfall Measurement

ObsID	MID
:obs1	:m1
:obs2	:m2
:obs3	:m3

Temperature Measurement

ObsID	MID
:obs4	:m4
:obs5	:m5
:obs6	:m6

Rainfall Instant

ObsID	Sampling Time
:obs1	:time1
:obs2	:time2
:obs3	:time3

Temperature Instant

ObsID	Sampling Time
:obs4	:time4
:obs5	:time5
:obs6	:time6

F-Measurement RDF-MT

MMID	Value	Unit
:mM1	20.0	cm
:mM2	24.8	°F

Instant RDF-MT

Sampling Time	Timestamp
:time1	ts1
:time2	ts2
:time3	ts3
:time4	ts4
:time5	ts5
:time6	ts6

(b) Factorized RDF-MT based Tables

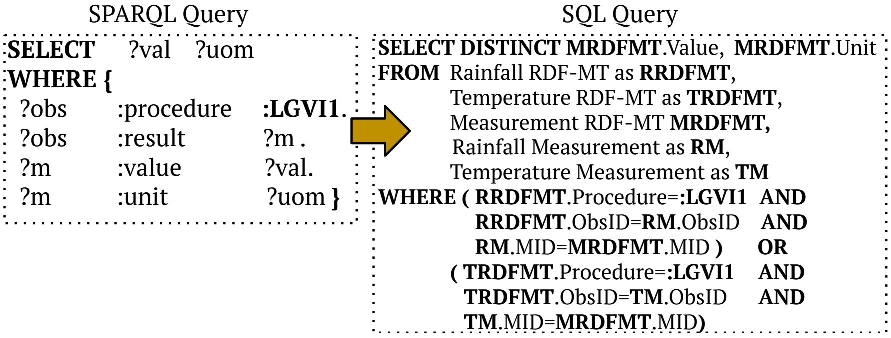
Figure 5.12: **RDF-MT based Tabular Representation of RDF Graphs**. Parquet tables are utilized to represent RDF graphs in Spark (RDF Graphs in Figures 5.3a and 5.6b). (a) Each RDF-MT based table stores an RDF molecule template in the original RDF graph; parquet table columnar storage efficiently stores duplicated data.(b) Factorized RDF graph is represented in compact RDF-MT based tables. Primary keys are denoted by underlined attribute names. A foreign key is represented by repeating the name of a primary key in another table.

`RDF-MT`. The associations of the `:RainfallObs` and `:TempObs` RDF molecule templates with the `:MeasureData` and `:Instant` RDF molecule templates are shown in Figure 5.3a. In Figure 5.12a, `Rainfall Measurement` models the association between the `:RainfallObs` and `:MeasureData` RDF molecule templates using the primary keys, `ObsID` and `MID`, from the RDF-MT based tabular representations `Rainfall RDF-MT` and `Measurement RDF-MT`, respectively. Also, the association between the `:RainfallObs` and `:Instant` RDF molecule templates is modeled with the tabular representation `Rainfall Instant` using the primary keys, `ObsID` and `SamplingTime`, from the tabular representations `Rainfall RDF-MT` and

`Instant RDF-MT`, respectively. Similarly, the association between the `:TempObs` and `:MeasureData` RDF molecule templates is modeled in the tabular representation `Temperature Measurement` using the primary keys, `ObsID` and `MID`, from the tables `Temperature RDF-MT` and `Measurement RDF-MT`, respectively. Likewise, the link between the `:TempObs` and `:Instant` RDF molecule templates is represented in `Temperature Instant` using the primary keys, `ObsID` and `SamplingTime`, from the tabular representations `Temperature RDF-MT` and `Instant RDF-MT`, respectively, describing temperature and timestamps RDF molecule templates.

Factorization of RDF graphs also generates the factorized RDF-MT based tabular representations. The RDF-MT based tabular representations of the factorized RDF graph in Figure 5.6b are shown in Figure 5.12b. The tabular representation `F-Rainfall RDF-MT` models the properties `:procedure` and `:property`, describing the RDF molecule templates around the surrogate rainfall observations in the factorized RDF graph, with the attributes `Procedure` and `Property`, respectively. Similarly, the RDF molecule templates of the surrogate temperature observations are modeled in the RDF-MT based tabular representations `F-Temperature RDF-MT` with attributes `Procedure` and `Property` corresponding to the temperature properties `:procedure` and `:property`, respectively. The `:value` and `:unit` properties, describing the surrogate measurements, are modeled in the `F-Measurement RDF-MT` tabular representation using the attributes `Value` and `Unit`, respectively. RDF-MT based tabular representations `Instant RDF-MT` contains `:timestamp` property, describing the timestamps, and is modeled with `Timestamp` attribute.

The associations between the RDF molecule templates of the surrogate observations and measurements are represented in `Factorized Rainfall Measurement` and `Factorized Temperature Measurement` for the rainfall and temperature observations, respectively, with the primary keys from the corresponding RDF-MT based tabular representations. Moreover, the explicit mappings between the original rainfall observations and the surrogate rainfall observations described using the property `:instanceOf` are represented in the table `Rainfall Observation` with attributes `ObsID` and `ObsMID` corresponding to the original and surrogate observations, respectively. Similarly, the mappings between the original and surrogate temperature observations are modeled in `Temperature Observation`. In addition, the links between the original rainfall observations and measurements are modeled in tabular representations `Rainfall Measurement` using the attributes `ObsID` and `MID` corresponding the original observations and measurements, respectively. Similarly, the associations between original temperature observations and measurements are presented in `Temperature Measurement`. Furthermore, the links between the original rainfall observations and the timestamps are represented in the table `Rainfall Instant` using the attributes `ObsID` and `SamplingTime`



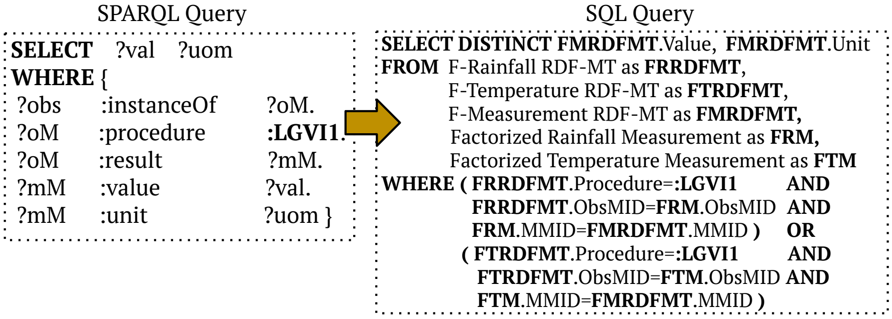
SPARQL Query

```
SELECT    ?val   ?uom
WHERE {
 ?obs      :procedure    :LGVI1.
 ?obs      :result       ?m .
 ?m        :value        ?val.
 ?m        :unit         ?uom }
```

SQL Query

```
SELECT DISTINCT MRDFMT.Value,  MRDFMT.Unit
FROM  Rainfall RDF-MT as RRDFMT,
      Temperature RDF-MT as TRDFMT,
      Measurement RDF-MT MRDFMT,
      Rainfall Measurement as RM,
      Temperature Measurement as TM
WHERE ( RRDFMT.Procedure=:LGVI1  AND
        RRDFMT.ObsID=RM.ObsID   AND
        RM.MID=MRDFMT.MID )     OR
      ( TRDFMT.Procedure=:LGVI1  AND
        TRDFMT.ObsID=TM.ObsID   AND
        TM.MID=MRDFMT.MID)
```

(a) Query RDF-MT based Tables



SPARQL Query

```
SELECT   ?val   ?uom
WHERE {
 ?obs     :instanceOf       ?oM.
 ?oM      :procedure        :LGVI1.
 ?oM      :result           ?mM.
 ?mM      :value            ?val.
 ?mM      :unit             ?uom }
```

SQL Query

```
SELECT DISTINCT FMRDFMT.Value,  FMRDFMT.Unit
FROM  F-Rainfall RDF-MT as FRRDFMT,
      F-Temperature RDF-MT as FTRDFMT,
      F-Measurement RDF-MT as FMRDFMT,
      Factorized Rainfall Measurement as FRM,
      Factorized Temperature Measurement as FTM
WHERE ( FRRDFMT.Procedure=:LGVI1        AND
        FRRDFMT.ObsMID=FRM.ObsMID  AND
        FRM.MMID=FMRDFMT.MMID )    OR
      ( FTRDFMT.Procedure=:LGVI1        AND
        FTRDFMT.ObsMID=FTM.ObsMID AND
        FTM.MMID=FMRDFMT.MMID )
```

(b) Query Factorized RDF-MT based Tables

Figure 5.13: **Query Evaluation Over RDF-MT based Tables**. SPARQL queries over original and factorized RDF graphs and corresponding SQL queries against RDF-MT based tables are presented. (a) An SQL query over RDF-MT based parquet tables of an original RDF graph; four joins, two filters, and a selection and a distinct modifier are required to collect values and units of measurement. (b) An SQL query against RDF-MT based parquet tables of a factorized RDF graph; four joins, two filters and a selection and a distinct modifier are needed to express the rewritten SPARQL query.

corresponding to the original rainfall observations and the primary key from the tabular representations `Instant RDF-MT`, respectively. Similarly, the associations between original temperature observations and the timestamps are modeled in tabular representations `Temperature Instant`. To run the queries over the RDF-MT based tabular representations, SPARQL queries against original and factorized RDF graphs are translated into SQL queries against the relevant RDF-MT based tabular representations. Figure 5.13 illustrates the corresponding RDF-MT based SQL representations of the SPARQL queries in Figure 5.7. The results of the SQL queries against RDF-MT based tabular representations of the original and factorized RDF graphs are the same as the SPARQL queries over the original and factorized RDF graphs.

Theorem 5.3.1. *The decomposition of the* `Observation` *universal table into factorized tables:* `Observation`*,* `Compact Observation Molecule`*, and* `Compact Measurement Molecule`*, is* loss-less join*.*

Proof. Considering that the following functional dependencies hold in both the universal, and factorized tables:

- `ObsMID` → `Type`, `Procedure`, `Property`, `MMID`
- `MMID` → `Value`, `Unit`
- `ObsID` → `SamplingTime`, `Timestamp`, `MID`, `ObsMID`

We can prove using the algorithm[50] that the factorized tables are a *loss-less join* decomposition of universal table T that includes all the attributes in the `Observation` universal plus `ObsMID` and `MMID`. The attributes of the `Observation` universal can be projected from G', thus, satisfying the *loss-less join* condition. ☐

Theorem 5.3.2. *If* G *is an SSN RDF graph and* G' *is a factorized RDF graph of* G*, and* T_1 *is the factorized tabular representation of* G'*, then* T_1 *is in third normal form with respect to the universal representation of* G*.*

Proof. Recall [25], a table is in third normal form if for every $X \rightarrow Y$

- X is a super key, or
- $Y - X$ is a prime attribute

Considering that the following functional dependencies hold in both the universal, and factorized tables:

- `ObsMID` → `Type`, `Procedure`, `Property`, `MMID`
- `MMID` → `Value`, `Unit`

Table 5.2: **Datasets**: Description of the semantically described sensor datasets; weather observations are collected since 2003 from around 20,000 weather stations during the hurricane and blizzard seasons in the United States.

Dataset ID	**Climate Event**	**Date**	**#RDF Triples**	**# Obs**
D1	Blizzard	April, 2003	38,054,493	4,092,492
D2	Hurricane Charley	August, 2004	108,644,568	11,648,607
D3	Hurricane Katrina	August, 2005	179,128,407	19,233,458

- `ObsID` → `SamplingTime`, `Timestamp`, `MID`, `ObsMID`

It can be demonstrated that all the tables created after factorization are in third normal form. ☐

Theorem 5.3.3. *The decomposition of the* `RDF-MT` *based tables representing sensor data into the* `factorized RDF-MT` *based tables is* loss-less join*.*

Proof. Considering that the following functional dependencies hold in both the `RDF-MT`, and `factorized RDF-MT` based tables:

- `ObsMID` → `Procedure`, `Property`
- `MMID` → `Value`, `Unit`
- `ObsMID`, `MMID` → `ObsMID`, `MMID`
- `ObsID`, `ObsMID` → `ObsID`, `ObsMID`
- `ObsID`, `MID` → `ObsID`, `MID`
- `ObsID`, `SamplingTime` → `ObsID`, `SamplingTime`
- `SamplingTime` → `Timestamp`

We can prove using the algorithm[50] that the `factorized RDF-MT` based tables are a *loss-less join* decomposition of the `RDF-MT` based tables that includes all the attributes in the `RDF-MT` tables plus `ObsMID` and `MMID`. The attributes of the `RDF-MT` tables can be projected from G', thus, satisfying the *loss-less join* condition. ☐

Theorem 5.3.4. *If G is an SSN RDF graph and G' is a factorized RDF graph of G, and T_2 is the* `RDF-MT` *based tabular representation of G', then T_2 is in third normal form with respect to the* `RDF-MT` *based tabular representation of G.*

Proof. Recall [25], a table is in third normal form if for every $X \rightarrow Y$

- X is a super key, or
- $Y - X$ is a prime attribute

Considering that the following functional dependencies hold in `RDF-MT` based tables:

- `ObsMID` → `Procedure`, `Property`
- `MMID` → `Value`, `Unit`

- `ObsMID`, `MMID` → `ObsMID`, `MMID`
- `ObsID`, `ObsMID` → `ObsID`, `ObsMID`
- `ObsID`, `MID` → `ObsID`, `MID`
- `ObsID`, `SamplingTime` → `ObsID`, `SamplingTime`
- `SamplingTime` → `Timestamp`

It can be demonstrated that all the tables created after factorization are in third normal form. □

5.4 Experimental Study

We empirically study the impact of the proposed factorization techniques on several implementations of RDF data accessible through different query engines. We evaluate the impact on the size of the factorized RDF graphs as well as on query execution time in different query engines. RDF-3X [73] is utilized to evaluate the impact on the RDF stores. Spark [106] is utilized to evaluate the tabular representation of RDF graphs. Federated query engines, like MULDER[34] and ANAPSID[4], allow access the RDF data available through the endpoints and produce results incrementally. MULDER and ANAPSID are used to assess the impact of the factorization approach on the engines utilizing RDF implementations available through the endpoints. In this chapter, we investigated the following research questions: **ResearchQ1)** Are the proposed factorization techniques able to reduce the size of the semantic sensor data? **ResearchQ2)** How is the factorization time affected by the size of the RDF graphs? **ResearchQ3)** What is the impact of the queries against factorized RDF graphs on the query execution time? **ResearchQ4)** Is the performance of queries against factorized RDF graphs affected by the size of the factorized RDF graphs or RDF implementations? The experimental configuration to evaluate these research questions is as follows:

Datasets: Evaluation is conducted over three sensor datasets [77] described using the SSN Ontology. These RDF datasets are collected from around 20,000 weather stations in the United States and comprised of observations of different climate phenomena, e.g., visibility, temperature, precipitation, wind speed, and humidity, during the hurricane and blizzard seasons in the years 2003, 2004, and 2005[6]. Table 5.2 describes the main characteristics of these RDF datasets.

Queries: The SRBench-Version 0.9 queries[7] are used as baseline in our experimental testbed. Because RDF-3X does not evaluate queries with the OPTIONAL

---

[6] Available at: `http://wiki.knoesis.org/index.php/LinkedSensorData`

[7] `https://www.w3.org/wiki/SRBench`

operator, query 2 is modified to include only one BGP. Also, the STREAM clause, ASK queries, aggregate modifiers like AVG, GROUP BY, and HAVING clauses are not supported. So, only SELECT queries without aggregate modifiers are part of our testbed. Queries range from simple queries with one triple pattern to complex queries having up to fourteen triple patterns with UNION and FILTER clauses [8].
Metrics: We report on the following metrics: **a) Number of Triples (NT)** in the semantic sensor data collection. **b) Percentage Savings (%age NT Savings)** in the number of RDF triples after factorization. **c) Factorization Time (FT)** is the elapsed time between the request of factorization and the generation of the factorized RDF graph. **d) RDF3X Loading Time (LT)** is the time required to load RDF data to RDF3X store. **FT** and **LT** are computed as the *real time* of the *time* command of the Linux operating system. **e) Query Execution Time (ET)** is the elapsed time between the submission of the query to the engine and the complete output of the answer. In RDF and relational implementations **ET** is measured as the *real time* produced by the *time* command of the Linux operation system, whereas, in RDF implementations accessible through endpoints, **ET**, for SPARQL endpoints, is measured as the absolute wall-clock system time produced by the Python *time.time()* function. **f) dief@t** measures the continuous behaviour of a query engine that produces results incrementally. It computes the diefficiency of an engine in the first t time units of the query execution [3]. **g) Time For the First Tuple (TFFT)** is the elapsed time spent by the approach to produce the first query answer. **TFFT** is measured as the absolute wall-clock system time as reported by the Python *time.time()* function. **h) Completeness (Comp)** is the percentage of the number of answers produced by the approach after executing a query. **i) Throughput (T)** is the number of answers per second and is computed by dividing the total number of answers produced by the total execution time. For

Table 5.3: **Efficiency and Effectiveness of the Semantic Sensor Data Factorization**. Number of triples (**NT**) and savings (**%age NT Savings**) after factorization and the factorization time. Percentage savings in the number of triples (**%age NT Savings**) increases with the size of the dataset, while average number of triples per observation **Avg. NT per Obs.** decreases. Time that elapses during factorization (**FT**) as well as the RDF3X Loading Time (**LT**) for the factorized datasets are less than the RDF3X Loading Time (**LT**) for the original datasets.

Dataset ID	**Number of Triples(NT)**		**%age NT Savings**	**Avg. NT per Obs.**		**Factorization Time FT(s)**	**RDF3X LT(s)**	
	Original	**Factorized**		**Original**	**Factorized**		**Original**	**Factorized**
D1	38,054,493	17,800,156	53.22	9.29	4.34	417.229	460.511	252.976
D1D2	146,699,061	63,993,774	56.38	9.32	4.06	1,260.495	1,887.626	970.150
D1D2D3	325,827,468	136,979,696	**57.96**	9.31	**3.92**	2,147.239	3,822.723	1,982.697

---

[8] Details can be found at `https://sites.google.com/site/fssdexperimets/`

the RDF implementations available through endpoints, inverse of **TFFT** and **ET** are reported to have the same metric interpretation, i.e., higher is better.
Implementation: Three series of experiments were conducted over the gradually integrating sensor datasets in Table 5.2, i.e., D1, D1D2, and D1D2D3. **i)** SPARQL queries are executed using RDF3X engine over original and factorized RDF datasets. RDF3X engine executes queries over RDF data stored locally. The experiments are executed on a Linux Debian 8 machine with a CPU Intel I7 980X 3.3GHz and 32GB RAM 1333MHz DDR3. Queries are run on both cold and warm cache.[9] to access the query performance when data is cached. To run on warm cache, we executed the same query five times by dropping the cache just before running the first iteration of the query; thus, data temporally stored in cache during the execution of iteration i can be used in iteration $i + 1$. **ii)** In the second series of experiments, SQL queries were run using *Apache Spark*[10] over the universal, factorized, RDF-MT based tables and factorized representations of RDF-MT based tables. These tabular representations are stored using Parquet format in *HDFS*[11].

The experiments were conducted on a spark cluster consisting of one master node and three worker nodes created using *Docker*[12] containers, and the datasets are stored on a hadoop cluster containing one namenode and three datanodes created using docker containers. The experiments are performed on a machine with Intel(R) Xeon(R) Platinum 8160 CPU 2.10GHz and 23 RAM slots, where each RAM slot is DDR4 type, 32GB RAM size, and 2666MHz speed. Further, queries are run on cold and warm cache. **iii)** In the third series of experiments MULDER and ANAPSID engines are used to access the RDF datasets available as a SPARQL endpoint using *Virtuoso 7.2.2*, where each original and factorized dataset resides in a dedicated Virtuoso docker container. All queries are executed using MULDER and ANAPSID over the original and the factorized datasets. The experiments are conducted on a machine with Intel(R) Xeon(R) Platinum 8160 CPU 2.10GHz and 23 RAM slots, where each RAM slot is DDR4 type, 32GB RAM size, and 2666MHz speed.

5.4.1 Efficiency and Effectiveness of Factorized RDF

For evaluating the efficiency and effectiveness of the proposed factorization techniques and to answer the research questions **ResearchQ1** and **ResearchQ2**,

---

[9] To run cold cache, we clear the cache before running each query by performing the command `sh -c "sync ; echo 3 > /proc/sys/vm/drop_caches"`

[10] `http://spark.apache.org/`

[11] `https://hadoop.apache.org/`

[12] `https://www.docker.com/`

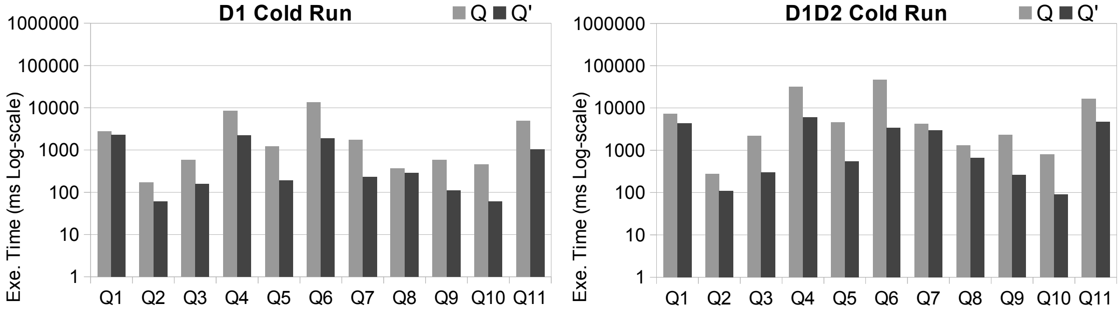
we execute Algorithm 4 by gradually integrating the datasets in Table 5.2, i.e., **D1**, **D1D2**, and **D1D2D3**. Effectiveness is reported based on the reduction of RDF triples (**NT**), while efficiency is measured in terms of factorization time (**FT**) and RDF3X loading time (**LT**). Table 5.3 reports on the number of RDF triples (**NT**) in datasets **D1**, **D1D2**, and **D1D2D3** before and after the factorization. The results demonstrate that the proposed factorization techniques are capable of reducing the RDF triples by at least **53.22%**. Moreover, the results report that the factorized representation of sensor observations requires in average a small number of RDF triples, i.e., five RDF triples instead of ten, while preserving all the information within the original RDF graph. These results allows us to positively answer research question **ResearchQ1**, i.e., factorized RDF graphs effectively reduce the size of RDF graphs. We also measure factorization time and factorized RDF loading time to RDF3X, and compare to the time required by RDF3X to upload the original RDF graphs (Table 5.3). In all datasets, Algorithm 4 as well as factorized RDF loading to RDF3X requires less than **50%** of the time consumed by RDF3X during original RDF data loading. Thus, with these results research question **ResearchQ2** can be also positively answered.

5.4.2 Impact of Factorized RDF on Query Processing

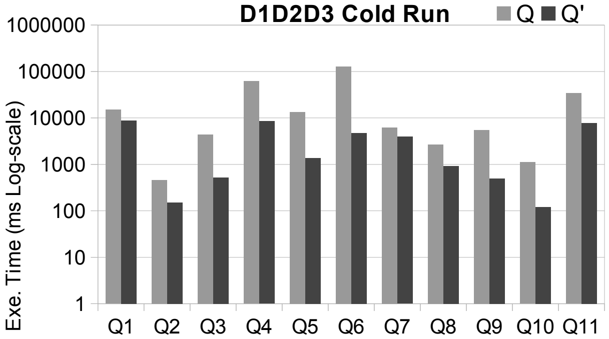
To answer research questions **ResearchQ3** and **ResearchQ4**, we analyze the efficiency of the proposed representations by running the queries generated using Algorithm 5 over diverse RDF implementations. We run the queries over original and factorized RDF data using centralized RDF engine, Big Data engines and federated RDF engines.

Query Execution over Centralized RDF Engines

In order to exploit the benefits of cache, queries are executed on cold and warm cache. The advantages of running these queries on cold and warm caches using RDF3X are analyzed over the gradually increasing RDF datasets. The original queries Q are compared to the reformulated queries Q'. Original queries (Q) are executed against the original datasets, while plans for reformulated queries (Q') are run against gradually increasing factorized datasets. Figure 5.14 report on the query execution time (milliseconds. log-scale) with cold cache, while Figure 5.15 depicts the observed execution time when queries are run on warm cache; the minimum value is reported in all the queries. In all cases, reformulated queries over factorized RDF graphs exhibit better performance whenever they are run on cold and warm caches. This observation supports the statement that because observation and measurement multiplicity is reduced to one in factorized RDF graphs,



(a) RDF3X Cold Run over Dataset $D1$ (b) RDF3X Cold Run over Dataset $D1D2$



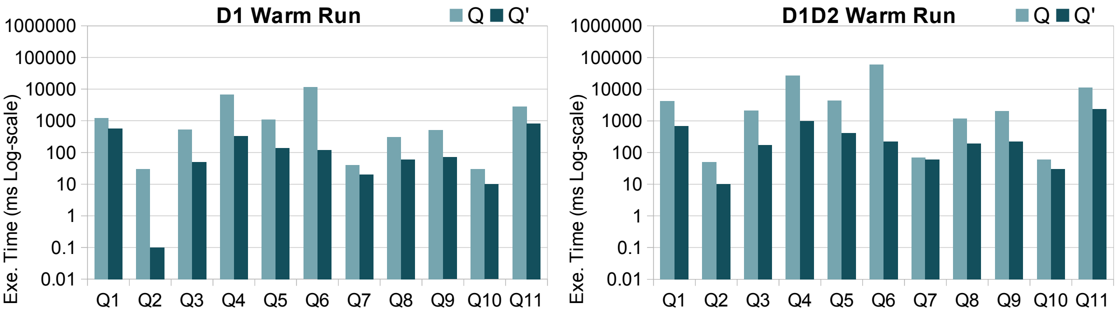
(c) RDF3X Cold Run over Dataset $D1D2D3$

Figure 5.14: **Query Execution Time ET (ms Log-scale) over RDF3X on Cold Cache**. SPARQL queries are executed over the gradually increasing original and factorized RDF data in **cold** cache. Original SPARQL queries Q and rewritten SPARQL queries Q' are evaluated on cold cache against original and factorized RDF graphs, respectively. Rewritten SPARQL queries reduce execution time on factorized RDF graphs by up one order of magnitude for all datasets.

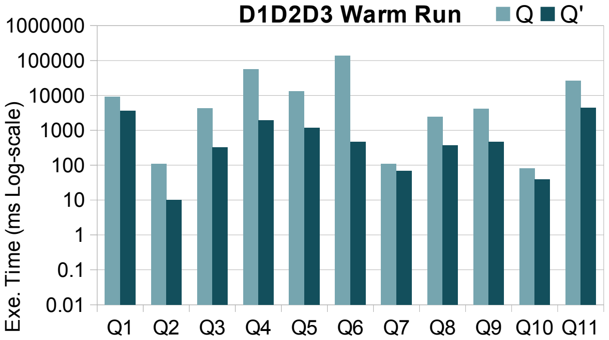
factorized queries produce small intermediate results which can be maintained in resident memory and re-used in further executions. Thus, the performance of factorized queries is considerable better with warm cache, overcoming other executions by up to three orders of magnitude, e.g., Q2 and Q6. Results also suggest that performance of reformulated queries is not affected by the RDF graph size, e.g., large RDF graphs like D1D2D3 with 325,827,468 RDF triples.

Query Execution over Big Data Engines

We evaluate the performance of the query execution task whenever queries are executed against the relational representations, i.e., universal and factorized



(a) RDF3X Warm Run over Dataset $D1$ (b) RDF3X Warm Run over Dataset $D1D2$



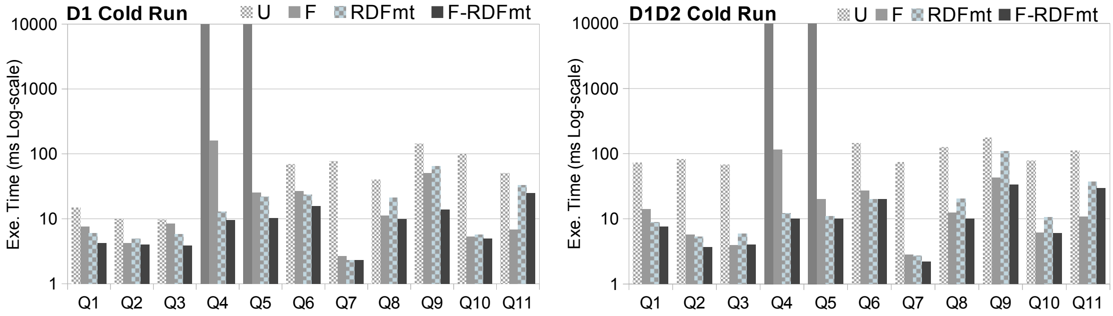
(c) RDF3X Warm Run over Dataset $D1D2D3$

Figure 5.15: **Query Execution Time ET (ms Log-scale) over RDF3X on Warm Cache**. SPARQL queries are executed over the gradually increasing original and factorized RDF data in **warm** cache. Original SPARQL queries Q and rewritten queries Q' are evaluated on warm cache against original and factorized RDF graphs, respectively. To warm cache up, content of resident memory is flushed, and each query is executed five times; the lowest value of execution time is reported. Rewritten queries over factorized RDF graphs produce intermediate results that can be maintained in resident memory and re-used in further executions, and reduce query execution time by up two order of magnitude.

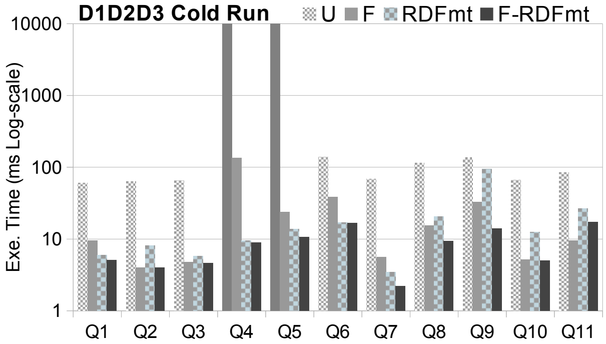
tables, and the tabular representations of original and factorized RDF data around the RDF molecule templates. The performance of queries over Parquet tables depends on the number of attributes included in the query, as well as on the ratio between the attributes in the query and the attributes in the tables [13]. In queries against the universal table, the ratio between the number of attributes used in

---

[13] `http://techblog.appnexus.com/blog/2015/03/31/parquet-columnar-storage-for-hadoop-data/`



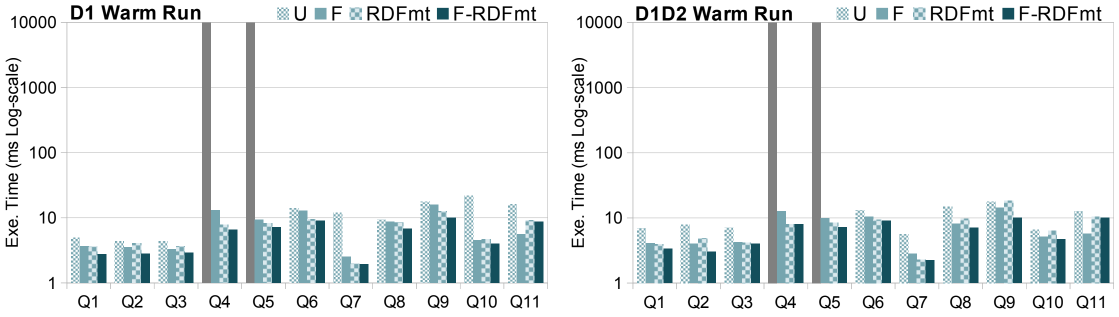
(a) Parquet Table Cold Run over Dataset $D1$ (b) Parquet Table Cold Run over Dataset $D1D2$



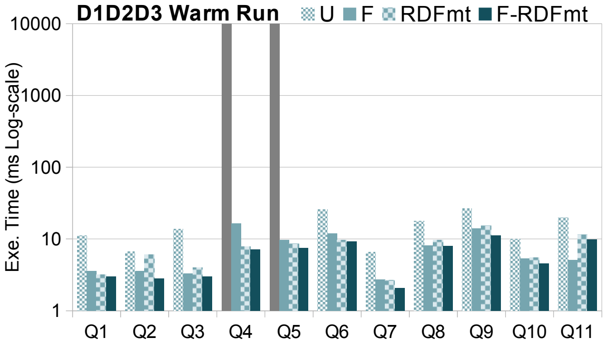
(c) Parquet Table Cold Run over Dataset $D1D2D3$

Figure 5.16: **Query Execution Time ET (ms Log-scale) over Relations on Cold Cache**. Query evaluation over the gradually increasing universal, factorized, and RDF-MT based tabular representations of the original and factorized RDF graphs stored using parquet tables in **cold** cache. SQL queries representing the original and rewritten SPARQL queries are evaluated over the universal *(U), factorized (F), RDF-MT based tabular representations of original (RDFmt) and factorized (F-RDFmt) RDF graphs stored in parquet tables. Execution are timed out after 100 minutes. SQL version of the rewritten SPARQL queries over the parquet tables storing the factorized (F) and RDF-MT based tables of the factorized RDF graph (F-RDFmt) reduce execution time by up two orders of magnitude.*

the universal table queries and the attributes varies from **0.09** to **0.45**. While the ratio in factorized queries is in the range from **0.46** to **0.75**, and in original and factorized RDF molecule templates is **0.25** and **0.78**. So, based on this statement, queries over the universal table should be faster than queries over the factorized tables and RDF molecule template based tabular representations. However, as observed in Figures 5.16 and 5.17, reformulated queries over factorized RDF molecule template parquet tables speed up execution time to almost two orders of magni-



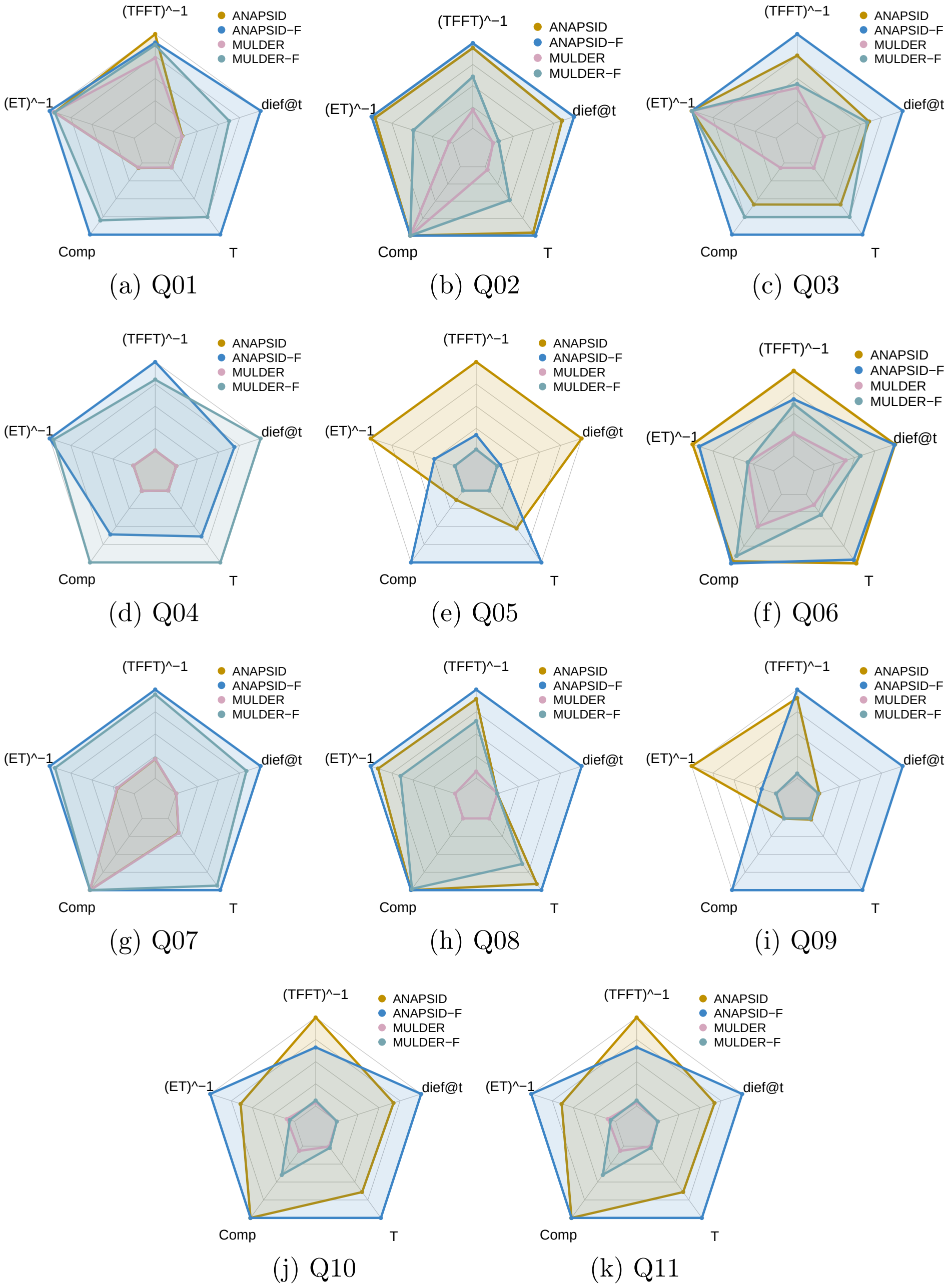
(a) Parquet Table Warm Run over Dataset $D1$ (b) Parquet Table Warm Run over Dataset $D1D2$



(c) Parquet Table Warm Run over Dataset $D1D2D3$

Figure 5.17: **Query Execution Time ET (ms Log-scale) over Relations on Warm Cache**. Query evaluation over the gradually increasing universal, factorized, RDF-MT based tabular representations of the original and factorized RDF graphs stored using parquet tables in **warm** cache. To warm cache up, content of resident memory is flushed, and each query is executed five times; the lowest value of execution time is reported. SQL queries representing the original and rewritten SPARQL queries are evaluated over the universal (*U*), *factorized (F), RDF-MT based tabular representations of original (RDFmt) and factorized (F-RDFmt) RDF graphs stored in parquet tables. Execution are timed out after 100 minutes. SQL versions, of the rewritten queries, over the parquet tables storing the factorized (F) and RDF-MT based tabular representations (F-RDFmt) reduce execution time by up one orders of magnitude.*

tude, except Q11 where factorized tables are performing better. Factorized RDF molecule templates reduce the size of tables by creating them around each molecule template and factorization further removes data redundancies. Actually, in two queries (Q4 and Q5), query execution over the universal table times out after 60 minutes. These results allow us to conclude that formulated queries also reduce



(a) Q01 (b) Q02 (c) Q03 (d) Q04 (e) Q05 (f) Q06 (g) Q07 (h) Q08 (i) Q09 (j) Q10 (k) Q11

Figure 5.18: **Performance of Factorized RDF Graphs over SPARQL Endpoints.** Original and factorized RDF graphs accessible via SPARQL endpoints are queried using ANAPSID and MULDER query engines. SPARQL queries are executed over the original and factorized RDF graphs representing dataset D1D2D3 using ANAPSID and MULDER. Both query engines generate complete answers with improved diefficiency and throughput for the factorized data, while the query execution time is reduced. Same results are observed in datasets $D1$ and $D1D2$.

execution time on Parquet tables.

Query Execution over Federated RDF Engines

We evaluate query performance over RDF implementations available through virtuoso endpoints using MULDER and ANAPSID query engines. The results for the largest dataset D1D2D3, shown in Figure 5.18, indicate that the factorization improves performance of ANAPSID and MULDER to process RDF data available through endpoints, and same results are observed for the other two datasets, i.e., D1 and D1D2. Overall ANAPSID performs better over factorized RDF data for almost all the queries, except query Q4 where MULDER performs better over the factorized RDF data. The query engines give significant improvement in query execution time, throughput, and dief@t while producing complete answers over the factorized data. In Q5 MULDER times out over factorized and original RDF data, whereas, ANAPSID generates complete results from factorized data in 300 seconds as opposed to the original data which generates only one fourth of the results from the factorized data in 73.49 seconds. Dief@t is computed at the timestamp that is smaller among all the timestamps required by the approaches to be compared, i.e., 73.49 seconds. Since at this time all the results for the original data have been computed, whereas only some of the factorized data results are computed at this time, thus, giving a smaller value for dief@t for the factorized data. Actually, computing complete query answers from the factorized data takes more time than producing few number of query results from the original data These results reveal that query performance improves over the factorized data available through SPARQL endpoints. Thus, we can positively answer questions **ResearchQ3** and **ResearchQ4**.

5.5 Summary

Large collections of sensor data are semantically described using ontologies, e.g., the Semantic Sensor Network (SSN) ontology. Semantic sensor data are RDF descriptions of sensor observations from related sampling frames or sensors at multiple points in time, e.g., climate sensor data. Sensor values can be repeated in a sampling frame, e.g., a particular temperature value can be repeated several times, resulting in a considerable increase in data volume. We devise a factorized compact representation of semantic sensor data using linked data technologies to reduce repetition of same sensor values, and propose algorithms to generate collections of factorized semantic sensor data that can be managed by existing RDF triple stores. In addition, we present tabular-based representations for semantic sensor data to exploit Big Data frameworks. We empirically study the effectiveness and efficiency of the proposed RDF and tabular-based representations of semantic sensor data. We show that the size of semantic sensor data is reduced by more than 50% on average without loss of information. Further, we have evaluated the impact of the proposed representations of semantic sensor data on query execution. Results suggest that query optimizers can be empowered with semantics from factorized representations to generate query plans that effectively speed up query execution time on factorized semantic sensor data in RDF as well as Big Data engines.

Chapter 6

Integration of Streaming Observational Data

In streaming data, produced by diverse IoT devices over time, the problem of tackling redundancies is more challenging and requires the redundancies to be detected on the fly. Despite the enormous benefits of optimization, monitoring, and maintenance rendered by IoT devices, an ample amount of observational data is generated continuously. Semantically describing IoT generated data using ontologies enables a precise interpretation of this data. However, ontology-based descriptions tremendously increase the size of the data, and in the presence of repeated sensor measurements, a large number of observations are duplicated that do not contribute to new insights during query processing or data analytics. Chapter 5 deals with historical data and addresses the problem of redundancies in RDF knowledge graphs, whereas this chapter deals with data on motion, i.e., data is received on the fly, and addresses the problem of on-demand knowledge graph creation from streaming data. In this chapter, we capture the redundancies in streaming data on the fly and compute knowledge graph without any duplicates. Moreover, we present query execution techniques over streaming data. Figure 6.1 presents the challenge we tackle to solve the problem and the contribution to address the challenge. The research work presented in this chapter is based on the publications [54, 56]. This chapter addresses the following research questions:

RQ3: How can on-demand knowledge graph building reduce the size of the streaming observational data?

RQ4: How can on-demand knowledge graph building speed up query processing?

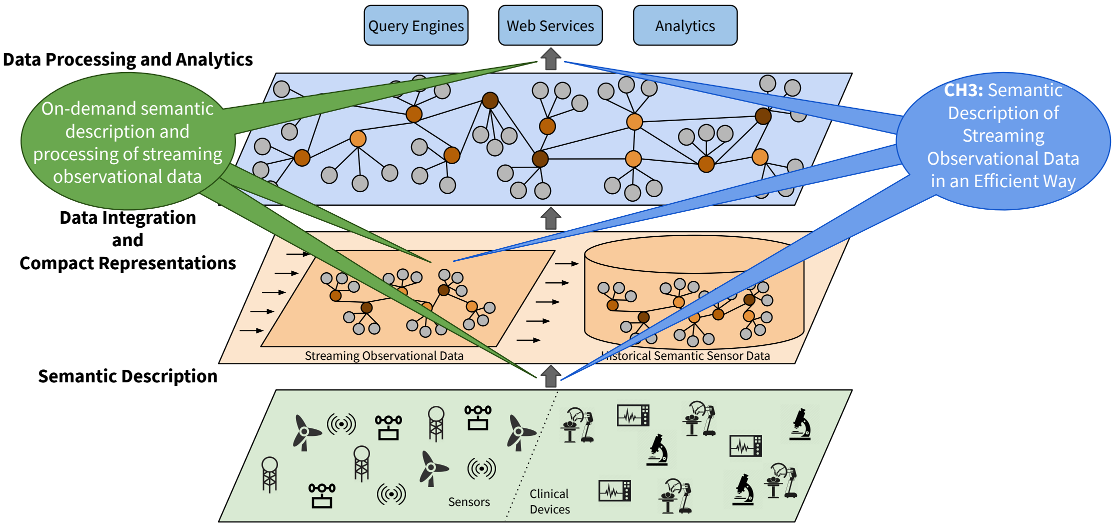

Query Engines
Web Services
Analytics
Data Processing and Analytics
On-demand semantic description and processing of streaming observational data
CH3: Semantic Description of Streaming Observational Data in an Efficient Way
Data Integration and Compact Representations
Streaming Observational Data
Historical Semantic Sensor Data
Semantic Description
Sensors
Clinical Devices

Figure 6.1: **Challenges and Contributions.** This chapter focuses on the problem of semantic description of streaming observational data, and presents techniques for on-demand semantic description of streaming observation data.

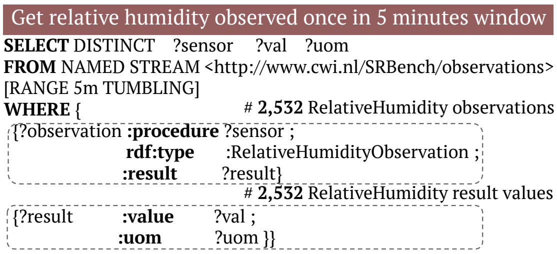
In order to address the research questions **RQ3** and **RQ4**, we devise a knowledge-driven approach named DESERT for streaming data that is able to on-Demand factorizE and Semantically Enrich stReam daTa. DESERT resorts to a knowledge graph to describe IoT stream data; it utilizes only the data required to answer an input continuous SPARQL query and applies a novel method of data factorization to reduce duplicated measurements in the knowledge graph. The main contributions of this chapter are:

- A formal framework for defining on-demand stream data factorization and semantification methods, as well as the implementation of this framework in DESERT.
- An extensive analysis of the main characteristics of the state-of-the-art knowledge graph building and continuous SPARQL query processing.
- An extensive empirical evaluation of DESERT framework, demonstrating the impact of several parameters, e.g., the streaming window size and data stream speed on the knowledge graph size and continuous query execution on the DESERT performance.

This chapter is structured as follows: Section 6.1 motivates the work presented in this chapter by building knowledge graphs from streaming data given a query over the data stream. A knowledge graph constructed from all the data in the stream is colossal, whereas the one created using the data required to answer the query is relatively sparse. However, several data duplicates are found in the sparse knowledge graph resulting in an increase in the knowledge graph size. Section 6.2 formalizes the problem solved in this chapter. Section 6.3 describes the main components of the DESERT architecture that is proposed to address the problem presented in Section 6.2. DESERT resorts to a knowledge graph to describe IoT stream data; it utilizes only the data required to answer an input continuous SPARQL query and applies a novel method of data factorization to reduce duplicated measurements in the knowledge graph. Section 6.4 reports the results of our empirical evaluation. The performance of DESERT is empirically studied on a collection of continuous SPARQL queries from SRBench, a benchmark of IoT stream data and continuous SPARQL queries. Furthermore, data streams with various combinations of uniform and varying data stream speeds and streaming window size dimensions are considered in the study. Experimental results suggest that DESERT is capable of speeding up query processing while creates knowledge graphs that include no replications. We summarize this chapter in Section 6.5.

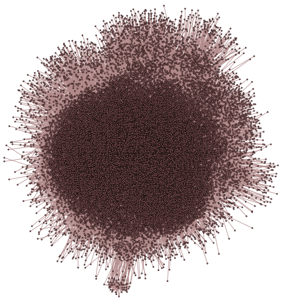
6.1 Motivating Example

Figure 6.2b illustrates a knowledge graph (KG) obtained by semantically describing, using the SSN ontology, a stream of 18, 632 observations about different weather phenomena in the year 2008; the stream contains observations required to answer the continuous SPARQL query in Figure 6.2a. The sensors generating the data stream produce 15 observations per second of type rainfall, precipitation, pressure, wind direction, relative humidity, wind speed, and temperature. During the 20 minutes of query execution, all the observations in the stream are semantically described within five minutes streaming window. An inspection to a graphical visualization of the knowledge graph, generated by the Cytoscape tool [96], illustrates that the knowledge graph is dense with 173, 238 RDF triples describing 23, 792, 818 weather observations. Moreover, Figure 6.2d reports on a network analysis of the knowledge graph from Figure 6.2b. It shows the high values 43, 889.0, 7.0, and 18, 632.0, of the *number of nodes*, *avg. number of neighbors*, and *multi-edge node pairs*, respectively, revealing thus the complexity of the knowledge graph due to various connections among the nodes. In addition, the knowledge graph in Figure 6.2c represents an ontology-based description of 2, 532 relative humidity observations within a five-minute streaming window, during a 20-minutes execution of the query in Figure 6.2a. The generated knowledge graph is sparse containing 28, 997 RDF triples, describing the weather observations about relative humidity required to answer the query. Furthermore, the network analysis of the knowledge graph from Figure 6.2c, shown in Figure 6.2d, demonstrates

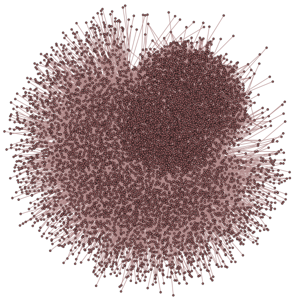


```
Get relative humidity observed once in 5 minutes window
SELECT DISTINCT  ?sensor   ?val   ?uom
FROM NAMED STREAM <http://www.cwi.nl/SRBench/observations>
[RANGE 5m TUMBLING]
WHERE {                         # 2,532 RelativeHumidity observations
{?observation :procedure ?sensor ;
              rdf:type    :RelativeHumidityObservation ;
              :result     ?result}
                                # 2,532 RelativeHumidity result values
{?result      :value      ?val ;
              :uom        ?uom }}
```

(a) Query Q1 from SRBench



(b) **Complete** KG



(c) **Reduced** KG

Parameter	C.KG	R.KG
# of Nodes	43,889	9,894
Avg.# of neighbor	7	5
Multiedge node pair	18,632	3,793

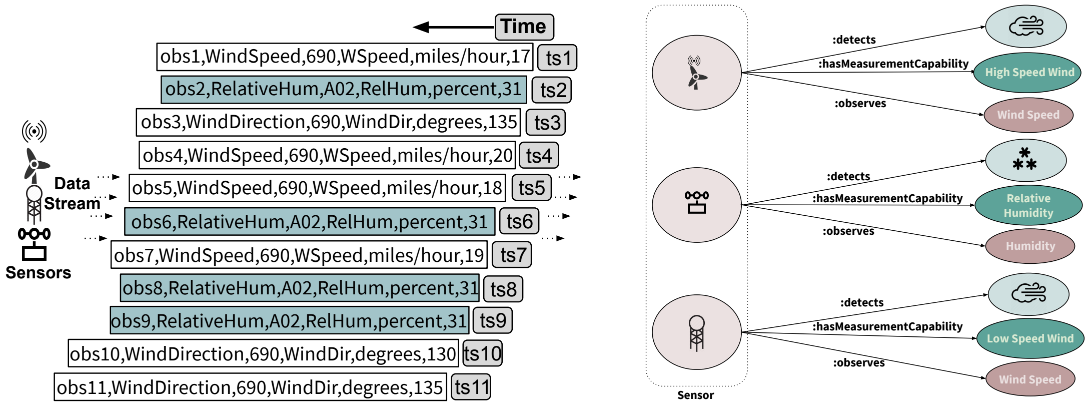
(d) **Analysis** of KGs

Figure 6.2: **Motivating Example**. Knowledge Graphs (KGs) with observations from the year 2008 MesoWest data stream of speed 15 obs/sec., for 20 min. (a) A continuous SPARQL query from SRBench retrieves relative humidity observations within five minutes window; (b) A knowledge graph with all observations sensed within five minutes window for 20 minutes; (c) A knowledge graph with the relative humidity observations, required to answer the query in Figure 6.2a, for 20 minutes; (d) Analysis of complete (C.KG) and reduced (R.KG) knowledge graphs in Figures 6.2b and 6.2c, respectively. The knowledge graphs in Figures 6.2b and 6.2c, and the analysis in Figure 6.2d are generated by `Cytoscape tool`[1].

that the knowledge graph is relatively less complex with low values, $9,894.0$, 5.0, and $3,793.0$ of the *number of nodes*, *avg. number of neighbors*, and *multi-edge node pairs*, respectively. However, an inquiry of the data in knowledge graph, from Figure 6.2c, shows that there are only 85 distinct measurement values among the $2,532$ relative humidity observations. Such a small number of distinct measurement values within the streaming data indicates that the number of *distinct values* produced by the IoT devices is much lower than the number of the *observations*. With cases like this being a common occurrence in the semantic description of the streaming data, further techniques are needed to improve the knowledge graph

---

[1] `http://www.cytoscape.org/`


Time
obs1,WindSpeed,690,WSpeed,miles/hour,17 ts1
obs2,RelativeHum,A02,RelHum,percent,31 ts2
obs3,WindDirection,690,WindDir,degrees,135 ts3
obs4,WindSpeed,690,WSpeed,miles/hour,20 ts4
obs5,WindSpeed,690,WSpeed,miles/hour,18 ts5
obs6,RelativeHum,A02,RelHum,percent,31 ts6
obs7,WindSpeed,690,WSpeed,miles/hour,19 ts7
obs8,RelativeHum,A02,RelHum,percent,31 ts8
obs9,RelativeHum,A02,RelHum,percent,31 ts9
obs10,WindDirection,690,WindDir,degrees,130 ts10
obs11,WindDirection,690,WindDir,degrees,135 ts11
Data Stream
Sensors
:detects
:hasMeasurementCapability
:observes
High Speed Wind
Wind Speed
Relative Humidity
Humidity
Low Speed Wind
Sensor

(a) **Stream** of Data (b) Knowledge Graph

Figure 6.3: **Example of IoT Data Stream and Knowledge Graph**. (a) A stream of IoT data containing *WindSpeed*, *RelativeHum*, and *WindDirection* observations over a period of time. The continuous SPARQL query execution in Figure 6.2a requires only *RelativeHum* observations to produce results. *RelativeHum* observations, *obs*2, *obs*6, *obs*8, and *obs*9 at timestamps *ts*2, *ts*6, *ts*8, and *ts*9, respectively, have the same measurement value 31; (b) A Knowledge Graph composed of wind speed and humidity weather phenomena; RDF triples semantically describe a sensor's measurement capability and observable property.

creation and continuous query processing.

6.2 Problem Statement and Proposed Solution

We leverage the semantics encoded in the streaming data generated continuously by the IoT devices over time, and define the tackled problem and the proposed solutions.

Definition 6.2.1 (Knowledge Graph [84])**.** *Given a set T of RDF triples, a knowledge graph is represented as $G = (V, E, L)$, where V is a set of nodes defined as $V = \{s \mid (s\ p\ o) \in T\} \cup \{o \mid (s\ p\ o) \in T\}$; E is a set of edges defined as $E = \{(s\ p\ o) \in T\}$, and L is a set of edge labels defined as $L = \{p \mid (s\ p\ o) \in T\}$.*

Example 6.2.1. *Figure 6.3b illustrates a knowledge graph that semantically describes three sensors observing wind speed and relative humidity weather phenomena in the data stream in Figure 6.3a; for clarity, URIs are excluded. The knowledge graph semantically describes the measurement capability and observable property of sensors. The sensors are able to observe the low and high wind speeds and the relative humidity measurements over time.*

Definition 6.2.2 (Observed Tuple)**.** *Given obs, $proc$, ph, pp, val, and uom corresponding to an observation identifier, procedure, observed phenomenon, observed property, a measurement value, and measurement unit, respectively. An Observed Tuple is a pair $obs^{ts} = (oo, ts)$ where oo is an n-tuple $oo = \langle obs, ph, proc, pp, uom, val\rangle$ and ts is a numeric value; $obs^{ts} = (oo, ts)$ represents an observation obs of a phenomenon ph and parameter pp, collected at timestamp ts by procedure $proc$ with measurement value val and unit uom.*

Example 6.2.2. *Figure 6.3a presents observed tuples representing wind speed, wind direction, and relative humidity observations generated by the sensors $A02$ and 690, within time interval $[ts1, ts11]$. All the tuples contain the complete information about the corresponding observations. Each observation is identified using a unique identifier, e.g., $obs1$, $obs5$, and $obs10$, within an observed tuple. The IoT stream data represented by each tuple in Figure 6.3a can be seen as the data about the observations and measurements collected during the act of carrying out observations. The data about an observation is described by the unique identifier, e.g., $obs5$, the phenomenon being observed, e.g., WindSpeed, the sensor generating the observation, e.g., 690, and the property of the phenomenon being observed, e.g., WSpeed. The measurement data contain the measurement value of the observed property, e.g., 18, and the measurement unit, e.g., miles/hour.*

Observation tuples are collected in streams of data which are defined as follows:

Definition 6.2.3 (Data stream)**.** *A data stream DS is an ordered set of observed tuples:*

$$DS = \{obs_i^{ts} | obs_i^{ts} = (oo_i, ts_i) \wedge ts_{i-1} \leq ts_i < ts_{i+1}\}$$

Knowledge graphs can be built by semantically describing the observed tuples, generated within the data streams, in a particular range of time or *window*. The SSN ontology can be used to create these *Complete Knowledge Graphs* which are defined as follows:

Definition 6.2.4 (Complete Knowledge Graph)**.** *Given a set $DSS = \{DS_1, DS_2, \dots, DS_n\}$ of data streams DS_i and a window $\omega = [ts_a, ts_b]$. A Complete Knowledge Graph over* DSS *in the window ω named KG^{ω}_{DSS}, is defined as the union of the knowledge graphs $KG^{\omega}_{DS_i}$ over data streams DS_i in DSS.*

A knowledge graph $KG^{\omega}_{DS_i}$ is built from RDF triples in $T^{\omega}_{DS_i}$ that describe all the observation tuples of DS_i in the window ω using the transformation function $\delta(.)$:

- $T^{\omega}_{DS_i}$ is a set of RDF triples described using the SSN ontology: $T^{\omega}_{DS_i} = \{\delta(obs_i^{ts}) | obs_i^{ts} = (oo_i, ts_i) \wedge ts_i \in \omega \wedge obs_i^{ts} \in DS_i\}$

- $\delta(.)$ is a transformation function that outputs the RDF representation of an observation tuple obs_i^{ts}.

Example 6.2.3. *Figure 6.2b illustrates an RDF graph that includes all the RDF triples of the data streams with observations sensed within five minute window in a period of* 20 *minutes. As observed in Figure 6.2, a complete knowledge graph may include RDF triples that are not required to answer a set of input continuous queries. We define a reduced knowledge graph as the knowledge graph that is composed of the RDF triples required to produce the complete answer of a continuous query. For simplicity, we define a continuous SPARQL query as a set of basic graph patterns BGP_j associated with an RDF type representing a phenomenon and produces RDF data mappings within the streaming window over time.*

Definition 6.2.5 (Continuous SPARQL Query)**.** *A continuous query Q is a triple $Q = \langle BGP, PH, \omega \rangle$ where:*

- *BGP is a set of basic graph patterns $BGP_j = \{t_1, \ldots, t_n\}$ where t_k is a triple pattern; each BGP_j is associated with a phenomenon RDF type in PH.*
- *The evaluation of BGP_j against a data stream DS_i in a window ω, named $[[BGP_j]]_{DS_i}^{\omega}$, produces mappings μ from variables in BGP_j to resources and literals in DS_i within the window ω.*

Based on a continuous SPARQL query Q against the streams of data generated continuously by the IoT devices over time, a reduced knowledge graph is defined as follows:

Definition 6.2.6 (Reduced Knowledge Graph)**.** *Given a continuous SPARQL query $Q = \langle BGP, PH, \omega \rangle$, a set $DSS = \{DS_1, DS_2, \ldots, DS_n\}$ of data streams DS_i, and a window $\omega = [ts_a, ts_b]$, a Reduced Knowledge Graph over* DSS *in the window ω for Q, named KG_{DSS}^{Q}, is defined as the union of the knowledge graphs $KG_{DS_i}^{Q}$ over data streams DS_i in* DSS *that only include the results of $[[BGP]]_{DS_i}^{\omega}$.*

Example 6.2.4. *Figure 6.2c presents a knowledge graph that only includes RDF triples required to answer the continuous SPARQL query in Figure 6.2a. A reduced knowledge graph also provides a compact representation of the observed data, where measurements are represented only once and associated with their corresponding observations. Consider, for instance, the observations $obs2$, $obs6$, $obs8$ and $obs9$ in Figure 6.3a; they report the same value of* 31 *percent of relative humidity. A compact representation in a reduced knowledge graph allows for the description of the measurement with the value* 31 *and the unit percent and the association of this measurement to the corresponding observations. The temporal measurement multiplicity of the measurement with value* 31 *and unit percent within time interval $[ts1, ts11]$ is four. Instead of repeating these values, the multiplicity of measurements in a reduced knowledge graph is always one.*

We formally define a temporal measurement multiplicity as follows:

Definition 6.2.7 (Temporal Measurement Multiplicity)**.** *Given an data stream DS, a window ω, the temporal multiplicity of a measurement with values val and uom in DS within ω, $M_m(val, uom, DS, \omega)$, corresponds to the number of observation tuples in the window ω that have same value val and unit uom.*

$$M_m(val, uom, DS, \omega) = \mid \{obs^{ts} \mid obs^{ts} \in DS \wedge obs^{ts} = (\langle obs, ph, proc, pp, uom, val\rangle, ts) \wedge ts \in \omega\} \mid$$

In this chapter, we address the problem of building a knowledge graph on-demand according to the streaming data required to answer a continuous SPARQL query. Observations in data streams are represented in a knowledge graph in a way that multiplicity of the entities represented in the knowledge graph is reduced to one. We call this compact representation of the data in a knowledge graph, *on-demand factorization.* In previous work, Karim et al. [55] propose techniques for factorizing historical RDF data; the experimental results show evidence that representing factorized RDF data not only reduces the size of the RDF datasets but also speeds up query execution time. Building on factorization methods for RDF data, we devise solutions to the problem of generating knowledge graphs from streaming data on demand; the generated knowledge graph maintains the characteristics observed and demonstrated in RDF historical data, i.e., answers of queries over original and factorized RDF data are the same while the knowledge graph size is reduced. In the next section, we formally define the problem of creating factorized knowledge graphs on-demand and present DESERT, a continuous SPARQL query engine able to create and query these knowledge graphs.

6.2.1 Problem Statement

We define the problem of building a knowledge graph composed of the RDF triples required to answer a continuous SPARQL query; RDF triples in the data are factorized. We name this problem, the *on-demand knowledge graph creation* (OKGC); it is defined as follows:

Given a continuous SPARQL query $Q = \langle BGP, PH, \omega\rangle$, a set DSS= $\{DS_1, DS_2, \ldots, DS_n\}$ of data streams, and a window ω, the on-demand knowledge graph creation problem corresponds to finding a reduced knowledge graph $\mathrm{KG}^{Q}_{\mathrm{DSS}}$, such that:

- Answer correctness and completeness of Q is enforced in $\mathrm{KG}^{Q}_{\mathrm{DSS}}$. The results of evaluating Q over $\mathrm{KG}^{Q}_{\mathrm{DSS}}$ and over $\mathrm{KG}^{\omega}_{\mathrm{DSS}}$ are the same.
- Measurements are factorized in $\mathrm{KG}^{Q}_{\mathrm{DSS}}$. For each measurement value *val* and unit *uom* in the observation tuples of the data streams DS in DSS,

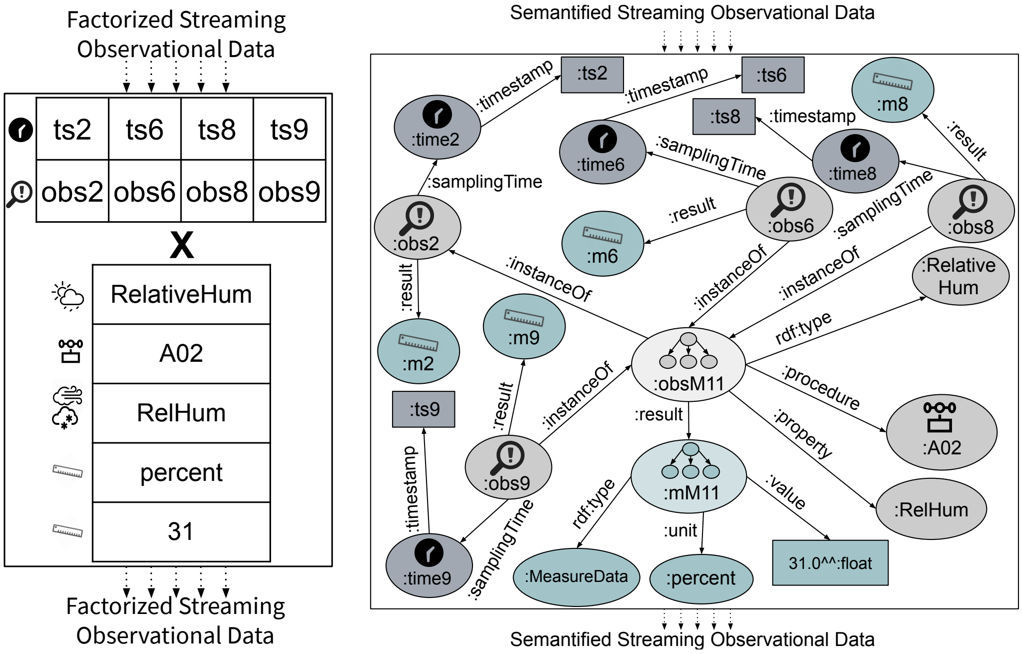
temporal measurement multiplicity of *val* and *uom* is equal to one, i.e., $M_m(val, uom, DS, \omega)$=1.

- Only relevant observation tuples are semantically enriched and included in $\mathrm{KG}^{Q}_{\mathrm{DSS}}$. There is no observation tuple obs^{ts} in a data stream DS of DSS such that $\delta(obs^{ts})$ is in $\mathrm{KG}^{Q}_{\mathrm{DSS}}$, but if $\delta(obs^{ts})$ is removed, the results of evaluating Q over $\mathrm{KG}^{Q}_{\mathrm{DSS}}$ and against $\mathrm{KG}^{\omega}_{\mathrm{DSS}}$ remain the same.

Example 6.2.5. *Figure 6.4 illustrates an instance of the on-demand knowledge graph creation problem, given the continuous SPARQL query in Figure 6.2a and the stream data in Figure 6.3a. Firstly, the relevant observation tuples are selected and factorized to ensure that the temporal multiplicity is equal to one; Figure 6.4a shows factorization of relative humidity measurements with value* 31% *over the observations* $obs2$, $obs6$, $obs8$ *and* $obs9$. *Figure 6.4b presents the knowledge graph resulting from applying the transformation function* $\delta(.)$ *to the factorized observations in Figure 6.4a; note that there are no nodes with the same label.*

6.2.2 Proposed Solution

We propose DESERT, a continuous SPARQL query engine able to solve the on-demand knowledge graph creation problem. DESERT is build over the CSPARQL-engine and is able to receive a continuous SPARQL query and IoT data streams as inputs. Streaming data from different IoT devices is retrieved by the DESERT framework using the customized wrappers based on the observation tuples required to answer the input continuous SPARQL query. The retrieved data, corresponding to the observations required to answer the query, is factorized, semantified and integrated into a knowledge graph. The knowledge graph models the semantics of the integrated data in terms of the observations required to answer the query, as well as the relationships among them. DESERT implements query rewriting techniques, decomposes the input queries into subqueries against the knowledge graph and the data streams, factorizes and semantifies the data received from different data streams, and integrates the semantified such data into the knowledge graph. Finally, it transforms and optimizes the queries to run against the knowledge graph to produce the query answers. Further, wrappers hide the complex implementations of the heterogeneous data streams. The selective retrieval of data streams, in terms of observation tuples that are required to answer an input continuous SPARQL query, reduces the size of the knowledge graph. In addition, the factorization of the measurements produce more compact representations of the knowledge graph, resulting in faster query execution times.



(a) **On-demand Factorization** of a Data Stream

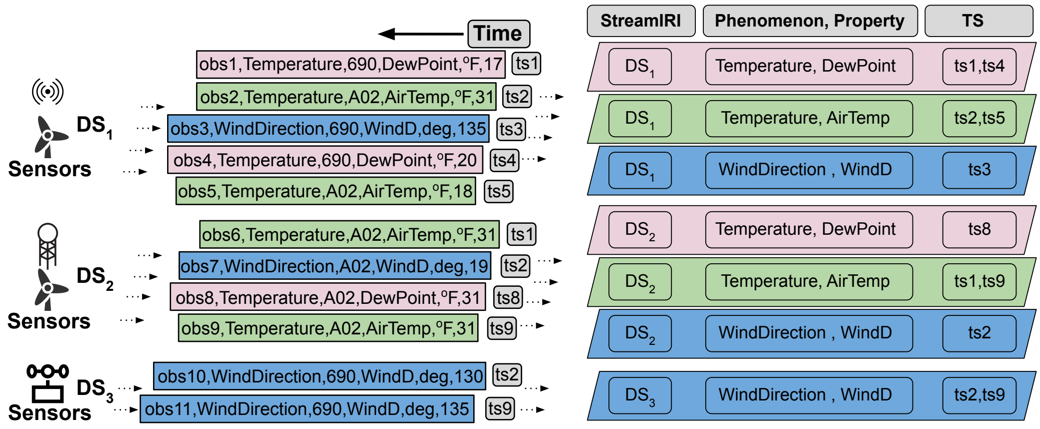
(b) **On-demand Semantic Enrichment** of a Data Stream containing factorized data required to answer a continuous SPARQL query

Figure 6.4: **Instance of the On-demand Knowledge Graph Creation Problem**. (a) Relative humidity observations $obs2$, $obs6$, $obs8$, and $obs9$, at timestamp $ts2$, $ts6$, $ts8$ and $ts9$ respectively, are required to answer the query in Figure 6.2a and have same values for *:property*, *:procedure*, *:value*, and *:unit*. These observations are extracted from the streaming data and are factorized; (b) The factorized relative humidity observations are semantified using ontologies.

6.2.3 Knowledge Graph Description Model

The knowledge graph is built on-demand by answering continuous SPARQL queries against data streams containing data produced from different IoT devices. A knowledge graph can be described in terms of the observed phenomenon and the property of phenomena being observed including a timestamp at which the observations are measured in the data stream. DESERT relies on Observed Tuple Molecule Templates to describe the type of the phenomenon and the property of the phenomenon sensed by observations and timestamps at which they are taken by IoT devices in data streams.

Definition 6.2.8 (Observed Tuple Molecule Template (OT-MT)). *An Observed Tuple Molecule Template (OT-MT) is a triple* $= \langle StreamIRI, C, TS \rangle$, *where:*


Time
obs1,Temperature,690,DewPoint,°F,17 ts1
obs2,Temperature,A02,AirTemp,°F,31 ts2
obs3,WindDirection,690,WindD,deg,135 ts3
obs4,Temperature,690,DewPoint,°F,20 ts4
obs5,Temperature,A02,AirTemp,°F,18 ts5
DS1
Sensors
obs6,Temperature,A02,AirTemp,°F,31 ts1
obs7,WindDirection,A02,WindD,deg,19 ts2
obs8,Temperature,A02,DewPoint,°F,31 ts8
obs9,Temperature,A02,AirTemp,°F,31 ts9
DS2
Sensors
obs10,WindDirection,690,WindD,deg,130 ts2
obs11,WindDirection,690,WindD,deg,135 ts9
DS3
Sensors
StreamIRI
Phenomenon, Property
TS
DS1 Temperature, DewPoint ts1,ts4
DS1 Temperature, AirTemp ts2,ts5
DS1 WindDirection , WindD ts3
DS2 Temperature, DewPoint ts8
DS2 Temperature, AirTemp ts1,ts9
DS2 WindDirection , WindD ts2
DS3 WindDirection , WindD ts2,ts9

(a) IoT Data Streams (b) **Description** of Knowledge Graph

Figure 6.5: **Example of Knowledge Graph Description**. (a) Three streams, DS_1, DS_2 and DS_3, of data with observations about *wind direction* and *temperature* measured at different timestamps; (b) an instance of the description, based on the concept of *OT-MT*, of the on-demand knowledge graph, built from the streams of data in Figure 6.5b.

- *StreamIRI - is an IRI to identify the data stream DS;*
- *C - is a phenomenon type such that the observed tuple* $obs^{ts} = (oo, ts)$, *where* $oo = \langle obs, C, proc, pp, uom, val \rangle$, *is in DS;*
- *PP - is a property of the phenomenon C observed within the observed tuple* $obs^{ts} = (oo, ts)$, *where* $oo = \langle obs, C, proc, PP, uom, val \rangle$, *is in DS;*
- *TS - is a finite set of timestamps such that the observed tuple* $obs^{ts} = (oo, ts)$, *where* $ts \in TS$, *is in DS*

An OT-MT is created for each distinct phenomenon and the property observed within the data stream, and includes the timestamps at which the observations about the phenomenon are measured along with the IRI of the data stream. Figure 6.5 illustrates the creation of OT-MTs describing the data about several weather phenomena and the observed property integrated into the knowledge graph which is built on-demand from the data streams in Figure 6.5a. An OT-MT is created for each phenomenon and its property being observed within each data stream. Three OT-MTs, for temperature and wind direction phenomena, and the corresponding properties are created within DS_1, stream IRI and timestamps of the observations are included in OT-MTs. Similarly, three distinct OT-MTs are created for DS_2 for temperature and window direction phenomenon, and the

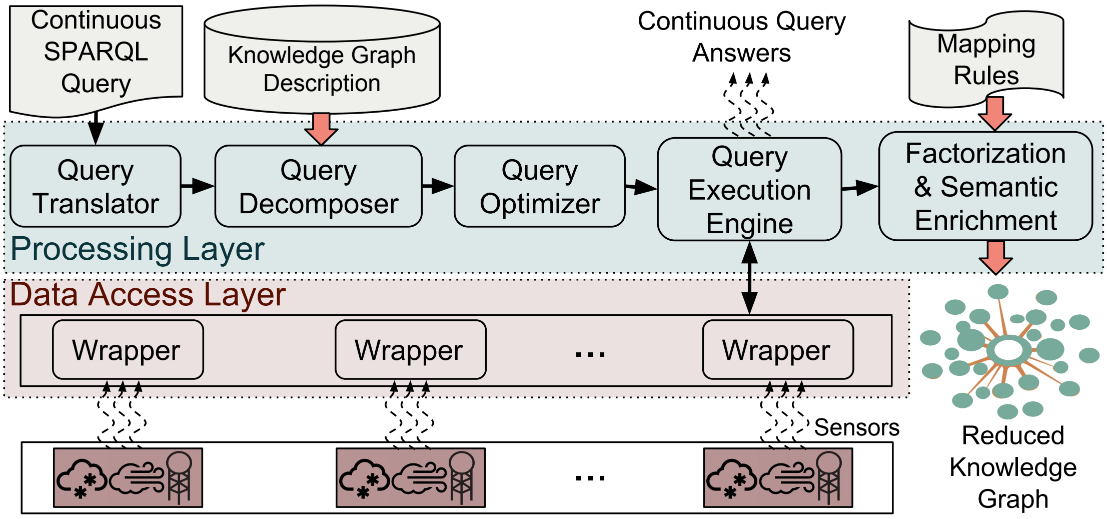


Figure 6.6: **The DESERT Architecture**. DESERT receives continuous SPARQL queries and produces continuous answers to these queries. Input queries are translated and decomposed into dynamic subqueries against the Reduced Knowledge Graph. Wrappers collect the stream data from various IoT devices. Sensor data that is not part of the Reduced Knowledge Graph yet gets semantified and factorized on-demand. The Query Optimizer and Execution Engine retrieve continuous query results from the Reduced Knowledge Graph.

stream IRI and timestamps of the observations are added. Only one OT-MT is created for DS_3 to describe the wind direction phenomenon observations observed at time $ts2$ and $ts9$.

6.3 The DESERT Architecture

Figure 6.6 depicts the architecture of DESERT; it comprises the following components:

Query Translator: implements query transformation rules and translates an input continuous SPARQL query, against a complete knowledge graph, into a continuous SPARQL query over a corresponding reduced knowledge graph containing the factorized streaming data. That is, the translated continuous SPARQL query includes a list of triple patterns that can find mappings from the reduced knowledge graph describing the factorized streaming data.

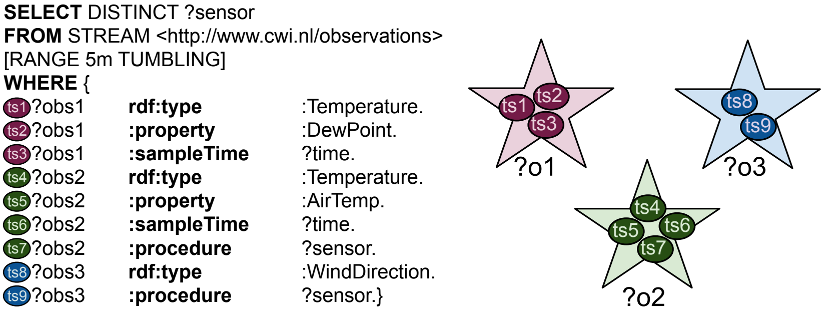
Query Decomposer: given the description of the knowledge graph, the *query decomposer* decomposes an input continuous SPARQL query into multiple simple dynamic (i.e., continuous) subqueries and determines the target source for the subqueries. The knowledge graph description is OT-MTs based and contains a list of *stream IRIs*, the phenomenon and property being observed within the stream, and the observations timestamps. Each input query is decomposed into multiple subqueries based on the type of observations required to answer the input query. A simple dynamic subquery is composed of a list of triple patterns that can provide the mappings to the most recent and relevant observations in the data stream and to the data integrated into the knowledge graph. Therefore, each subquery is against a particular type of observations that have been recently measured in the data stream. Then the IRIs of all the data streams, that contain the recently observed data about the observations type required to answer the dynamic subquery, are appended in the subquery. Figure 6.7 illustrates the decomposition of a continuous SPARQL query, composed of nine triple patterns, into three star-shaped subqueries [101] each around an observation phenomenon using OT-MTs from 6.5b. Answers to these subqueries are retrieved from the relevant data streams and knowledge graph and final query results are generated using the bushy-plan shown in Figure 6.7c. The dynamic subqueries are passed to the *query optimizer*.

Query Optimizer: generates an optimized query plan to integrate, over time, continuous answers of dynamic subqueries, generated by the query decomposer. Optimized plans are based on the star-shaped groups produced using OT-MTs as shown in Figure 6.7c.

Wrappers: provide an interface between DESERT and streams of data, and perform on-demand data retrieval from data streams. Complex implementations and heterogeneity of data streams remain hidden with the help of the *wrappers*. *Wrappers* convert the dynamic SPARQL subqueries into calls to data streams and retrieve the most recent observations, from the data streams, generated by IoT devices. Further, *wrappers* convert the retrieved data, from data streams, into the DESERT internal structures and pass these representations to the *query execution engine* component, which integrates these data representations to generate the continuous query results, and creates the knowledge graph.

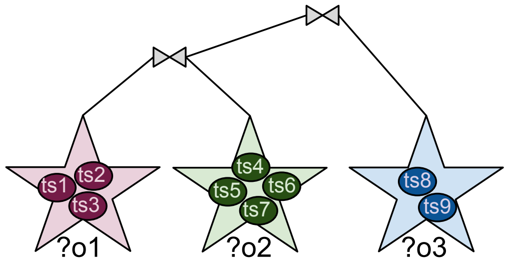
Query Execution Engine: is able to execute the optimized query plan and to integrate the continuous results of dynamic subqueries retrieved, continuously over time, using the customized *wrappers* in order to generate the answers for the input continuous SPARQL query. The answers to an input continuous SPARQL query are produced, over a period of time, within the streaming window size defined in the input continuous SPARQL query.

Factorization and Semantic Enrichment: receives a set of mapping rules to convert the input streaming data, retrieved from the wrappers, into the RDF knowledge graph. The *factorization and semantic enrichment* component continuously selects a portion, equal to the window size of the input continuous SPARQL



```
SELECT DISTINCT ?sensor
FROM STREAM <http://www.cwi.nl/observations>
[RANGE 5m TUMBLING]
WHERE {
ts1 ?obs1   rdf:type      :Temperature.
ts2 ?obs1   :property     :DewPoint.
ts3 ?obs1   :sampleTime   ?time.
ts4 ?obs2   rdf:type      :Temperature.
ts5 ?obs2   :property     :AirTemp.
ts6 ?obs2   :sampleTime   ?time.
ts7 ?obs2   :procedure    ?sensor.
ts8 ?obs3   rdf:type      :WindDirection.
ts9 ?obs3   :procedure    ?sensor.}
```

(a) Continuous SPARQL Query Q (b) **Star-shaped** subqueries



(c) **Execution Plan** for subqueries

Figure 6.7: **Query Decomposition**. (a) A continuous SPARQL query, over stream data, composed of 9 triple patterns that can be decomposed into star-shaped subqueries; (b) Three star-shaped subqueries around subjects in triple pattern describing different weather phenomena associated with three OT-MTs in Figure 6.5b; (c) A plan for star-shaped subqueries.

query, of the observation tuples received from the customized *wrappers*. The collected stream is factorized by integrating observation tuples with the same measurement unit and value in order to reduce the number of repeated measurement values and the corresponding data. Figure 6.4a illustrates the factorization of the stream in Figure 6.3a within the window $[ts1, ts11]$ with respect to the *relative humidity* observations that are required to answer the continuous SPARQL query in Figure 6.2a. Factorization integrates all measurements and observations with the same value into single measurement and observation RDF graphs, respectively. Once the observation and measurement RDF graphs have been constructed, the *factorization and semantic enrichment* component describes the semantics of the observation tuples using ontologies and integrates the semantified data into the knowledge graph. Figure 6.4b presents the knowledge graph generated by semantifying the factorized stream in Figure 6.4a. The factorized knowledge graph contains only data required to answer the SPARQL query.

6.4 Experimental Evaluation

In this chapter, we study effectiveness and efficiency of DESERT for streaming data generated at different speeds by IoT devices equipped with sensors. Particularly, we evaluate the performance in terms of throughput, and the impact on the size of the knowledge graph for different data stream speeds and window sizes. We consider three different configurations:

- *CSPARQL*: All types of entities in the data stream are semantified using conventional RDF stream processing implemented by the C-SPARQL engine.
- *onDS*: Only entities, within a streaming window, required for producing the answers of an input continuous SPARQL query are all semantified.
- *onDFS*: Only entities, within the streaming window, required for producing the answers of an input query are factorized, and then semantified.

Moreover, all these cases are evaluated in two dimensions, i.e., data stream speed and window size, by considering different combinations of uniformity and variation of the data stream speed and window size. We empirically assessed the following research questions: **ResearchQ1)** Is the size of knowledge graph impacted by the selectivity of the queries? **ResearchQ2)** Is query execution in DESERT affected by the the selectivity of queries? **ResearchQ3)** Is the performance of DESERT affected by the data stream speed and size of the streaming window? **ResearchQ4)** Are the continuous query answers equivalent in all three cases? The experimental configuration to evaluate these research questions is as follows:
Datasets: Experiments are conducted on datasets including observations of different climate phenomena, e.g., temperature, visibility, and precipitation, during the blizzard season in the United States of year 2003[2]. Several data streams with a particular velocity are generated, for 20 minutes and for 1 hour, from these datasets. The main characteristics of each data stream, generated for 20 minutes and 1 hour with different combinations of data stream speed and window size dimensions, are described in the next corresponding sections.
Queries: The SRBench-Version 0.9 queries[3] provide the testbed for our experimental evaluation. The continuous SPARQL queries range from simple queries with four triple patterns to complex one having up to fourteen triple patterns including UNION, OPTIONAL, and FILTER clauses. From the 17 continuous SPARQL queries only ten SELECT queries with OPTIONAL and UNION operators, aggregate modifiers like AVG, GROUP BY, and HAVING and FILTER

---

[2] Datasets can be downloaded from `http://wiki.knoesis.org/index.php/LinkedSensorData`

[3] `https://www.w3.org/wiki/SRBench`

clauses are included in our experimental testbed. Furthermore, each continuous SPARQL query is provided with one and ten minutes streaming window sizes.
Metrics: We report on the following metrics for each of the ten queries executed with different combinations of uniform and varying data stream speeds and window sizes: **a)** *Inverse Throughput (Inv.Throughput)* is the inverse of the number of answers produced per unit of time. **b)** *Knowledge Graph Size (KG.Size)* is the average size of the knowledge graph in megabytes in the given window. **c)** *Used Memory (Used.Memory)* is the average memory used by the system to generate answers in the given window. **d)** *Number of Normalized Triples (Norm.Triples)* is the number of RDF triples generated in the knowledge graph divided by the maximum number of triples in the three cases for the given window. **e)** *Inverse Percentage of Answers Completeness (Inv.AnsComp)* is the inverse of the percentage of produced answers completeness. For all metrics lower values are better.
Implementation: The experiments were performed on a Ubuntu 17.10 machine with CPU Intel Xeon(R) W-2133 CPU @ 3.06GHz x 12 and 64GB RAM. *Node-Red v0.18.4 (npm)*[4] is used to synthesize the streams of data generated by the IoT devices. A websocket interface passes the data streams from Node-Red to DESERT for processing. Afterwards, continuous SPARQL queries are run over the data streams in the three cases with different combinations of the data stream speeds and window sizes: (i) all types of entities in the data stream are semantified (*CSPARQL*), (ii) only entities facilitating the query execution are semantified within the streaming window (*onDS*), and (iii) the entities facilitating query execution are semantified as well as factorized within the streaming window (*onDFS*).

6.4.1 Performance with Uniform Data Stream Speed and Window Size

To evaluate the performance of DESERT over uniform data stream speed and window size, we executed ten SRBench Version 0.9 queries with ten minutes window size over a data stream, with 20 observations per second speed, for one hour.

Table 6.1: **Data Stream Description for Uniform Data Stream Speed and Window Size Dimensions**: Observations are collected from around 20,000 weather stations in the United States with twenty observations per second speed and ten minutes streaming window size.

ID	**Climate Event**	**Stream Rate (# Obs/sec)**	**Window Size (sec)**	**# Obs/Window**	**# Obs**
DS1	Blizzard	20	600	12,000	72,000

---

[4] `https://nodered.org/`

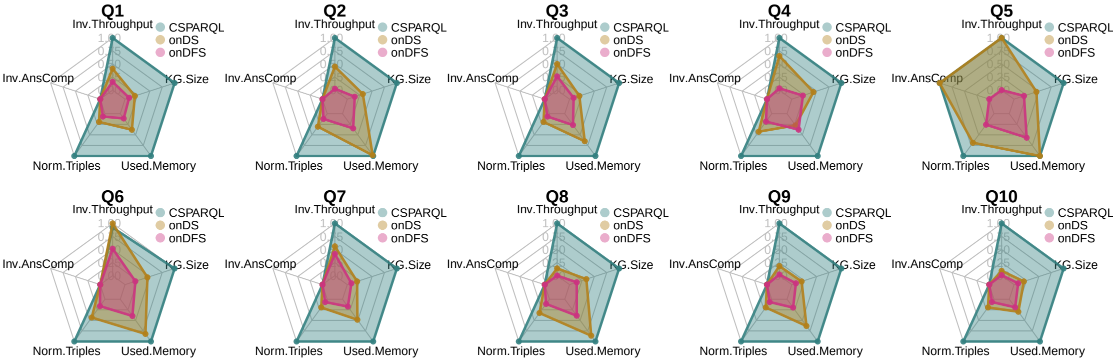


Figure 6.8: **Performance with Uniform Data Stream Speed and Window Size**. Continuous queries, with 10 min. window, are executed over IoT stream data, with 20 obs. per sec. speed, by **(i)** semantifying all entities in the stream, i.e., CSPARQL engine (*CSPARQL*), **(ii)** semantifying only the entities that satisfy the SPARQL queries, i.e., On-Demand Semantification (*onDS*) and **(iii)** factorizing and semantifying only the entities that satisfy the SPARQL queries, i.e., On-Demand Factorization and Semantification (*onDFS*).

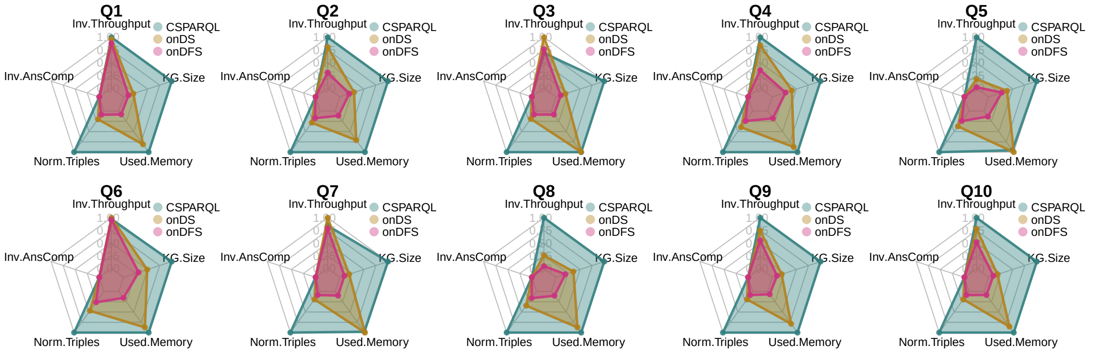
Table 6.1 describes the main characteristics of the data stream generated for one hour. Figure 6.8 compares the three cases (*CSPARQL*, *onDS*, *onDFS*) for queries Q1–Q10 with respect to *Inv.Throughput*, *KG.Size*, *Used.Memory*, *Norm.Triples*, and *Inv.AnsComp*, for ten minutes window, with twenty observations per second data stream speed. The interpretation of the metrics *Inv.Throughput*, *KG.Size*, *Used.Memory*, *Norm.Triples*, and *Inv.AnsComp* is, lower is better. Although for all queries the results are complete in all cases, we observe big differences in the inverse throughput values, as well as the knowledge graph size, the number of RDF triples produced and the used memory. Regarding knowledge graph size (*KG.Size*) and produced RDF triples (*Norm.Triples*) the advantages of on-demand semantification and on-demand factorization and semantification are more obvious. Compared with *CSPARQL*, we observed up to 80.14% and 80.60% savings for *onDS*, and up to 91.63% and 93.37% savings for *onDFS*, with respect to knowledge graph size and the number of RDF triples produced respectively, within the ten minutes window size. These results allow us to answer positively answer research question **ResearchQ1**, i.e., knowledge graph size can be significantly reduced using on-demand factorization and semantification techniques. Moreover, the results show that the inverse throughput in on-demand semantification is comparable to the full semantification (lower for most of the queries), whereas with both on-demand factorization and semantification the inverse throughput is reduced up to three orders of magnitude, e.g., Q8 and Q10. However, for the complex query Q5, we were able to retrieve answers only in the case of *onDFS* (for *CSPARQL* and *onDS* we got timeouts after one hour). Finally, the memory usage in *onDFS* is reduced compared to *onDS* and *CSPARQL*. Thus, research question **ResearchQ2** can be positively answered, i.e., the lower inverse throughput (i.e., inverse of the number of retrieved answers in a unit of time) shows that the throughput using DESERT can be improved, especially when both semantification and factorization are applied on-demand. Further, the equivalent values, zero for *onDFS*, *onDS* and *CSPARQL* for the inverse percentage of answers completeness, allows us to positively answer research question **ResearchQ4**, i.e., the returned results are complete, in all cases.

6.4.2 Performance with Uniform Data Stream and Varying Window Size

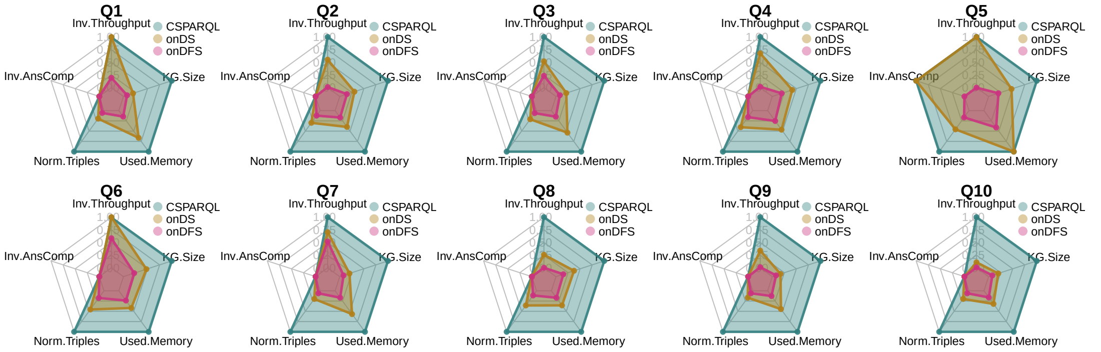
To evaluate the performance of DESERT over uniform data stream speed and varying window size, we executed ten SRBench-Version 0.9 queries with one and ten minutes windows size over a data stream, with ten observations per second speed, for twenty minutes and one hour, respectively. Table 6.2 describes the main characteristics of the data stream generated for twenty minutes and one hour. Figures 6.9a and 6.9b compare the three aforementioned cases (*CSPARQL*, *onDS*, *onDFS*) for queries Q1–Q10 with respect to *Inv.Throughput*, *KG.Size*, *Used.Memory*, *Norm.Triples*, and *Inv.AnsComp*, for one and ten minutes windows, with ten observations per second data stream speed. All the metrics *KG.Size*, *Inv.Throughput*, *Used.Memory*, *Norm.Triples*, and *Inv.AnsComp* are lower the better. For all queries, the results are complete in all cases; however, differences can be observed whenever the window size is increased and the stream is larger. Consistently, the knowledge graph size, the number of RDF triples produced, and the used memory increase as well. Within one minute window, the knowledge graph size and number of produced RDF triples are reduced by *onDFS* and *onDS*. Nevertheless,*onDFS* outperforms *CSPARQL* and *onDS* when larger streams are observed, i.e., the window size is increased. Compared with *CSPARQL*, we observed up to 92.53% and 94.34% savings for *onDFS*, with respect to knowledge

Table 6.2: **Data Stream Description for Uniform Data Stream Speed and Varying Window Size Dimensions**: Observations are collected from around 20,000 weather stations in the United States with ten observations per second, and one and ten minutes windows size.

ID	**Climate Event**	**Stream Rate (# Obs/sec)**	**Window Size (sec)**	**# Obs/Window**	**# Obs**
DS2	Blizzard	10	60	600	12,000
DS3	Blizzard	10	600	6,000	36,000



(a) Continuous SPARQL queries execution with one minute window over 10 obs.per.sec. data stream for 20 minutes.



(b) Continuous SPARQL queries execution with ten minutes window over 10 obs.per.sec. data stream for one hour.

Figure 6.9: **Performance with Uniform Data Stream Speed and Varying Window Size**. Continuous queries, with one and 10 minutes windows, are executed over IoT stream data, with 10 obs. per sec. speed, by **(i)** semantifying all entities in the stream, i.e., the C-SPARQL engine (*CSPARQL*), **(ii)** semantifying only the entities that satisfy the queries, i.e., On-Demand Semantification (*onDS*) and **(iii)** factorizing and semantifying only the entities that satisfy the queries, i.e., On-Demand Factorization and Semantification (*onDFS*). *onDFS* outperforms *CSPARQL* and *onDS* in all the dimensions for selective queries over large streams. In a small stream (one minute window over 10 obs.per sec.), the three engines exhibit the same throughput for Q1, Q6, and Q7.

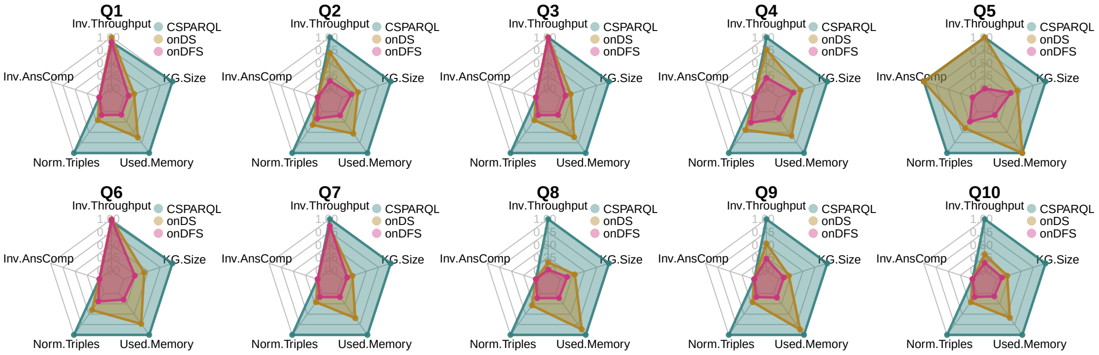
graph size and the number of RDF triples produced respectively, with increasing window size. Moreover, the results report the inverse throughput in *onDFS* and *onDS* is comparable to *CSPARQL* (lower for most of the queries), whereas with increasing window size, *onDFS* improvements can be seen in the inverse throughput values reducing up to two orders of magnitude, e.g., Q2, Q4, Q8, Q9, and Q10. However, for the complex query Q5, we were able to retrieve answers only in the case of *onDFS* both for one minute and ten minute windows, for *onDS* Q5 results are obtained only for one minute window and for ten minute window Q5 timed out, whereas for *CSPARQL* we got timeouts for one and ten minutes windows after twenty minutes and one hour, respectively. Finally, the memory usage in *onDFS* is reduced compared to *onDS* and *CSPARQL* with increasing window size. These results allow us to positively answer the research question **ResearchQ3** regarding the size of the streaming window, i.e., more savings in terms of knowledge graph size and better query execution time can be achieved using *onDFS*, with the increasing streaming window size.

6.4.3 Performance with Varying Data Stream and Uniform Window Size

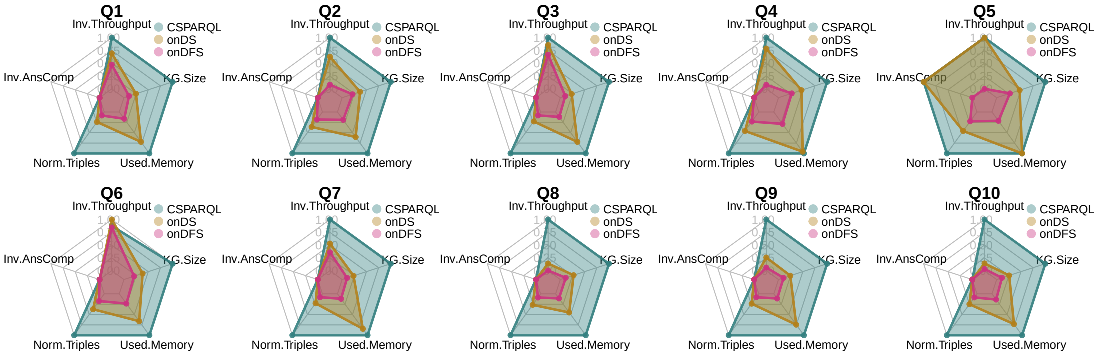
To evaluate the performance of *DESERT* over varying data stream speed and uniform window size, we executed ten SRBench-Version 0.9 queries with one minute window size over two data streams, with 25 observations per second data stream speed, for twenty minutes. Table 6.3 describes the main characteristics of the data stream generated for twenty minutes. Figures 6.10a and 6.10b compare the three aforementioned cases (*CSPARQL*, *onDS*, *onDFS*) for queries Q1–Q10 with respect to *Inv.Throughput*, *KG.Size*, *Used.Memory*, *Norm.Triples*, and *Inv.AnsComp*, for one and ten minutes windows, with ten observations per second data stream speed. All the metrics *Inv.Throughput*, *KG.Size*, *Used.Memory*, *Norm.Triples*, and *Inv.AnsComp*, are lower the better. For all queries, the results are complete in all cases; however, with increasing data stream speed, we observe big differences in the inverse throughput values, as well as the knowledge graph size, the number of RDF triples produced and the used memory. Up to 90.36% and 92.53% savings are observed for *onDFS* in knowledge graph size and number of triples produced, respectively, with increasing data stream speed. Moreover, in-

Table 6.3: **Data stream Description for Varying Data Stream Speed and Uniform Window Size Dimensions**: Observations are collected from around 20,000 weather stations in the United States with 20 and 50 observations per second, and one minute window size.

ID	**Climate Event**	**Stream Rate (# Obs/sec)**	**Window Size (sec)**	**# Obs/Window**	**# Obs**
DS4	Blizzard	20	60	1,200	24,000
DS5	Blizzard	50	60	3,000	60,000



(a) Continuous SPARQL queries execution with one min. window over 20 obs.per.sec. data stream for 20 minutes.



(b) Continuous SPARQL queries execution with one min. window over 50 obs.per.sec. data stream for 20 minutes.

Figure 6.10: **Performance with Varying Data Stream Speed and Uniform Window Size**. Continuous queries, with one minute window, are executed over IoT stream data, with 20 and 50 obs. per sec. speeds, by **(i)** semantifying all entities in the stream, i.e., CSPARQL engine (*CSPARQL*), **(ii)** semantifying only the entities that satisfy the queries, i.e., On-Demand Semantification (*onDS*) and **(iii)** factorizing and semantifying only the entities that satisfy the queries, i.e., On-Demand Factorization and Semantification (*onDFS*). *onDFS* outperforms *CSPARQL* and *onDS* in all the dimensions for selective queries over large streams. In a small stream (one minute window over 20 obs.per sec.), the three engines exhibit the same throughput for Q1, Q6, and Q7.

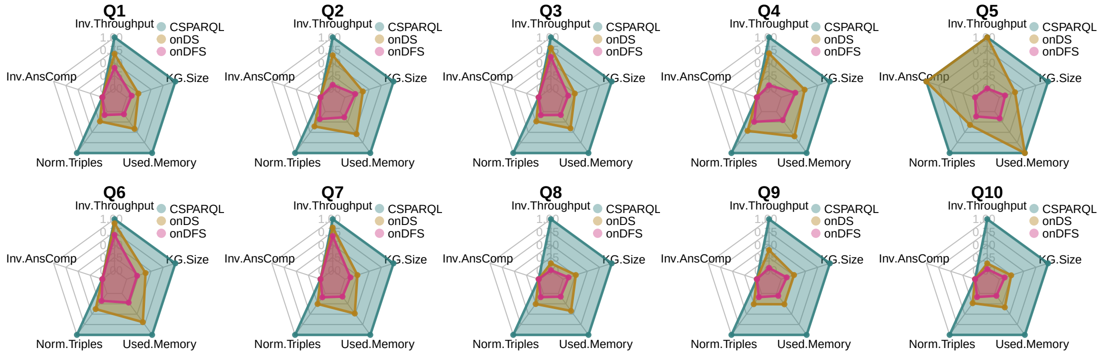
verse throughput for *onDS* is relatively lower than the *CSPARQL*, while for *onDFS* the inverse throughput reduced up to three orders of magnitude, e.g., Q2 and Q4, with increasing data stream speed. Nonetheless, for the complex query Q5, we were able to retrieve answers only in the case of *onDFS*, whereas for *CSPARQL* and *onDS*, Q5 timed out both for twenty and fifty observations per second data stream speed. Finally, the memory usage in *onDFS* remains lower than *onDS* and *CSPARQL* with increasing data stream speed. These results allow us to positively answer **ResearchQ3** regarding the data stream speed, i.e., more savings in terms of knowledge graph size and better query execution time can be achieved using *onDFS* whenever the size of the IoT stream data is increased.

6.4.4 Performance with Varying Data Stream and Window Size

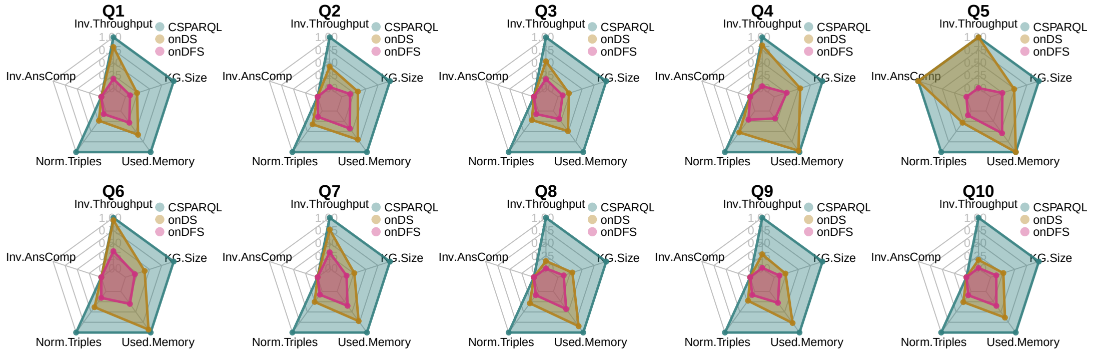
To evaluate the performance of *DESERT* over varying data stream speed and window size, we executed ten SRBench-Version 0.9 queries with one minute window over an data stream with seventy observations per second speed and ten minutes window size over data stream with fifty observations per second speed, for twenty minutes and one hour, respectively. Table 6.4 describes the main characteristics of the data stream generated for twenty minutes and one hour. Figures 6.11a and 6.11b compare the three aforementioned cases (*CSPARQL*, *onDS*, *onDFS*) for queries Q1–Q10 with respect to *Inv.Throughput*, *KG.Size*, *Used.Memory*, *Norm.Triples*, and *Inv.AnsComp*, for one and ten minutes windows, with ten observations per second data stream speed. All the metrics *KG.Size*, *Inv.Throughput*, *Used.Memory*, *Norm.Triples*, and *Inv.AnsComp*, are lower the better. As in previous experiments, the query answers are complete in all cases and varying data stream speed and window size impact on the engine performance. First, we observe big differences in the inverse throughput values, as well as the knowledge graph size, the number of RDF triples produced and the used memory. More savings are observed in knowledge graph size and number of triples with ten minute window and fifty observations per second data stream speed. Moreover, inverse throughput for *onDS* is relatively lower than the *CSPARQL* for most of the queries, while for *onDFS* the inverse throughput is less than 25% for all queries with ten minute window and fifty observations per second data stream speed. For

Table 6.4: **Data Stream Description for Varying Data Stream Speed and Window Size Dimensions**: Observations are collected from around 20,000 weather stations in the United States with 70 and 50 obs. per sec. speed within one and 10 minutes windows, respectively.

ID	**Climate Event**	**Stream Rate (# Obs/sec)**	**Window Size (sec)**	**# Obs/Window**	**# Obs**
DS6	Blizzard	70	60	4,200	84,000
DS7	Blizzard	50	600	30,000	180,000



(a) Continuous SPARQL queries execution with one min. window over 70 obs.per.sec. data stream for 20 minutes.



(b) Continuous SPARQL queries execution with ten min. window over 50 obs.per.sec. data stream for one hour.

Figure 6.11: **Performance with Varying Data Stream Speed and Window Size**. Continuous queries, with one and 10 min. windows, are executed over IoT stream data, with 70 and 50 obs.per.sec. speeds, respectively, by **(i)** semantifying all entities in the stream, i.e., the C-SPARQL engine (*CSPARQL*), **(ii)** semantifying only the entities that satisfy the queries, i.e., On-Demand Semantification (*onDS*) and **(iii)** factorizing and semantifying only entities that satisfy queries, i.e., On-Demand Factorization and Semantification (*onDFS*). *onDFS* outperforms *CSPARQL* and *onDS* in all the dimensions over large streams.

most of the queries inverse throughput is significantly reduced whenever the size of the stream is large. For example, *onDFS* outperforms *CSPARQL* and *onDS* in terms of the inverse throughput when the window size is ten minutes and fifty observations are collected per second. Nevertheless, in smaller streams, e.g., one minute window with seventy observations per second, this difference in the inverse throughput is not always observed, e.g., Q1, Q3, Q6, and Q7. For the complex query Q5, only *onDFS* produces answers, whereas *CSPARQL* and *onDS* timed out within both one and ten minutes windows with seventy and fifty observations per second data stream speed, respectively. Finally, the memory usage in *onDFS* remains lower than *onDS* and *CSPARQL* during all executions. These results suggest that the on-demand knowledge graph techniques implemented by *onDFS* provide a scalable solution for continuous query processing, supporting thus a positive answer of **ResearchQ3**, i.e., *DESERT* performance is impacted by the size of the IoT stream data.

6.5 Summary

Big data and streaming nature of IoT data imposes several challenges that need to be addressed in order to provide efficient and scalable management and processing of IoT data. We address these challenges and focus on the problems of describing the meaning of IoT data using ontologies and integrating this data in a knowledge graph. We devise DESERT, a SPARQL query engine able to on-Demand factorizE and Semantically Enrich stReam daTa in a knowledge graph. Resulting knowledge graphs model the semantics or meaning of merged data in terms of entities that satisfy the SPARQL queries and relationships among those entities; thus, only data required for query answering is included in the knowledge graph. We empirically evaluate the results of DESERT on SRBench, a benchmark of Streaming RDF data, using different combinations of data stream speed and windows size. The experimental results suggest that DESERT allows for speeding up query execution while the size of the knowledge graphs remains relatively low.

Chapter 7

Conclusion and Future Directions

Observational data comprise observations that are expressed as measurements whose values can be repeated several times in a sampling frame, resulting in a considerable increase in streaming and historical observational data volume. With the maturing of semantic technologies and their increasing industrial use, scalability, performance, and robustness progressively shift into the focus. RDF Knowledge graphs provide descriptions of observations from related sampling frames or sensors at multiple points in time, e.g., patient medical records or climate sensor data. One particular important dimension of RDF knowledge graphs is to represent observations over time, i.e., time series. Examples of such data are sensor measurements, clinical study data, stock prices, logistics, and traffic data. Also, for making RDF knowledge graphs representations more suitable to be applied in Big Data scenarios the velocity and volume dimensions have to be better supported. Representing observational data and their meaning directly in RDF will result in a significant expansion of the data volume due to repetition. In this thesis, we study the problem of identifying redundancies in observational data semantically described in RDF knowledge graphs and present compact representations of RDF knowledge graphs. Furthermore, we address the problem of efficient query processing over historical semantic sensor data, and propose compact representations of historical semantic sensor data that can be managed by existing RDF and Big Data stores. Moreover, we investigate the problem of on-demand knowledge graph creation from streaming data, and develop a continuous query processing framework. The proposed approach models, in knowledge graphs, the meaning of streaming data in terms of observation entities which are needed to answer an input continuous query.

7.1 Revising the Research Questions

RQ1: What are the criteria to identify frequent star patterns?

Chapter 4 presents computational methods to identify frequent star patterns and to generate a *factorized RDF graph*, with a minimized number of frequent star patterns. A frequent star pattern contains class entities linked to the objects or other resources using labeled edges annotated with properties in the class. These frequent star patterns introduce redundancy in terms of edges and nodes. Our proposed computational methods implement the frequent star pattern detection algorithm based on search space pruning techniques to identify the classes and properties involved in frequent star patterns. Furthermore, the proposed factorization techniques generate compact representation of RDF graphs, *factorized RDF graph*, by replacing a frequent star pattern with a compact RDF molecule, composed of a surrogate entity connected to the object in the frequent star pattern using the labeled edges annotated with relevant properties. We empirically study the effectiveness of the frequent star pattern detection algorithm to identify class and properties involved in the frequent star pattern. Furthermore, we evaluate the impact of the factorization techniques on the gradually increasing RDF graphs size and different combinations of class properties. Experimental results suggest that the proposed computational methods successfully identify the class properties involved in the frequent star patterns and remove redundancy caused by these frequent star patterns. For the best set of properties, identified by the frequent star pattern detection algorithm, the RDF graph size is reduced by up to 66.56%. Our work broadens the repertoire of techniques for representing and storing knowledge graphs by providing RDF graph compression techniques which exploit the semantics encoded in the data; these techniques generate compact representations of RDF graphs to help improving query processing over RDF graphs without requiring a customized engine. Our work contributes to the crucial knowledge graph representation and provides the basics for further development of the efficient processing techniques over the compact knowledge graphs.

RQ2: How can efficient representations be exploited to manage historical semantic sensor data?

Chapter 5 presents compact RDF representations for semantic sensor data to reduce data redundancy without losing any encoded information, enhance the performance of diverse RDF implementations, ensure correctness of query answers, and preserve complexity of query processing tasks. Furthermore, tabular representations for a loss-less large-scale storage of factorized semantic sensor data are presented. A factorization algorithm transforms original observations and measurements to a more compact representation where data redundancy is reduced. Additionally, query rewriting rules and a query re-writing algorithm are presented. The query rewriting algorithm exploits the rewriting rules to rewrite SPARQL queries against factorized RDF graphs, and speeds up query execution time. The factorized observations and measurements are also exploited to produce tabular representations for factorized RDF graphs utilizing Parquet tables. We empirically evaluate the effectiveness of the proposed factorization techniques and results confirm that exploiting semantics encoded in semantic sensor data allow for reducing redundancy by up to 57.96%, while the time taken by the process of factorizing RDF data is less than 50% of loading time for the original RDF data in the state-of-the-art RDF stores. Moreover, for all RDF datasets, the loading time for factorized RDF data is reduced by more than 45% of the loading time of original RDF data in native RDF stores. Also, we evaluated the impact of proposed compact representations on the diverse implementations available for RDF data, i.e., native RDF implementations, non-native large-scale tabular based implementations, and centralized RDF data accessible via SPARQL endpoints. Our experiments confirm the efficiency of the queries generated by the *CSSD* approach over several RDF implementations. In summary, factorization techniques provide efficient RDF representations that are able to solve the problem RDF data redundancy and enhance the performance of query engines over diverse RDF implementations.

RQ3: How can on-demand knowledge graph building reduce the size of the streaming observational data?

Chapter 6 presents on-demand factorization and semantification techniques for IoT stream data. The proposed techniques have been implemented in DESERT, a continuous SPARQL query engine. DESERT is able to create a knowledge graph on-demand by integrating IoT data collected from heterogeneous stream data sources, with different data generation speeds. Furthermore, DESERT exploits semantics encoded within the IoT stream data to generate knowledge graphs. We have explained the retrieval of stream data based on input continuous SPARQL query using customized wrappers; we also presented how on-demand factorization and semantification techniques can be used to reduce the size of the resulting knowledge graph. We evaluated the DESERT framework with various combinations of uniform and varying data stream speed and streaming window size dimensions. The results of the empirical evaluation suggest that DESERT is able to effectively reduce the size of the integrate knowledge graph compared to the conventional semantification techniques. The experiments also provide evidence that in comparison to the existing RDF stream processing engines, the on-demand factorization and semantification techniques implemented in DESERT are able to integrate IoT stream data in a knowledge graph efficiently and speed up query execution. Savings are significant in all the dimensions, in large streams of data produced by high-speed sensors or observed during large periods of time.

RQ4: How can on-demand knowledge graph building speed up query processing?

Chapter 6 implements efficient query processing techniques for streaming observational data in DESERT, a SPARQL query engine able to on-Demand factorizE and Semantically Enrich stReam daTa. DESERT has been implemented on top of C-SPARQL engine [15], an RDF engine for continuous SPARQL queries. DESERT receives a continuous SPARQL query and executes query over the heterogeneous observational data sources and produces knowledge graphs on-demand. We have empirically evaluated DESERT on SRBench[107], a state-of-the-art benchmark for continuous query processing of SPARQL queries; observed results evidence that knowledge graphs created by DESERT are able to both speed up query processing and reduce the knowledge graph size.

7.2 Limitations

Despite the overall achieved research objectives, we acknowledge that there are limitations of this research work which have not been covered in the scope of the thesis. Firstly, the proposed factorization techniques are worthy if the dataset under consideration has the characteristics of observational data, i.e., a large number of redundancies occur in the data. These redundancies in the dataset generate frequent star patterns when these datasets and the knowledge encoded in the data are described in knowledge graphs using Semantic Web technologies. In case, the number of frequent star patterns is not sufficient the proposed techniques generate overhead in terms of knowledge graph size and processing. Secondly, DESERT relies on the blocking SPARQL operators implemented by C-SPARQL; in consequence, query answers are not produced incrementally. These limitations need to be addressed in future research work. In order to maximize the scope of factorization techniques certain heuristics can be adopted, e.g., discretization of measurement values improves the savings and defining a canonical ontology will enable a joint factorization of multiple datasets that are described using various ontologies. Moreover, multi-query optimizations using non-blocking operators can be implemented in order to produce query answers incrementally from data streams.

7.3 Future Directions

Based on our findings, and the contributions made in this thesis, we now present some of the future directions of this work for the research community:

- Exploit parallel processing frameworks to efficiently find frequent star patterns.
- Exploit parallel processing frameworks for the RDF graph factorization to efficiently minimize the frequent graph patterns.
- Performance evaluation of the proposed RDF and tabular-based representations over federated engines.
- Extend the streaming query optimization techniques over on-demand factorization and semantification of streaming observational data by implementing multi-query optimization approaches based on the concept of streaming windows and the multiple continuous SPARQL queries.

7.4 Closing Remarks

The growing amount of streaming and historical data generated from diverse IoT devices demands efficient data integration approaches. These approaches should be able to transform such big and streaming data into actionable knowledge for decision making. Transforming IoT data into actionable knowledge requires novel and scalable techniques for enabling not only data streams semantification, but also for efficient large-scale semantic data integration, exploration, and discovery. In this thesis, we have shown that the factorized representations of data described in RDF knowledge graphs reduce the size of the data while all the encoded information is preserved. Moreover, these factorized RDF representations enhance query processing over diverse native and non-native RDF frameworks. In addition, on-demand factorization and semantification techniques are able to efficiently manage the streaming data. These results suggest that the techniques proposed in this thesis provide an efficient solution for supporting IoT and facilitating the management of IoT data and implementation of data-driven approaches.

Bibliography

[1] Daniel J Abadi et al. “The design of the borealis stream processing engine.” In: *Cidr*. Vol. 5. 2005. 2005, pp. 277–289.

[2] Daniel Abadi, Samuel Madden, and Miguel Ferreira. “Integrating compression and execution in column-oriented database systems”. In: *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. ACM. 2006, pp. 671–682. DOI: `10.1145/1142473.1142548`.

[3] Maribel Acosta, Maria-Esther Vidal, and York Sure-Vetter. “Diefficiency metrics: measuring the continuous efficiency of query processing approaches”. In: *International Semantic Web Conference*. Springer. 2017, pp. 3–19.

[4] Maribel Acosta et al. “ANAPSID: An Adaptive Query Processing Engine for SPARQL Endpoints”. In: *ISWC*. 2011. DOI: `10.1007/978-3-642-25073-6_2`.

[5] Muhammad Intizar Ali, Feng Gao, and Alessandra Mileo. “CityBench: a configurable benchmark to evaluate RSP engines using smart city datasets”. In: *International Semantic Web Conference*. Springer, 2015, pp. 374–389.

[6] David Allen et al. “Understanding Trolls with Efficient Analytics of Large Graphs in Neo4j”. In: *BTW 2019* (2019). DOI: `10.18420/btw2019-23`.

[7] Sandra Alvarez-Garcia et al. “A succinct data structure for self-indexing ternary relations”. In: *Journal of Discrete Algorithms* 43 (2017), pp. 38–53. DOI: `10.1016/j.jda.2016.10.002`.

[8] Sandra Álvarez-García et al. “Compressed k2-triples for full-in-memory RDF engines”. In: *arXiv preprint arXiv:1105.4004* (2011). URL: `http://arxiv.org/abs/1105.4004`.

[9] Arvind Arasu et al. “Stream: The stanford stream data manager”. In: *IEEE Data Eng. Bull.* 26.1 (2003), pp. 19–26.

[10] Marcelo Arenas, Claudio Gutierrez, and Jorge Pérez. “Foundations of RDF databases”. In: *Reasoning Web. Semantic Technologies for Information Systems*. Springer, 2009, pp. 158–204. DOI: `10.1007/978-3-642-03754-2_4`.

[11] Sören Auer et al. “Towards a Knowledge Graph for Science”. In: *Proceedings of the 8th International Conference on Web Intelligence, Mining and Semantics, WIMS 2018*. 2018. URL: `https://doi.org/10.1145/3227609.3227689`.

[12] Yijian Bai et al. “A data stream language and system designed for power and extensibility”. In: *Proceedings of the 15th ACM international conference on Information and knowledge management*. ACM. 2006, pp. 337–346.

[13] Nurzhan Bakibayev, Dan Olteanu, and Jakub Zavodny. "FDB: A Query Engine for Factorised Relational Databases". In: *PVLDB* 5.11 (2012), pp. 1232–1243.

[14] Nurzhan Bakibayev et al. "Aggregation and Ordering in Factorised Databases". In: *PVLDB* 6.14 (2013), pp. 1990–2001.

[15] Davide Francesco Barbieri et al. "An execution environment for C-SPARQL queries". In: *Proceedings of the 13th International Conference on Extending Database Technology.* ACM. 2010, pp. 441–452.

[16] Davide Francesco Barbieri et al. "C-SPARQL: a continuous query language for RDF data streams". In: *International Journal of Semantic Computing* 4.01 (2010), pp. 3–25.

[17] Davide Francesco Barbieri et al. "Querying rdf streams with c-sparql". In: *ACM SIGMOD Record* 39.1 (2010), pp. 20–26.

[18] Payam Barnaghi et al. "Sense and sens' ability: Semantic data modelling for sensor networks". In: *Conference Proceedings of ICT Mobile Summit 2009.* 2009.

[19] Christian Bizer, Tom Heath, and Tim Berners-Lee. "Linked data: The story so far". In: *Semantic services, interoperability and web applications: emerging concepts.* IGI Global, 2011, pp. 205–227. DOI: `10.4018/jswis.2009081901`.

[20] Kyoungsoo Bok et al. "Provenance compression scheme based on graph patterns for large RDF documents". In: *The Journal of Supercomputing* (2019), pp. 1–23.

[21] Peter A Boncz, Marcin Zukowski, and Niels Nes. "MonetDB/X100: Hyper-Pipelining Query Execution." In: *Cidr.* Vol. 5. 2005, pp. 225–237. URL: `http://cidrdb.org/cidr2005/papers/P19.pdf`.

[22] Robert K Brayton. "Factoring logic functions". In: *IBM Journal of research and development* 31.2 (1987), pp. 187–198.

[23] Nieves R Brisaboa, Susana Ladra, and Gonzalo Navarro. "k2-trees for compact web graph representation". In: *International Symposium on String Processing and Information Retrieval.* Springer. 2009, pp. 18–30. DOI: `10.1007/978-3-642-03784-9\_3`.

[24] Jim Chase. *The Evolution of the Internet of Things (White Paper).* Texas Instruments. 2013.

[25] Edgar F Codd. "Further normalization of the data base relational model". In: *Data base systems* (1972), pp. 33–64.

[26] Diego Collarana. "A Semantic Integration Approach for Building Knowledge Graphs On-Demand". In: *International Conference on Web Engineering.* Springer. 2017, pp. 575–583.

[27] Diego Collarana et al. "Semantic Data Integration for Knowledge Graph Construction at Query Time". In: *11th IEEE International Conference on Semantic Computing, ICSC.* 2017, pp. 109–116.

[28] Michael 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. DOI: `10.1016/j.websem.2012.05.003`.

[29] George P Copeland and Setrag N Khoshafian. "A decomposition storage model". In: *Acm Sigmod Record.* Vol. 14. 4. ACM. 1985, pp. 268–279. DOI: `10.1145/318898.318923`.

[30] Daniele Dell'Aglio and Emanuele Della Valle. "Incremental Reasoning on RDF Streams". In: *Linked Data Management.* 2014, pp. 413–435.

[31] AnHai Doan, Alon Halevy, and Zachary Ives. *Principles of data integration.* Elsevier, 2012.

[32] Jin-Hang Du et al. "HadoopRDF: A scalable semantic data analytical engine". In: *International Conference on Intelligent Computing.* Springer. 2012, pp. 633–641.

[33] Mohammed Elseidy et al. "Grami: Frequent subgraph and pattern mining in a single large graph". In: *Proceedings of the VLDB Endowment* 7.7 (2014), pp. 517–528.

[34] Kemele M Endris et al. "MULDER: querying the linked data web by bridging RDF molecule templates". In: *International Conference on Database and Expert Systems Applications.* Springer. 2017, pp. 3–18.

[35] Patrick Ernst, Amy Siu, and Gerhard Weikum. "Knowlife: a versatile approach for constructing a large knowledge graph for biomedical sciences". In: *BMC bioinformatics* 16.1 (2015), p. 157. DOI: `10.1186/s12859-015-0549-5`.

[36] Javier D. Fernández, Alejandro Llaves, and Óscar Corcho. "Efficient RDF Interchange (ERI) Format for RDF Data Streams". In: *The Semantic Web - ISWC 2014 - 13th International Semantic Web Conference, Riva del Garda, Italy, October 19-23, 2014. Proceedings, Part II.* 2014, pp. 244–259. DOI: `10.1007/978-3-319-11915-1\_16`.

[37] Javier D Fernández et al. "Binary RDF representation for publication and exchange (HDT)". In: *Web Semantics: Science, Services and Agents on the World Wide Web* 19 (2013), pp. 22–41. DOI: `10.1016/j.websem.2013.01.002`.

[38] Lianli Gao, Michael Bruenig, and Jane Hunter. "Semantic-based detection of segment outliers and unusual events for wireless sensor networks". In: *arXiv preprint arXiv:1411.2188* (2014).

[39] Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi. *Data Stream Management: Processing High-Speed Data Streams.* Springer, 2016.

[40] Aditya Gaur et al. "Smart city architecture and its applications based on IoT". In: *Procedia computer science* 52 (2015), pp. 1089–1094.

[41] Lukasz Golab and M Tamer Özsu. "Processing sliding window multi-joins in continuous queries over data streams". In: *Proceedings of the 29th international conference on Very large data bases-Volume 29.* VLDB Endowment. 2003, pp. 500–511.

[42] Faming Gong et al. "Neo4j graph database realizes efficient storage performance of oilfield ontology". In: *PloS one* 13.11 (2018), e0207595. URL: `https://doi.org/10.1371/journal.pone.0207595`.

[43] Irlán Grangel-González et al. "Knowledge Graphs for Semantically Integrating Cyber-Physical Systems". In: *Database and Expert Systems Applications - 29th International Conference.* 2018. DOI: `10.1007/978-3-319-98809-2\_12`.

[44] Andrey Gubichev and Manuel Then. "Graph Pattern Matching: Do We Have to Reinvent the Wheel?" In: *Proceedings of Workshop on GRAph Data management Experiences and Systems.* ACM. 2014, pp. 1–7. DOI: `10.1145/2621934.2621944`.

[45] Alon Y Halevy. "Answering queries using views: A survey". In: *The VLDB Journal* 10.4 (2001), pp. 270–294.

[46] Cory Andrew Henson et al. “An ontological representation of time series observations on the Semantic Sensor Web”. In: (2009).

[47] Richard Hull. “Managing semantic heterogeneity in databases: a theoretical prospective”. In: *Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. ACM. 1997, pp. 51–61.

[48] S Idreos et al. “Monetdb: Two decades of research in column-oriented database”. In: *IEEE Data Engineering Bulletin* (2012).

[49] Sohail Jabbar et al. “Semantic interoperability in heterogeneous IoT infrastructure for healthcare”. In: *Wireless Communications and Mobile Computing* 2017 (2017).

[50] D UUman Jeffrey. *Principles of database and knowledge-base systems*. 1989.

[51] Amit Krishna Joshi, Pascal Hitzler, and Guozhu Dong. “Logical linked data compression”. In: *Extended Semantic Web Conference*. Springer. 2013, pp. 170–184. DOI: `10.1007/978-3-642-38288-8\_12`.

[52] Farah Karim, Maria-Esther Vidal, and Sören Auer. “Efficient Processing of Semantically Represented Sensor Data.” In: *WEBIST*. 2017, pp. 252–259.

[53] Farah Karim, Maria-Esther Vidal, and Sören Auer. “Factorization Techniques for Longitudinal Linked Data”. In: ().

[54] Farah Karim et al. “DESERT: a continuous SPARQL query engine for on-demand query answering”. In: *International Journal of Semantic Computing* 12.03 (2018), pp. 373–397.

[55] Farah Karim et al. “Large-scale storage and query processing for semantic sensor data”. In: *Proceedings of the 7th International Conference on Web Intelligence, Mining and Semantics*. ACM. 2017, p. 8. DOI: `10.1145/3102254.3102260`.

[56] Farah Karim et al. “Semantic enrichment of IoT stream data on-demand”. In: *2018 IEEE 12th International Conference on Semantic Computing (ICSC)*. IEEE. 2018, pp. 33–40.

[57] Vaibhav Khadilkar et al. “Jena-HBase: A distributed, scalable and efficient RDF triple store”. In: *Proceedings of the 11th International Semantic Web Conference Posters & Demonstrations Track, ISWC-PD*. Vol. 12. Citeseer. 2012, pp. 85–88.

[58] Jeong-Hee Kim et al. “Building a service-oriented ontology for wireless sensor networks”. In: *Computer and Information Science, 2008. ICIS 08. Seventh IEEE/ACIS International Conference on*. IEEE. 2008, pp. 649–654.

[59] Manolis Koubarakis and Kostis Kyzirakos. “Modeling and Querying Metadata in the Semantic Sensor Web: The Model stRDF and the Query Language stSPARQL”. In: *The Semantic Web: Research and Applications, 7th Extended Semantic Web Conference, ESWC*. 2010, pp. 425–439.

[60] Tomas Lampo et al. “To cache or not to cache: The effects of warming cache in complex sparql queries”. In: *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*. Springer. 2011, pp. 716–733.

[61] Ora Lassila, Ralph R Swick, et al. “Resource description framework (RDF) model and syntax specification”. In: (1998). URL: `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.44.6030`.

[62] Jens Lehmann et al. “DBpedia–a large-scale, multilingual knowledge base extracted from Wikipedia”. In: *Semantic Web* 6.2 (2015), pp. 167–195. DOI: `10.3233/SW-140134`.

[63] Nicolás Lehmann and Jorge Pérez. "Implementing graph query languages over compressed data structures: A progress report". In: *Alberto Mendelzon International Workshop on Foundations of Data Management.* 2015, p. 96. URL: `http://ceur-ws.org/Vol-1378/AMW%5C_2015%5C_paper%5C_19.pdf`.

[64] Maurizio Lenzerini. "Data integration: A theoretical perspective". In: *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems.* ACM. 2002, pp. 233–246.

[65] *Linked Data.* `https://www.w3.org/DesignIssues/LinkedData.html`. Accessed: 2019-12-01.

[66] Matteo Lissandrini, Martin Brugnara, and Yannis Velegrakis. "Beyond macrobenchmarks: microbenchmark-based graph database evaluation". In: *Proceedings of the VLDB Endowment* 12.4 (2018), pp. 390–403. DOI: `10.14778/3297753.3297759`.

[67] Roger MacNicol and Blaine French. "Sybase IQ multiplex-designed for analytics". In: *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30.* VLDB Endowment. 2004, pp. 1227–1230.

[68] Mohamed Nadjib Mami et al. "Towards semantification of big data technology". In: *International Conference on Big Data Analytics and Knowledge Discovery.* Springer. 2016, pp. 376–390.

[69] Norbert Martinez-Bazan, Sergio Gomez-Villamor, and Francesc Escale-Claveras. "DEX: A high-performance graph database management system". In: *2011 IEEE 27th International Conference on Data Engineering Workshops.* IEEE. 2011, pp. 124–127. DOI: `10.1109/ICDEW.2011.5767616`.

[70] Norbert Martínez-Bazan et al. "Dex: high-performance exploration on large graphs for information retrieval". In: *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management.* ACM. 2007, pp. 573–582. DOI: `10.1145/1321440.1321521`.

[71] Michael Meier. "Towards rule-based minimization of RDF graphs under constraints". In: *International Conference on Web Reasoning and Rule Systems.* Springer. 2008, pp. 89–103. DOI: `10.1007/978-3-540-88737-9\_8`.

[72] Thomas Neumann and Gerhard Weikum. "RDF-3X: a RISC-style engine for RDF". In: *Proceedings of the VLDB Endowment* 1.1 (2008), pp. 647–659.

[73] Thomas Neumann and Gerhard 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.

[74] Zhi Nie et al. "Efficient SPARQL query processing in mapreduce through data partitioning and indexing". In: *Asia-Pacific Web Conference.* Springer. 2012, pp. 628–635.

[75] Jeff Z Pan et al. "Graph pattern based RDF data compression". In: *Joint International Semantic Technology Conference.* Springer. 2014, pp. 239–256. DOI: `10.1007/978-3-319-15615-6\_18`.

[76] Nikolaos Papailiou et al. "H 2 RDF+: High-performance distributed joins over large-scale RDF graphs". In: *2013 IEEE International Conference on Big Data.* IEEE. 2013, pp. 255–263.

[77] Harshal Patni, Cory Henson, and Amit Sheth. "Linked sensor data". In: *Collaborative Technologies and Systems (CTS), 2010 International Symposium on.* IEEE. 2010, pp. 362–370.

[78] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. "Semantics and Complexity of SPARQL". In: *International semantic web conference.* Springer. 2006, pp. 30–43.

[79] Matthew Perry, Prateek Jain, and Amit P Sheth. "SPARQL-ST: Extending SPARQL to Support Spatio temporal Queries". In: *Geospatial semantics and the semantic web.* Springer, 2011, pp. 61–86.

[80] Danh Le Phuoc et al. "The Graph of Things: A step towards the Live Knowledge Graph of connected things". In: *J. Web Sem.* 37-38 (2016), pp. 25–35.

[81] Danh Le-Phuoc et al. "A native and adaptive approach for unified processing of linked streams and linked data". In: *International Semantic Web Conference.* Springer. 2011, pp. 370–388.

[82] Danh 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), pp. 25–35.

[83] Reinhard Pichler et al. "Redundancy elimination on RDF graphs in the presence of rules, constraints, and queries". In: *International Conference on Web Reasoning and Rule Systems.* Springer. 2010, pp. 133–148. DOI: `10.1007/978-3-642-15918-3\_11`.

[84] Giuseppe Pirrò. "Explaining and suggesting relatedness in knowledge graphs". In: *ISWC.* Springer. 2015, pp. 622–639.

[85] E Prud'hommeaux and A Seaborne. *SPARQL query language for RDF. W3C Recommendation (January 15, 2008).* 2011. URL: `https://www.w3.org/TR/rdf-sparql-query/`.

[86] Eric Prud'hommeaux. "SPARQL query language for RDF, W3C recommendation". In: *http://www. w3. org/TR/rdf-sparql-query/* (2008).

[87] Roshan Punnoose, Adina Crainiceanu, and David Rapp. "Rya: a scalable RDF triple store for the clouds". In: *Proceedings of the 1st International Workshop on Cloud Intelligence.* ACM. 2012, p. 4.

[88] *RDF 1.1 Concepts and Abstract Syntax.* `https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/`. Accessed: 2019-12-01.

[89] *RDF Schema 1.1 Recommendation.* `https://www.w3.org/TR/2014/REC-rdf-schema-20140225/`. Accessed: 2019-12-01.

[90] Mark A Roth and Scott J Van Horn. "Database compression". In: *ACM Sigmod Record* 22.3 (1993), pp. 31–39. URL: `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.464.643&rank=1`.

[91] David J Russomanno, Cartik Kothari, and Omoju Thomas. "Sensor ontologies: from shallow to deep models". In: *Proceedings of the Thirty-Seventh Southeastern Symposium on System Theory.* 2005.

[92] Alexander Schätzle et al. "Cascading Map-Side Joins over HBase for Scalable Join Processing." In: *SSWS+ HPCSW@ ISWC.* 2012, pp. 59–74.

[93] Alexander Schätzle et al. "PigSPARQL: A SPARQL Query Processing Baseline for Big Data." In: *International Semantic Web Conference (Posters & Demos).* Vol. 1035. 2013, pp. 241–244.

[94] Alexander Schätzle et al. “Sempala: Interactive SPARQL query processing on hadoop”. In: *International Semantic Web Conference*. Springer. 2014, pp. 164–179.

[95] Michael Schmidt, Michael Meier, and Georg Lausen. “Foundations of SPARQL query optimization”. In: *Proceedings of the 13th International Conference on Database Theory*. ACM. 2010, pp. 4–33.

[96] Paul Shannon et al. “Cytoscape: A Software Environment for Integrated Models of Biomolecular Interaction Networks”. In: *Genome Research* 13 (2012), pp. 2498–2504.

[97] Amit Singhal. “Introducing the knowledge graph: things, not strings”. In: *Official google blog* 5 (2012). URL: `https://www.blog.google/products/search/introducing-knowledge-graph-things-not/`.

[98] Mike Stonebraker et al. “C-store: a column-oriented DBMS”. In: *Proceedings of the 31st international conference on Very large data bases*. VLDB Endowment. 2005, pp. 553–564. DOI: `10.1145/3226595.3226638`.

[99] Jeffrey D Ullman. “Information integration using logical views”. In: *International Conference on Database Theory*. Springer. 1997, pp. 19–40.

[100] Jeffrey D Ullman. *Principles of database systems*. Galgotia publications, 1984.

[101] Maria-Esther Vidal et al. “Efficiently Joining Group Patterns in SPARQL Queries”. In: *The Semantic Web: Research and Applications, 7th Extended Semantic Web Conference, ESWC 2010, Heraklion, Crete, Greece, May 30 - June 3, 2010, Proceedings, Part I*. 2010, pp. 228–242.

[102] Maria-Esther Vidal et al. “On the selection of SPARQL endpoints to efficiently execute federated SPARQL queries”. In: *Transactions on Large-Scale Data-and Knowledge-Centered Systems XXV*. Springer, 2016, pp. 109–149.

[103] Maria-Esther Vidal et al. “Transforming Heterogeneous Data into Knowledge for Personalized Treatments A Use Case”. In: *Datenbank-Spektrum* (), pp. 1–12. URL: `https://doi.org/10.1007/s13222-019-00312-z`.

[104] Till Westmann et al. “The implementation and performance of compressed databases”. In: *ACM Sigmod Record* 29.3 (2000), pp. 55–67. DOI: `10.1145/362084.362137`.

[105] Xifeng Yan and Jiawei Han. “gspan: Graph-based substructure pattern mining”. In: *2002 IEEE International Conference on Data Mining, 2002. Proceedings.* IEEE. 2002, pp. 721–724.

[106] Matei Zaharia et al. “Apache Spark: a unified engine for big data processing”. In: *Commun. ACM* 59.11 (2016), pp. 56–65. DOI: `10.1145/2934664`. URL: `http://doi.acm.org/10.1145/2934664`.

[107] Ying Zhang et al. “SRBench: A Streaming RDF/SPARQL Benchmark”. In: *The Semantic Web - ISWC 2012 - 11th International Semantic Web Conference, Boston, MA, USA, November 11-15, 2012, Proceedings, Part I*. 2012, pp. 641–657.

[108] Marcin Zukowski et al. “Super-Scalar RAM-CPU Cache Compression.” In: *Icde*. Vol. 6. 2006, p. 59. DOI: `10.1109/ICDE.2006.150`.