

On Supporting Interoperability between RDF and Property Graph Databases

Dissertation
zur
Erlangung des Doktorgrades (Dr. rer. nat.)
der
Mathematisch-Naturwissenschaftlichen Fakultät
der
Rheinischen Friedrich-Wilhelms-Universität Bonn

von
Harsh Vrajeshkumar Thakker
aus
Jaora, Ratlam, Madhya Pradesh, Indien

Bonn, 09.08.2020

Angefertigt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät der
Rheinischen Friedrich-Wilhelms-Universität Bonn

1. Gutachter: Prof. Dr. Sören Auer
2. Gutachter: Prof. Dr. Jens Lehmann
Tag der Promotion: 30.04.2021
Erscheinungsjahr: 2021

या देवी सर्वभूतेषु विद्या रूपेण संस्थिता
नमस्तस्यै-नमस्तस्यै, नमस्तस्यै नमो नमः।

- Translation -

I bow down to you, O Mother Divine, who are the very manifestation of
knowledge in all living beings.

कर्मण्येवाधिकारस्ते मा फलेषु कदाचन।
मा कर्मफलहेतुर्भूर्मा ते सङ्गोऽस्त्वकर्मणि।

- Translation -

You have a right to perform your prescribed duty, but you are not entitled to the
fruits of actions

Never consider yourself the cause of the results of your activities, and never be
attached to not doing your duty.

Abstract

Over the last few years, the amount and availability of machine-readable Open, Linked, and Big data on the web has increased. Simultaneously, several data management systems have emerged to deal with the increased amounts of this structured data. RDF and Graph databases are two popular approaches for data management based on modeling, storing, and querying graph-like data. RDF database systems are based on the W3C standard RDF data model and use the W3C standard SPARQL as their defacto query language. Most graph database systems are based on the Property Graph (PG) data model and use the Gremlin language as their query language due to its popularity amongst vendors. Given that both of these approaches have distinct and complementary characteristics – RDF is suited for distributed data integration with built-in world-wide unique identifiers and vocabularies; PGs, on the other hand, support horizontally scalable storage and querying, and are widely used for modern data analytics applications, – it becomes necessary to support interoperability amongst them. The main objective of this dissertation is to study and address this interoperability issue. We identified three research challenges that are concerned with the *data interoperability*, *query interoperability*, and *benchmarking* of these databases. First, we tackle the *data interoperability* problem. We propose three direct mappings (schema-dependent and schema-independent) for transforming an RDF database into a property graph database. We show that the proposed mappings satisfy the desired properties of semantics preservation and information preservation. Based on our analysis (both formal and empirical), we argue that any RDF database can be transformed into a PG database using our approach. Second, we propose a novel approach for querying PG databases using SPARQL using Gremlin traversals – GREMLINATOR to tackle the *query interoperability* problem. In doing so, we first formalize the declarative constructs of Gremlin language using a consolidated graph relational algebra and define mappings to translate SPARQL queries into Gremlin traversals. GREMLINATOR has been officially integrated as a plugin for the Apache TinkerPop graph computing framework (as `sparql-gremlin`), which enables users to execute SPARQL queries over a wide variety of OLTP graph databases and OLAP graph processing frameworks. Finally, we tackle the third, *benchmarking* (performance evaluation), problem. We propose a novel framework – LITMUS BENCHMARK SUITE that allows a choke-point driven performance comparison and analysis of various databases (PG and RDF-based) using various third-party real and synthetic datasets and queries. We also studied a variety of intrinsic and extrinsic factors – data and system-specific metrics and Key Performance Indicators (KPIs) that influence a given system’s performance. LITMUS incorporates various memory, processor, data quality, indexing, query typology, and data-based metrics for providing a fine-grained evaluation of the benchmark. In conclusion, by filling the research gaps, addressed by this dissertation, we have laid a solid formal and practical foundation for supporting interoperability between the RDF and Property graph database technology stacks. The artifacts produced during the term of this dissertation have been integrated into various academic and industrial projects.

Acknowledgements

This dissertation results from almost five years of hard work, patience, failures, perseverance, success, and believing in oneself. That being said, there are many people without whose *behind-the-curtain* efforts and constant support; this day would not have come. I take this opportunity to thank as many I can. I fondly remember the famous Indian mystic poet and saint Kabir Das couplet “*Guru Gobind dono khade, Kako lagoon paay? Balihari Guru aapne Gobind diyo batay*”. Here Kabir says Teacher is above even God, because it is only because of teacher’s illumination (teaching) that I can experience God. Similarly, I would like to thank my mentors Prof. Dr. Sören Auer and Prof. Dr. Jens Lehmann. I started with Sören in late 2015, this incredible journey of my Ph.D. and then continued with Jens since 2017. I have learned the art of research, time management, networking, and above all, how to be a visionary and pursue my dreams. They have been patient, kind, and extremely accommodating throughout my journey and have always encouraged me to take on the uphill battles and taught me how to win.

I would also like to express my gratitude to Ass. Prof. Dr. Renzo Angles of University of Talca, Chile; and Dr. Marko Rodriguez (inventor & founder of the Gremlin language and Apache TinkerPop project), for being huge contributors to my technical understanding of the formal concepts and the fantastic collaboration opportunities that have led to such an impacting thesis. I want to thank Dr. Christoph Lange and all the staff members of the Enterprise Information Systems (EIS) and Smart Data Analytics (SDA) for the great time I had.

My heartfelt thanks to the fantastic women – Asst. Prof. Dr. Maria Maleshkova, Dr. Petra Selmer (Software Engineer & Researcher, Neo4j), and Frau Martina Doelp (our department secretary at the university). Thank you for being so kind and helping me out with your guidance through my Ph.D., without your feedback and encouragement, this would not have been so cherishable.

Special shout-out to Dharmen Punjani, Sashwat Saincher, Alexandra Scholz, Anjani Dhrangdhariya, Vyoma Brahmhatt, Shraddha Patel, Mohnish & Alpana Dubey, Gezim Sejdiu, Denis Lukovnikov, Dr. Kuldeep & Mrs. Ankita Singh, Dr. Diego Esteves, Mehdi Ali, Yash Patel, Sussane Uibel, Deshna Jain, Giulia Reinbothe, Stephen Mallette, Ganesh Iyer, Tanvi Goswami, Nandita Khetan, Claudia Nuñez, Karthik Pillai, Vishwani Gupta, and all my other friends at the dormitory for the wonderful times.

This Ph.D. thesis is dedicated to my father, Vrajesh Thakkar, and my mother, Hina Thakkar, who are the pillars of my success. They are to my life what the Sun is to the solar system.

Contents

1	Introduction	1
1.1	Motivation, Problem Statement and Challenges	2
1.1.1	Challenge 1: Data Interoperability	4
1.1.2	Challenge 2: Query Interoperability	6
1.1.3	Challenge 3: RDF vs Property graph Database Benchmarking	7
1.1.4	Methodology and Approach	7
1.2	Research Questions	9
1.3	Thesis Overview	11
1.3.1	Contributions	11
1.3.2	Publications	14
1.4	Thesis Outline	18
2	Background and Preliminaries	21
2.1	Semantic Web & Linked Data	21
2.1.1	RDF Data Model and OWL	23
2.1.2	SPARQL Query Language	25
2.1.3	RDF Databases	28
2.2	Graph Databases	29
2.2.1	Property Graph Data Model	31
2.2.2	Gremlin Traversal Language and Machine	33
2.2.3	Apache TinkerPop Graph Computing Framework	36
2.3	Knowledge Graphs	39
2.4	Summary	42
3	Related Work	43
3.1	Data Interoperability between Databases	44
3.1.1	Syntactic Interoperability	44
3.1.2	Semantic Interoperability	46
3.1.3	Other Approaches for Data Interoperability	49
3.2	Query Interoperability between Databases	50
3.2.1	Graph-based \leftrightarrow Relational	50
3.2.2	Graph-based \leftrightarrow Document-based	53
3.2.3	Graph-based \leftrightarrow Hierarchical	53
3.2.4	Other Intermediate Graph-based Approaches	54
3.2.5	Commercial Database Approaches	55
3.3	Benchmarking Frameworks for RDF and Property graph Databases	57
3.4	Summary	60

4	Directly Mapping RDF Databases to Property graph Databases	61
4.1	RDF Database (as an edge-labeled graph)	62
4.1.1	RDF Graph	63
4.1.2	RDF Graph Schema	67
4.1.3	Valid RDF Graph	69
4.2	Property Graph Database	71
4.2.1	Property Graph	71
4.2.2	Property Graph Schema	73
4.2.3	Valid Property Graph	75
4.3	Direct Mappings for Data Transformation	75
4.3.1	Simple Database Mapping (SDM)	76
4.3.2	Generic Database Mapping (GDM)	78
4.3.3	Complete Database Mapping (CDM)	84
4.4	Experimental Evaluation	89
4.4.1	Implementation	89
4.4.2	Methodology and Experimental Setup	90
4.4.3	Experimental Results	91
4.4.4	Interoperability in Practice	94
4.4.5	Limitations	96
4.5	Summary	97
5	GREMLINATOR: Querying Property graph Databases using SPARQL	101
5.1	Graph Relational Operators of Gremlin Traversal Language	103
5.1.1	Defining Gremlin Operators	103
5.1.2	Graph Pattern Matching via Traversing	105
5.1.3	Mapping Gremlin traversals to Graph Algebra	110
5.2	GREMLINATOR Approach	112
5.2.1	Mapping SPARQL BGPs to Gremlin SSTs	112
5.2.2	Mapping SPARQL Queries to Gremlin Traversals	113
5.2.3	Explanation of the Transformation	115
5.3	Implementation	117
5.3.1	Encoding Prefixes	117
5.3.2	GREMLINATOR (sparql-gremlin) Architecture	117
5.3.3	SPARQL Coverage and Limitations	119
5.4	Experimental Evaluation	119
5.4.1	Evaluation Methodology	119
5.4.2	Evaluation Metrics	120
5.4.3	Datasets	121
5.4.4	Queries	122
5.4.5	System Setup	123
5.5	Results and Discussion	124
5.5.1	Q1 - Query Preservation	124
5.5.2	Q2 - Translation Validity	124
5.5.3	Q3 - Performance Analysis	124
5.5.4	Discussion	129
5.6	Gremlinator as a Reusable Resource	130
5.6.1	Technical Quality	130

5.6.2	Availability	131
5.6.3	Reusability and Maintenance	131
5.7	Community Adoption (Use Cases)	131
5.7.1	IBM Research AI use case	131
5.7.2	SANSA Stack use case	132
5.7.3	Contextualised Knowledge Graph use case	132
5.7.4	Open Research Knowledge Graph use case	132
5.8	Summary	132
6	Automatic Benchmarking of RDF and Graph Databases	135
6.1	LITMUS Framework Architecture	136
6.1.1	Data Facet	137
6.1.2	Query Facet	137
6.1.3	System Facet (DMS Facet)	138
6.1.4	Benchmarking Core	138
6.2	The LITMUS Environment	138
6.2.1	Integrated Datasets	140
6.2.2	Integrated DMSs	141
6.2.3	Supported Queries	142
6.2.4	Execution Environment	143
6.3	Performance Evaluation	144
6.3.1	Selected Parameters	144
6.3.2	Selected Metrics and KPIs	145
6.3.3	Data Visualisation	147
6.4	Experimental Evaluation	147
6.4.1	System Setup	147
6.4.2	Results and Discussion	148
6.4.3	Limitations	150
6.5	Summary	153
7	Conclusion and Future Directions	155
7.1	The Interoperability Story in a Nutshell	155
7.2	Limitations and Future Work	160
7.3	Outlook and Closing Remarks	161
	Bibliography	163
	A Full Results of the SPARQL - Gremlin Performance Comparison	181
	B Complete List of Publications	185
	C Best Paper Awards	189
	List of Figures	193
	List of Tables	195
	Listings	199

Introduction

What started as an experimental infrastructure to interconnect, share and retrieve scientific documents, between scientists working at various universities and organizations by Sir Tim Berners-Lee at CERN in 1989¹, led to revolutionizing the ways of information sharing as we know it today. These winds of revolution began on April 30, 1993, when this infrastructure, today is known as the World Wide Web (WWW)[1] (or simply the Web), was made public in order to facilitate its dissemination. This resulted in about 500 more servers using the free source code, which accounted for the Web to around 1% of the global internet traffic by late 1993. These numbers increased exponentially, leading to 10,000 servers by the end of 1994 and hundreds of millions, even billions more, as of writing this thesis.

The Web's main advantage was that it linked all the disconnected, heterogeneous, and inconsistent data from various distributed sources under one umbrella of a global interconnected, uniquely identifiable distributed open information network. This capability of ubiquitous publication and availability of information quick-started the rise of collaborative innovation and creativity throughout the globe. As more and more organizations, stakeholders and individuals were exposed to this information technology, the nature, amount, and rate of the information being created and published quickly exploded, rendering the classical methods of data storage and information retrieval in search of novel solutions. This resulted in a collaborative effort in data management research, leading to the development of database management systems [2].

However, the Web's published information was highly interconnected in nature, comprising complex relationships between different individuals, organizations, concepts, and entities from diverse domains resulting in a graph-like structure. The traditional relational database management systems proved inefficient in storage and querying the graph-structured complex cross-domain data with a loose requirement for data schema. Thus, a typical data management problem becomes a graph data management problem when it is concerned with not only the analysis of the values but also the discovery of the connections between them. Graphs are distinctly valued when it comes to choosing formalisms for modeling real-world scenarios such as biological, transport, communication, and social networks due to their intuitive data model.

This resulted in another fork of the mainstream research of information technology and data

¹ <https://home.cern/science/computing/birth-web>

management, giving rise to graph data management, which focuses on development and exploration of methods for efficient data representation (and modeling), storage, and querying (and traversing) of data in graph structures. Graph data management has revealed beneficial characteristics in terms of flexibility and scalability by differently balancing between query expressivity and schema flexibility. Graph analysis tools have turned out to be pioneering applications in understanding these natural and human-made networks [3]. This peculiar advantage has resulted in an unforeseen race of developing new task-specific graph systems, query languages, and data models. These include property graphs, key-value, wide column, resource description framework, etc. The two most popular data models for graph data management are the Property Graph (PG) and the W3C Resource Description Framework (RDF) [4, 5].

Semantic Web, an extension of the WWW, enables intelligent access to the Web’s data by exposing the data in a machine-readable format. The Semantic Web consists of a set of standards set by the World Wide Web Consortium (W3C) footnote<https://www.w3.org/>. In the Semantic Web, RDF is the standard data model [6], the standard query language is SPARQL [7], and there are languages to describe structure, restrictions and semantics on RDF data (e.g. RDF Schema [8], OWL [9], SHACL [10], and ShEx [11]). On the other hand, most Graph database systems are based on the Property graph data model [12]. Unlike the Semantic Web, there is no standard for data, query language (although there is a joint proposal GQL [13]), and the notions of graph schema and integrity constraints are limited [14]. Some of the popular graph query languages are Cypher [15] for Neo4j, Gremlin [16, 17] for the Apache TinkerPop [18]-enabled systems (these cover a wide variety of commercial graph databases systems, cf. Section 2.2), PGQL [19] of Oracle.

Both approaches have distinct and complementary characteristics – RDF is suited for distributed data integration with built-in world-wide unique identifiers and vocabularies; Property graphs on the other hand support horizontally scalable storage and querying, and are widely used for modern data analytics applications (including OLTP). Present-day graph query languages focus on flexible graph pattern matching (aka sub-graph matching), whereas graph computing frameworks aim to provide fast parallel (distributed) execution of instructions. The consequence of this rapid growth in the variety of graph-based data management systems has resulted in a lack of standardization, which is the cradle of the severe lack of interoperability between these systems (in particular the latter) [13].

1.1 Motivation, Problem Statement and Challenges

Given the intrinsic connection between RDF triple stores and Property graph databases, and their popularity for representing open knowledge, it becomes necessary to develop methods to allow interoperability among these systems.

This dissertation combines the data and semantics with the current technologies: Semantic Web and Property graphs to leverage the best of both worlds. The overarching research problem this dissertation investigates is:

Overarching Research Problem: How can we support interoperability between the Semantic Web and Property graph Databases?

The term “Interoperability” was introduced in the area of information systems. It could be defined as the ability of two or more systems or components to exchange information and to use the information that has been exchanged [20]. In the context of data management, interoperability is concerned with the support of applications that exchange and share information across the boundaries of existing databases [21].

Providing interoperability between database models, systems, and applications is a very concrete and pragmatic problem, which stems from the need for reusing existing systems and programs for building new applications [21]. Data and information interoperability is relevant for several reasons, including:

- Promotes data exchange and data integration [22];
- Allows us to have a common understanding of the meanings of the data [23];
- Allows the creation of information and knowledge, and their subsequent reuse and sharing [24];
- Facilitates access to a large number of independently created and managed information sources of broad variety [24];
- Facilitates the reuse of available systems and tools [21];
- Allows to explore the best features of different approaches and systems [25];
- Enables a fair comparison of database systems by using benchmarks [26];
- Supports the success of emergent systems and technologies [21];
- It is a crucial factor for the development of new information systems [27].

One can define several forms of interoperability in information systems [28]. For instance, focusing on the dimension of heterogeneity, Sheth [24] defined four levels of interoperability: system, syntax, structure, and semantic. The system-level interoperability concerns the heterogeneity of computer systems and communications. The syntax level considers machine-readable aspects of data representation (i.e., data formats and serializations). The structure level involves data modeling constructs and schematic heterogeneity. The semantic level requires that the information system understand the semantics of the users’ information request and those of information sources.

In the context of Web Languages and Ontologies, syntactic interoperability means that the applications can take advantage of parsers and APIs, providing syntactical manipulation facilities. Additionally, semantic interoperability implies that applications can understand the meaning of representations and set up automatically mappings between different representations by content analysis [29].

In the context of databases, interoperability can be divided into data (syntactic and semantic) and query interoperability.

Syntactic interoperability refers to the ability of a database system to use data from other database systems [30]. It could mean that both database systems can exchange information, although they may not be aware of the meaning of such information. Turtle, TriG, RDF/XML, RDF/JSON, and JSON-LD are data formats for encoding RDF data. In contrast, there is no data format to encode property graphs. Given this restriction, some systems use graph data formats (GraphML, DotML, GEXF, GraphSON). We discuss more about the several data serialisation formats for RDF and Property graphs in Chapter 3 Section 3.1.1.

Semantic interoperability can be defined as the ability of database systems to exchange data in a meaningful way. It implies that the systems have a common understanding of the meanings of the data [23]. Here, the transformation methods have to be information and semantics, preserving ensuring that there is no loss of data and meaning during the transformation process. In such methods both, data and schema must participate in the transformation. Ref to the chapter related work pointing to the existing works in this area. We discuss more about the semantic interoperability between RDF and Property graphs in Chapter 3 Section 3.1.1.

Query interoperability implies the existence of methods to transform different query languages or data accessing methods between two systems. It means that a query in the source database system can be translated into one that can be directly executed on the target system [31]. Here, the translation methods have to be query preserving, ensuring that the meaning of the original is not altered in the transformation process. We discuss more about the *query interoperability* topic in Chapters 3 and 5.

Thus, in general terms, syntactic interoperability between RDF and Property graph databases means data exchange at the level of serialization formats. Semantic interoperability implies the definition of data and schema mappings, and query interoperability implies query translations among SPARQL and property graph query languages.

Based on a comprehensive literature review (discussed in detail in Chapter 3) about RDF and Property graph interoperability, we identified several three main challenges, each comprising of specific sub-challenges or issues that need to be understood and addressed. Each challenge corresponds to a sub-research question (discussed in Section 1.2).

1.1.1 Challenge 1: Data Interoperability

The *data interoperability* challenge consists of two sub-challenges – syntactic and semantic interoperability as discussed earlier. We discuss each of these next.

Syntactic Interoperability:

- There is no standard data format for encoding property graphs. This is a crucial issue to support syntactic interoperability.

- The most RDF serializations are triple-centric, while the most Property graph serializations represent graph as lists of nodes and edges.
- Despite the serializations based on JSON or XML in both models, the syntaxes used are difficult to map.
- The support for multi-values is different in the models. A property graph support arrays, while RDF provides different types of lists.
- The RDF data model allows metadata about properties, i.e., edges between edges are allowed. Although this feature is not common in real data, data mapping should be able to manage it. Note that a property graph does not support multi-level metadata.
- RDF reification leads to an explosion in the size of the resulting graph. This can be avoided by implementing a “smart” transformation that can recognize a set of triples describing a reification, and map them to a single node in the property graph.

Semantic Interoperability:

- The RDF model presents features with special meaning (or semantics) that cannot be modeled by the property graph data model (at least not in a trivial manner). Blank nodes, reification, and entailment are some of these features. Similarly, it is not possible to model more than one edge with the same label (two same edges) between two nodes in the RDF data model, which can be represented in the PG data model.
- Usually, an RDF database contains a mix of data and schema. In such a case, it is necessary to decide whether to extract the schema (and transform it independently) or process the schema as part of the data.
- Another intrinsic feature of an RDF database is the occurrence of a partial schema. In such a case, we must define whether the schema will be used or not. In the first case, it could be necessary to “discover” the schema and then transform the data. Hence, such an approach could imply the use of a transformation method that is schema independent, or a combined method that supports data with or without schema.
- A semantic issue is the right and complete interpretation of a reified triple, and its representation in a property graph.
- RDF Schema supports the definition of subclass and subproperty. Current property graph database systems do not support these features.
- OWL, which is intended to be a layer above RDF Schema, supports more complex constraints for classes (e.g., intersection) and properties (e.g., transitivity). The property graph model does not support these features.
- An RDF database could contain semantic information that allows data inference (i.e., to infer new triples based on the existing triples). Current graph database systems have not

been designed to support inference.

- Discovering semantic information and resolving mismatches requires the application of human intelligence and judgment. Hence, the semantic interoperability is determined by the power of the translation methods to support data and semantics interpretation.

1.1.2 Challenge 2: Query Interoperability

- Unlike the standardization (via the W3C standards and ISO committees) of query languages for the relational databases (SQL) and the RDF databases (SPARQL), property graph databases do not have a standard query language². This has led to the development of a wide range of vendor-specific graph query languages (e.g., Cypher for Neo4j and Gremlin for Apache TinkerPop).
- Most of the current property graph query languages do not have a solid formal foundation (semantics, complexity, and expressiveness). This raises a critical challenge for supporting query interoperability since a formal mapping between SPARQL and a property graph language cannot be defined.
- The notion of schema in the context of property graph query languages is not strictly defined, or even absent in some cases due to their NoSQL oriented nature. This creates another challenge when aiming to transform RDF data (which consists of schema information) to property graph data.
- Property graph query languages address two different paradigms: *declarative* and *imperative*. For instance, Cypher is a declarative query language whereas Gremlin is an imperative graph traversal language which also offers a declarative construct. This adds an extra challenge since these two different paradigms operate on disparate sets of semantics (i.e., set vs. bag semantics) while aiming to support query interoperability.
- There are some on-going efforts, such as [33, 34], that advocate consolidating the relational and graph algebras to lay a foundation for proving the equivalences between the different transformations and mappings to support *query interoperability* between RDF and Property graphs. Nonetheless, there is still scope for improvement.

Therefore, there is a need to propose either a – (i) standardized query language for property graph databases (in progress), which will facilitate the formal definition and study of query transformation methods, and/or (ii) a common bridge that supports translation of SPARQL query language into several Property graph query languages or one widely popular language such as Gremlin.

² However, there is an early draft or a manifesto in progress at the time of writing this thesis – GQL [32].

1.1.3 Challenge 3: RDF vs Property graph Database Benchmarking

Benchmarking is widely used for evaluating databases. Benchmarks exist for a variety of abstractions, from simple data models to graphs and triple stores, to entire enterprise information systems. The process of benchmarking databases is exceptionally tedious. Thus it demands a high level of automation without compromising the quality of evaluation. Furthermore, benchmarking software is required to be fair (discard any system or data-specific bias) and modular architecture to promote reuse and reproducibility. While there exist several standalone benchmarking tools and complete benchmarking software for both RDF and Property graph databases, there is no single benchmark that caters all the above mention features and evaluates both RDF and Property graph databases within the same environment.

The successful development of such a framework will have to address the following research and implementation driven challenges [35], such as: (i) openness/fairness (provide support for transparency and control of fine coarse configuration settings of both RDF and Property graph databases within the framework), (ii) modularity (provide support for integration of existing third party data and systems for benchmarking), (iii) extensibility (provide easy support for extending existing and adding new features) (iv) abstraction (i.e. provide support of a Graphical User Interface (GUI) for the end user), (v) reproducibility (provide support for exporting data, queries, configurations and results in standard serialization formats); (vi) full automation (provide support for end to end automation of the complete benchmarking process, i.e. data transformation and loading, system configuration and setup, query loading and translation, benchmark execution, result and report generation), and lastly (vii) identifying suitable KPIs (define and identify relevant Key Performance Indicators (KPIs) in order to pinpoint the effect of various factors such as data quality, RAM, CPUs, Indexing, Query typology, etc., on the performance of the participating RDF and Property graph databases.

Therefore, the end goal is to develop a fully automated, open, extensible, and reusable framework that enables orchestrating end-to-end benchmarking for both RDF and Property graph databases using a variety of real and synthetic datasets.

1.1.4 Methodology and Approach

In this section, we layout the methodology and approach used to identify, understand and address the significant challenges (*Challenges 1, 2, and 3*) to be discussed in this dissertation by bringing them all under a single umbrella framework. Understanding the relationship between the Semantic Web and Property graph databases, we adopt the methodology used by Sequeda [36], and decompose each technology into corresponding layers. This allows us to recognize the similarities between these two technology stacks, thereby enabling us to build a well-defined layer by layer mapping approach. Figure 1.1 shows the relationship between the Semantic Web and Property graph databases.

We can make the following conclusions upon investigation:

- The first layer of both Semantic Web and Property graph databases are the graph data models – RDF and Property graph.

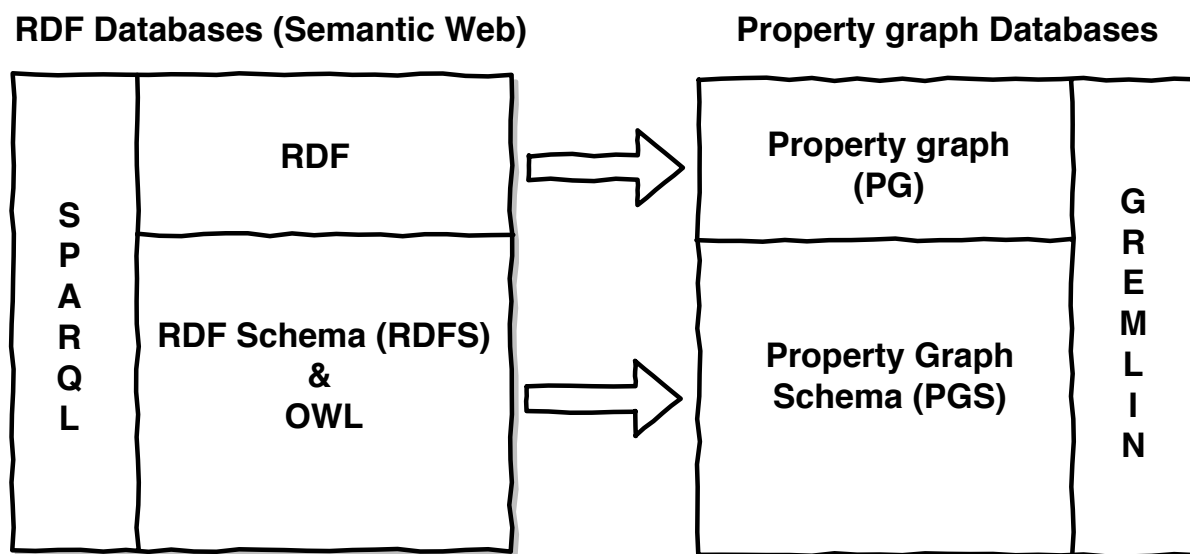


Figure 1.1: The Layered Cake mapping between the Semantic Web (RDF databases) and Property graph databases.

- The next layer caters support for describing the schema using the RDFS, OWL in Semantic Web, and a Property graph Schema.
- Both the technologies have a graph query language – SPARQL and Gremlin³
- Semantic Web technology has a standard for the data model and query language, whereas the Property graph database domain is much open in terms of choice due to the lack of a well-defined standard.

This direct correspondence between the two technology stacks shown in Figure 1.1 advocate that it is possible to develop a framework setting that enables means to transform (virtualize) the RDF databases as Property graph databases. Figure 1.2 presents the general framework for supporting the interoperability between RDF and Property graph database using automatic virtualization. This broadly identifies two challenges. First, one requires transforming (via direct mapping) the Semantic Web data and schema, which is the RDF data model and OWL into the corresponding Property graph data model and Property graph Schema (this is covered by **Challenge 1 - Data Interoperability**). Second, it requires transforming (via direct mapping) the SPARQL queries into corresponding Gremlin traversals (this is covered by **Challenge 2 - Query Interoperability**). The third challenge is concerned with the integration and evaluation of the proposed mappings (the results of addressing Challenge 1 and 2) in a unified, automatic, open and extensible environment, which requires the development of a benchmarking framework (this is covered by **Challenge 3 - Database Benchmarking**).

Thus, addressing all these three challenges enables us to answer the broad research problem of this dissertation – "How can we support interoperability between the Semantic Web and Property graph Databases?". In the next section we discuss the challenge-specific research

³ Our preferred choice of the Gremlin graph traversal language is explained in Chapter 2 Section 2.2.2.

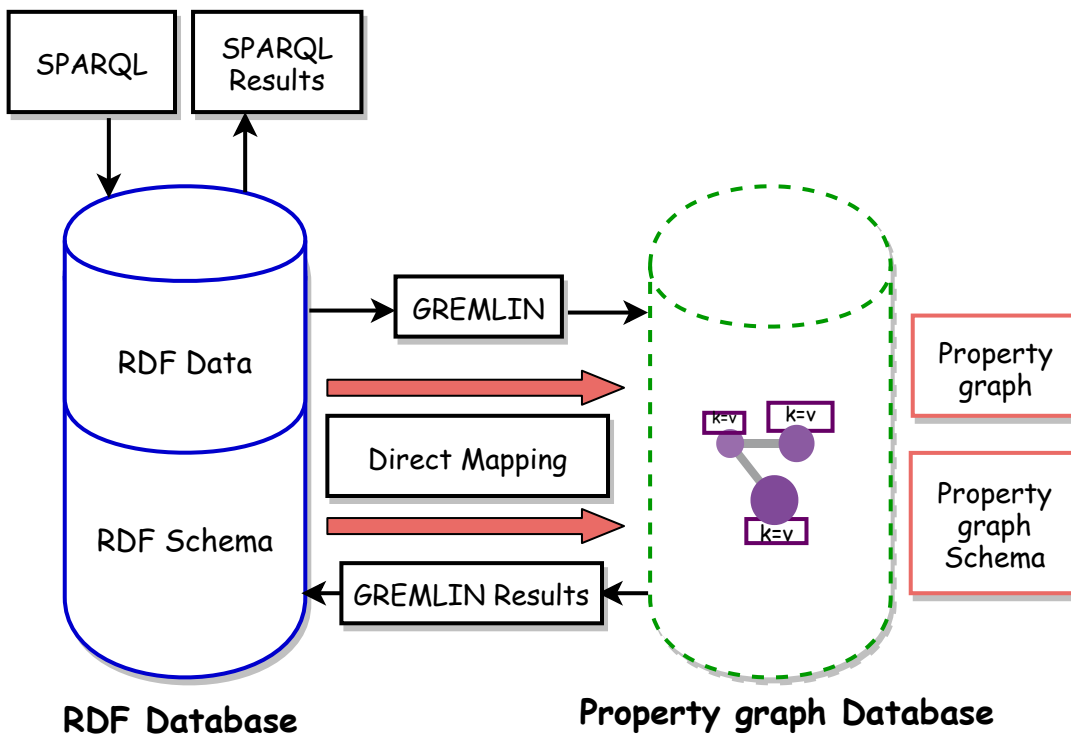


Figure 1.2: General Framework for supporting interoperability between the RDF and Property graph databases.

questions addressed by this thesis dissertation.

1.2 Research Questions

After identifying the main three challenges, we define the following three sub-research questions, one for addressing each challenge. As mentioned earlier, collectively, these three sub-research questions must be answered to tackle the overarching research problem of this dissertation, as illustrated in Figure 1.3.

Next, we explain each of these three sub research questions.

RQ1: Data Interoperability – How can we directly map RDF Databases to Property Graph Databases in an information preserving manner?

This research question's main objective is to study the data interoperability between the RDF and Property graph databases. Recall that this constitutes both syntactic and semantic interoperability, as described in Section 1.1.1 of the current chapter. In order to answer this question, we study existing works and define the notion of RDF, RDF Schema, Property graph, and Property graph Schema in the context of edge-labeled multigraphs. We further analyze the formal semantics and desirable properties that need to be captured in defining direct mappings.

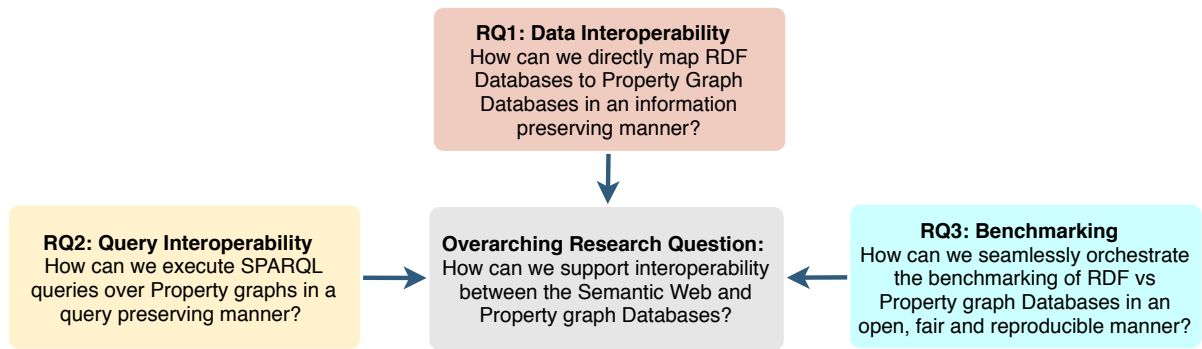


Figure 1.3: The three sub research questions contribute to the overall research objective of this dissertation.

In particular, we identify and study three desirable properties of the direct database mappings: computability, semantics preservation, and information preservation. We further delve into the RDF Blank nodes and reification and possible means to support these special features within the direct database mappings. Based on such analysis, we argue whether or not any RDF database can be transformed into a Property graph database.

RQ2: Query Interoperability – How can we execute SPARQL queries over Property graphs in a query preserving manner?

RDF and Property graph databases are two approaches for data management that are based on modeling, storing, and querying graph-like data. While SPARQL is a W3C standard for querying RDF databases, Gremlin is widely popular amongst the graph database community covering both OLTP and OLAP systems, and due to the broad adoption by most commercial Property graph databases. In this research question, we study the existing works on the SPARQL query language with respect to its syntax, expressivity, and semantics. We also study the foundations of the Gremlin traversal language and the Apache TinkerPop graph computing framework that is used to execute these traversals. In doing so, we quickly realize the lack of a formal foundation for capturing the semantics of Gremlin operators. We need to formally define and consolidate the missing query fragment of the Gremlin traversal language using existing work on graph relational algebra. We need to investigate further means to study the correspondence of Gremlin constructs with the SPARQL query language operators. In doing so, we undertake a short study about the interoperability between these two languages and review the current solutions to the problem, identify their features, and discuss the inherent issues. We also formally study the desirable properties required by the direct query mapping – query preservation.

RQ3: Benchmarking – How can we seamlessly orchestrate the benchmarking of RDF vs Property graph Databases in an open, extensible, fair and reproducible manner?

Over the last few years, the amount and availability of Open, Linked, and Big data on the web has increased. Simultaneously, there has been an emergence of several task-specific graph data management to deal with the increased amounts of structured data. The RDF and Property graph databases are the two most developed systems for graph data management. Apart from

the format of the dataset that they consume, there are several other differences in how they build indexes and execute queries. The main objective of this research question is to objectively evaluate both the RDF and Property graph database systems for a given set of specific scenarios, benchmarks involving particular query loads over characteristic datasets have been made openly available by the graph community. In doing so, we review the existing benchmarking framework and their suitability with respect to this dissertation’s research objective. However, none of the existing tools allow the users to benchmark both of the above-mentioned databases in an open, extensible, and transparent manner that also supports evaluating the underlying data and query transformation approaches. We also need to investigate various KPIs that could help understand the influence of various internal and external factors on the performance of the participating database systems. Thus, we argue that this necessitates the study and development of a benchmarking framework that is sufficiently versatile to orchestrate such benchmarks.

1.3 Thesis Overview

In order to present a high-level yet comprehensive overview of the achieved results, this section consolidates the main contributions of the thesis. It provides references to scientific articles addressing each research question and its corresponding contributions published throughout the whole term of this dissertation.

1.3.1 Contributions

Contributions for RQ1: RDF2PG – Direct mappings for transforming RDF data into Property graphs.

In order to answer the first research question (**RQ1**), concerned with supporting data interoperability between RDF and Property graph databases, we define three database mappings [37, 38]: a simple mapping which allows transforming an RDF graph into a PG without considering schema restrictions (in both sides); a generic mapping which allows transforming an RDF graph (and an RDF schema) into a PG that follows the restrictions defined by a generic PG schema; a complete mapping which allows transforming a complete RDF database into a complete PG database (i.e., schema and instance). We also study the desirable properties of the above database mappings: computability, semantics preservation, and information preservation [39]. Based on such analysis, we formally prove that the proposed mappings are, in fact, satisfying these three properties and argue that it is indeed possible to transform any RDF database into a PG database. In terms of data modeling, we can conclude that the PG data model subsumes the RDF data model’s information capacity. We also implemented [37] the proposed three direct mappings in a system – **RDF2PG**, which is openly available, demonstrating that our mappings work. Figure 1.4 depicts a typical conceptual architecture of the **RDF2PG** data mapper approach. After performing exhaustive experiments using a variety of openly available real and synthetic datasets, we present empirical evidence advocating the applicability and validity of our mappings [37].

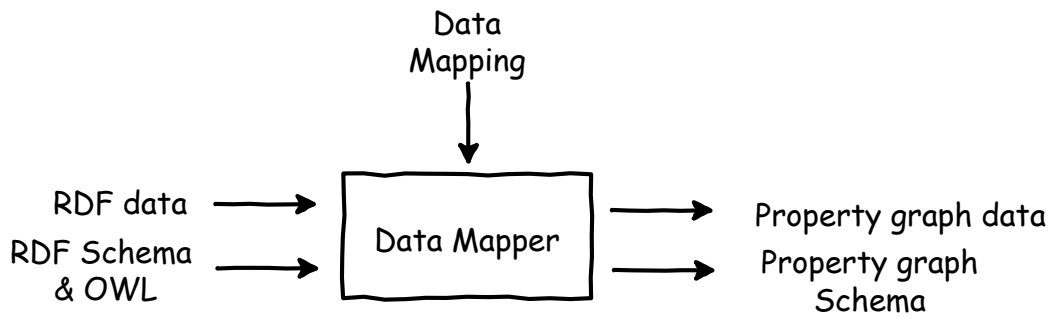


Figure 1.4: The conceptual architecture of the RDF2PG data mapping approach.

Contributions for RQ2: Gremlinator – A direct mapping approach for translating SPARQL queries into declarative Gremlin traversals.

In order to answer the second research question (**RQ2**), concerned with supporting query interoperability between RDF and Property graph databases, we:

- Studied the formal foundation of the Gremlin traversal language and identified the missing formal query operator fragments;
- Consolidated the existing material on graph relational algebra, thereby formally defining the missing operators in the Gremlin traversal language in a unified graph algebra [34, 40];
- Formally defined a direct mapping approach *Gremlinator* [41–43] (as shown in Figure 1.5) for translating the SPARQL queries to corresponding Gremlin traversals thereby incrementally covering the features of the SPARQL 1.0 and 1.1 query fragments.
- Developed an open implementation of our approach *Gremlinator*, which has been successfully integrated within the popular Apache TinkerPop graph computing framework as a plugin [43, 44].
- Performed an exhaustive empirical analysis of **Gremlinator** on a variety of openly available datasets to illustrate the applicability and validity of our proposed mappings [41]. Furthermore, **Gremlinator** has also been adopted by key players (such as IBM Research, National Library of Medicine, etc.) in the academia and industry in several use cases (Knowledge Graphs, Life Sciences, Pharmaceuticals, etc.) advocating the usefulness of our contribution [43].

Our work on **Gremlinator** was awarded as the Best Resource Paper⁴ at the 14th IEEE International Conference on Semantic Computing (ICSC 2020) organised in San Diego, California, USA from February 2 to 5, 2020.

⁴ <http://harshthakkar.in/wp-content/uploads/2020/04/icsc2020bestpaper.jpg>

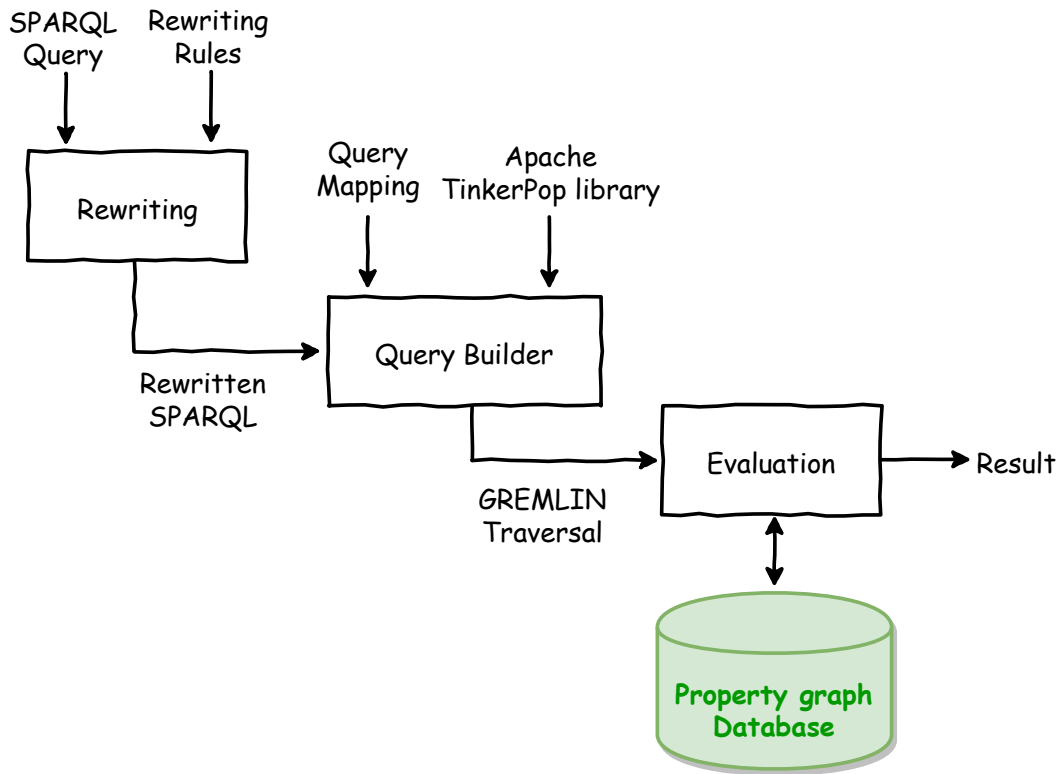


Figure 1.5: The conceptual architecture of the Gremlinator query mapping approach.

Contributions for RQ3: LITMUS Benchmark Suite – An automated open and extensible framework for benchmarking RDF and Property graph databases.

Finally, for answering the third research question (**RQ3**), concerned with the performance evaluation of the proposed mappings via benchmarking both RDF and Property graph databases, we developed the **LITMUS Benchmark Suite**. LITMUS is a comprehensive framework that enables the academicians, researchers, developers, and the organizations who employ databases for the efficient consumption of their data, by allowing a choke-point driven performance comparison and analysis of various RDF and Property graph-based graph data management systems, with respect to different third-party real and synthetic datasets and queries. LITMUS consolidates all the three sub research questions under the umbrella of one unified modular framework, as shown in Figure 1.6, for orchestrating a fair and objective evaluation of RDF and Property graph databases. We carried out all the experiments for evaluating our proposed mappings using the proposed LITMUS framework. We also studied the effect of various internal and external data, query, and system-specific factors on the overall performance of the database systems such as Data quality [45, 46], CPU, RAM, Query typology, Indexing, etc. in our pursuit of answering **RQ3**. We published several papers proposing [35], implementing [47] and demonstrating [48] our work on the LITMUS Benchmark Suite.

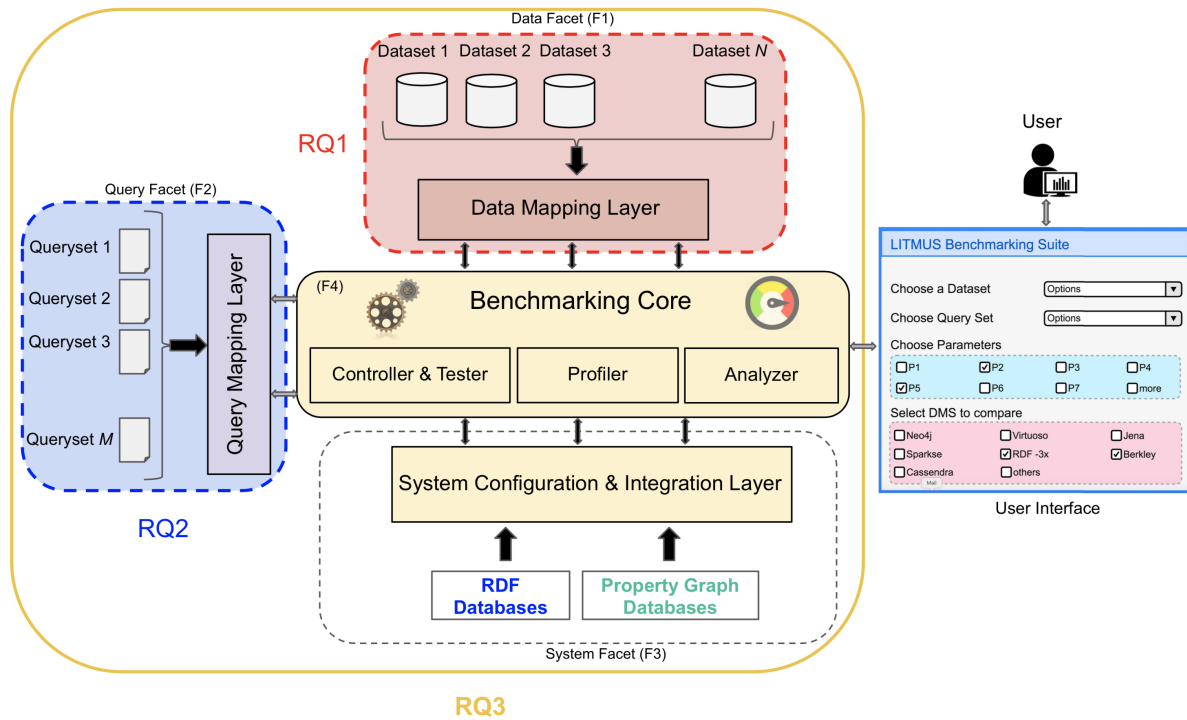


Figure 1.6: The LITMUS architectural framework for a user orchestrated benchmarking RDF and Property graph Databases.

Our work [47] on LITMUS received the Best Research & Innovation Paper Award⁵ at the SEMANTiCS 2017 conference organised in Amsterdam, Netherlands from September 11 to 14, 2017.

1.3.2 Publications

The following list of articles contributed to the scientific basis of the work carried out during the term of this dissertation. These publications consist of Journal, Conference, Workshop and Poster & Demo *peer reviewed* publications, and ArXiv pre-prints (not peer-reviewed) technical reports. Furthermore, as a result of international collaboration encouraged during the Marie-Curie ITN activities, several articles were published, some of which are indirectly related to this dissertation, but have not been included as contributions (listed as miscellaneous papers). For the co-authored publications, the ones with another Ph.D. student, include a description of the individual contribution. Please note that most of the co-authored papers include post-doctoral fellows, professors, master-or-bachelor students, and industry experts. The author of this dissertation (Harsh Thakkar) uses the "we" pronoun throughout the content. However, all of the contributions presented herein, except those with another Ph.D. student, originated from the work of the author exclusively himself. Besides the listed publications, other materials such as public software and datasets were also produced, which have been mentioned in their corresponding chapters. Three publications of the author have received the **Best Paper**

⁵ <https://2017.semantics.cc/awards>

Awards at conferences and workshops, during the term of this dissertation, which have been marked in **red ink** next to the item in the list below.

Journal Papers (peer reviewed):

1. Renzo Angles, **Harsh Thakkar**, and Dominik Tomaszuk. *Mapping RDF Databases to Property Graphs*. In IEEE Access, Vol. 8, 2020. In this publication Renzo and I made equal contributions.
2. Dominik Tomaszuk, Renzo Angles, and **Harsh Thakkar**. *PGO: Describing Property Graphs in RDF*. In IEEE Access, Vol. 8, 2020. In this publication I contributed to the formalisation of the data models derived from my earlier work (above).

Conference Papers (peer reviewed):

3. **Harsh Thakkar**, Renzo Angles, Marko Rodriguez, Stephen Mallette, and Jens Lehmann. *Let's build Bridges, not Walls: SPARQL Querying of TinkerPop Graph Databases with Sparql-Gremlin*. In Proceedings of the IEEE 14th International Conference on Semantic Computing (ICSC), pp. 408-415, San Diego, USA, 2020. [**Best Paper Award**]
4. **Harsh Thakkar**, Dharmen Punjani, Sören Auer, and Maria-Esther Vidal. *Towards an Integrated Graph Algebra for Graph Pattern Matching with Gremlin*. In Proceedings of the 28th International Conference on Database and Expert Systems Applications (DEXA 2017), Lyon, France, pp. 81-91. Springer, 2017.
5. **Harsh Thakkar**, Yashwant Keswani, Mohnish Dubey, Jens Lehmann, and Sören Auer. *Trying Not to Die Benchmarking – Orchestrating RDF and Graph Data Management Solution Benchmarks using LITMUS*. In Proceedings of the 13th International Conference on Semantic Systems (SEMANTiCS 2017), Amsterdam, Nederland, pages 120-127. ACM, 2017. [**Best Paper Award**]
6. Kemele M Endris, Josè M. Gimenez-García, **Harsh Thakkar**, Elena Demidova, Antoine Zimmermann, Christoph Lange, and Elena Simperl. *Dataset Reuse: An Analysis of References in Community Discussions, Publications and Data*. In Proceedings of The 9th International Conference on Knowledge Capture (K-CAP 2017), Austin, Texas, United States, 2017. This work was co-authored with PhD student Kemele Endris (Leibniz University of Hannover). In this paper, my contribution was designing and implementing a setup of experiments for assessing the impact of data quality metrics on the overall performance of the participating RDF databases. This was a derived extension of my original work in 2016 on the assessment of data quality on the linked open datasets.
7. **Harsh Thakkar**. *Towards an Open Extensible Framework for Empirical Benchmarking of Data Management Solutions: LITMUS*. In Proceedings of the 14th Extended Semantic Web Conferences (ESWC 2017), 2017.
8. **Harsh Thakkar**, Kemele M. Endris, Josè M. Gimenez-García, Jeremy Debattista, Christoph Lange, and Sören Auer. *Are Linked Datasets Fit for Open-domain Question Answering? A Quality Assessment*. In Proceedings of the 6th International Conference on

Web Intelligence, Mining and Semantics (WIMS 2016), Nîmes, France, June 13-15, pages 1-12, 2016. This is the foundational work on investigating the impact of data quality of the linked open datasets on the overall performance of the RDF databases which is a part of the third sub-research question (**RQ3**) orchestrating the benchmarking of RDF vs Property graph databases. I received help for conducting a part of the the state of the art survey for identifying existing works on the data quality metrics and performing the experiments.

Workshop Papers (peer reviewed):

9. **Harsh Thakkar**, Maria-Esther Vidal, Sören Auer. *Formalizing Gremlin Pattern Matching Traversals in an Integrated Graph Algebra (extended version)*. In Proceedings of the 2nd International Semantic Web Conference (ISWC) Workshop on Contextualised Knowledge Graphs (CKG), New Zealand, 2019.
10. Renzo Angles, **Harsh Thakkar**, and Dominik Tomaszuk. *RDF and Property Graphs Interoperability: Status and Issues*. In Proceedings of the 13th Alberto Mendelzon International Workshop on Foundations of Data Management, Asunción (AMW 2019), Paraguay, June 3-7, 2019.
11. Vinh Nguyen, Hong Yung Yip, **Harsh Thakkar**, Qingliang Li, Evan Bolton, and Olivier Bodenreider. *Singleton property graph: Adding a semantic web abstraction layer to graph databases*. In Proceedings of the 2nd International Semantic Web Conference (ISWC) Workshop on Contextualised Knowledge Graphs (CKG), New Zealand, 2019.
12. Saeedeh Shekarpour, Kemele M Endris, Ashwini Jaya Kumar, Denis Lukovnikov, Kuldeep Singh, **Harsh Thakkar**, and Christoph Lange. *Question Answering on Linked Data: Challenges and Future Directions*. In Companion Proceedings of the 25th International Conference Companion on World Wide Web (WWW), pages 693-698. 2016.

Poster & Demo Papers (peer reviewed):

13. **Harsh Thakkar**, Dharmen Punjani, Jens Lehmann, and Sören Auer. *Two for one: Querying Property Graph Databases using SPARQL via GREMLINATOR*. In Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and NetworkData Analytics (NDA), page 12, ACM, USA, 2018.
14. Yashwant Keswani, **Harsh Thakkar**, Mohnish Dubey, Jens Lehmann, and Sören Auer. *The LITMUS Test: Benchmarking RDF and Graph Data Management Systems*. In Proceedings of the CEUR-WS (Poster & Demo), SEMANTiCS 2017, Nederland, 2017.

Pre-prints (not peer reviewed):

15. Mohamed Nadjib Mami, Damien Graux, **Harsh Thakkar**, Simon Scerri, Sören Auer, and Jens Lehmann. *The Query Translation Landscape: A Survey*. Pre-print arXiv:1910.03118, pp. 1-25, 2019.
16. **Harsh Thakkar**, Dharmen Punjani, Yashwant Keswani, Jens Lehmann, and Sören Auer.

A Stitch in Time Saves Nine – SPARQL Querying of Property Graphs using Gremlin Traversals. Pre-print arXiv:1801.02911, pp. 1-24, 2018.

Edited Volumes & Online Resources:

17. Reza Samavi, Mariano P. Consens, Shahan Khatchadourian, Vinh Nguyen, Amit P. Sheth, José M. Giménez-García, and **Harsh Thakkar**. *Proceedings of the Blockchain enabled Semantic Web Workshop (BlockSW) and Contextualized Knowledge Graphs (CKG) Workshop* co-located with the 18th International Semantic Web Conference, BlockSW/CKG@ISWC, Auckland, New Zealand, 2019.
18. **Harsh Thakkar**, Renzo Angles, Marko Rodriguez, Stephen Mallette, Dharmen Punjani, Jens Lehmann, Sören Auer. *Gremlinator (sparql-gremlin) Resources*, available online at <https://doi.org/10.6084/m9.figshare.8187110.v3>, 2019.
19. Renzo Angles, **Harsh Thakkar**, and Dominik Tomaszuk. *RDF2PG Experimental Datasets*, available online at <https://doi.org/10.6084/m9.figshare.12021156.v10>, 2020.

Working Papers:

20. **Harsh Thakkar**, Renzo Angles, Marko Rodriguez, Jens Lehmann, Sören Auer. *GREMLINATOR: SPARQL Querying of Property Graph Databases using Gremlin Traversals*. IEEE Access submission, 2020. (in progress)
21. **Harsh Thakkar**, Renzo Angles, and Dominik Tomaszuk. *RDF2PG: Automatic Transformation of RDF to Property Graphs*. Demo paper, *Venue TBD*, 2020. (in progress)

Miscellaneous Papers (peer reviewed):

The following is the list of publications that originated during and are indirectly related to this thesis but are not included in this dissertation.

22. Felipe Quécole, Romao Martines, José M. Giménez-García, and **Harsh Thakkar**. *Towards Capturing Contextual Semantic Information about Statements in Web Tables*. In Joint Proceedings of the International Workshops on Contextualized Knowledge Graphs, and Semantic Statistics (CKGSemStats) co-located with 17th International Semantic Web Conference (ISWC 2018), USA, 2018.
23. Kuldeep Singh, Ioanna Lytra, Maria-Esther Vidal, Dharmen Punjani, **Harsh Thakkar**, Christoph Lange, and Sören Auer. *Qaestro - Semantic-based Composition of Question Answering Pipelines*. In 28th International Conference on Database and Expert Systems Applications (DEXA 2017), France, 2017.
24. José M. Giménez-García, **Harsh Thakkar**, and Antoine Zimmermann. *Assessing Trust with Pagerank in the Web of Data*. In Proceedings of the 3rd International Workshop on Dataset PROFiling and fEDerated Search for Linked Data (PROFILES '16) co-located with the 13th ESWC 2016 Conference, Greece, 2016. [**Best Paper Award**]

The following table presents an overview of the publication statistics of this dissertation. The

full list of publications completed during this dissertation term is available in Appendix B.

<i>Ph.D. Publication Statistics</i>		<i>From 07.2015 to 07.2020 (5 Years)</i>		
Venue	Accepted	In Rev.	Total	Thesis Related
<i>Journal</i>	2	0	2	2 (100%)
<i>Conference</i>	8	0	8	6 (75%)
<i>Workshop</i>	6	0	6	4 (67%)
<i>Poster/Demo</i>	2	0	2	2 (100%)
<i>Arxiv Preprints</i>	6	0	6	6 (100%)
No. of Publications	24	00	24	20 (83%)

1.4 Thesis Outline

This thesis is structured into the following seven chapters:

- **Chapter 1 – Introduction:** introduces the thesis explaining the study’s motivation, main research problem, challenges, and contributions that address each of the three relevant research questions. It also lists the scientific publications and their statistics produced during the term of this dissertation.
- **Chapter 2 – Background & Preliminaries:** introduces the fundamental concepts and background covering the broad fields of Semantic Web, Linked Data, and Property graphs, and Benchmarking that are required to align the context and provide a holistic overview of the research problem.
- **Chapter 3 – Related Work:** summarizes the state-of-the-art in the respective domain of the three research questions, highlighting the shortcomings of the existing works and thereby setting the motivation for the required work done in the context of this thesis.
- **Chapter 4 – Directly Mapping RDF Databases to Property graph Databases:** addresses the first research question (**RQ1**) concerned with the *data interoperability* issue by proposing the direct mappings for transforming any RDF data into a corresponding Property graph, covering both data and schema. It also defines the desired formal properties of the proposed mappings and shows that the mappings are information and semantics preserving. Furthermore, it discusses the implementation and an exhaustive empirical evaluation of the proposed direct mappings.
- **Chapter 5 – Gremlinator: Querying Property graph Databases using SPARQL:** addresses the second research question (**RQ2**) concerned with the *query interoperability* issue by proposing a direct mapping approach (**Gremlinator**) for translating SPARQL queries into a corresponding pattern matching Gremlin traversals. In doing so, it first formalizes the Gremlin traversal language operators using a consolidated graph relational

algebra. Furthermore, it discusses the implementation and integration of **Gremlinator** into the popular Apache TinkerPop framework, its use cases and presents an exhaustive empirical evaluation of the proposed query translation tool **Gremlinator** over both RDF and Property graph databases thereby demonstrating that the proposed mapping is query preserving.

- **Chapter 6 – Automatic Benchmarking of RDF and Graph Databases:** addresses the third and final research question (**RQ3**) concerned with the *database benchmarking* issue by presenting a novel benchmarking framework – **LITMUS Benchmark Suite**. **LITMUS** is a first of its kind open, extensible, and reusable benchmarking framework for benchmarking both RDF and Property graph databases. This chapter explains the design and implementation of **LITMUS** and how it integrates all the three research questions of this dissertation under one umbrella.
- **Chapter 7 – Conclusion & Future Directions:** finally concludes the work of this dissertation summarizing our core contributions and its impact on the broader community and lays out the direction of future work.

Background and Preliminaries

In this chapter we present and discuss the foundation technologies and formal concepts that are required to understand the work presented in the context of this dissertation. This chapter is structured into three main sections, as follows: Section 2.1 provides an overview of Semantic Web Technologies and Linked Data, i.e W3C standard RDF data model, W3C standard SPARQL query language, OWL and RDF triplestores (RDF databases). We do so using illustrative examples that explain these basic concepts of Semantic Web technology stack. Section 2.2 introduces the reader to the Graph database/technology stack and introduces the notions of Property graph data model, the Gremlin traversal language (also using illustrative examples) and the Apache TinkerPop graph computing framework. Section 2.3 introduces the concept of Knowledge Graphs (KGs) and highlight their importance summarizing the value they have created (in terms of revenue and applicability) in the modern day information technology industry. We also discuss the different types of KGs that are being currently used to address a variety of real world data-driven problems. Finally, Section 2.2 having introduced these concepts summarizes this chapter.

2.1 Semantic Web & Linked Data

The *Semantic Web* is an evolving extension of the Web that allows data to be shared and consumed across application, enterprise, and community boundaries in an intelligent manner.¹ The idea of Semantic Web was first proposed by Berners-Lee et al. [49] in 2001 with an intent of adding context information within the data itself that described concepts in the real world. The technology stack of the Semantic Web consists of a set of standards: RDF (the data model), OWL (the ontology language) and SPARQL (the query language). We introduce each of these Semantic Web terminologies in the coming subsections aligning the reader with the context of the work presented in this thesis.

The main advantage of Semantic Web lies in its powerful and structured representation of data for its easy publishing and consumption. Tim Berners-Lee has proposed a five star data

¹ <http://www.w3.org/2001/sw/>

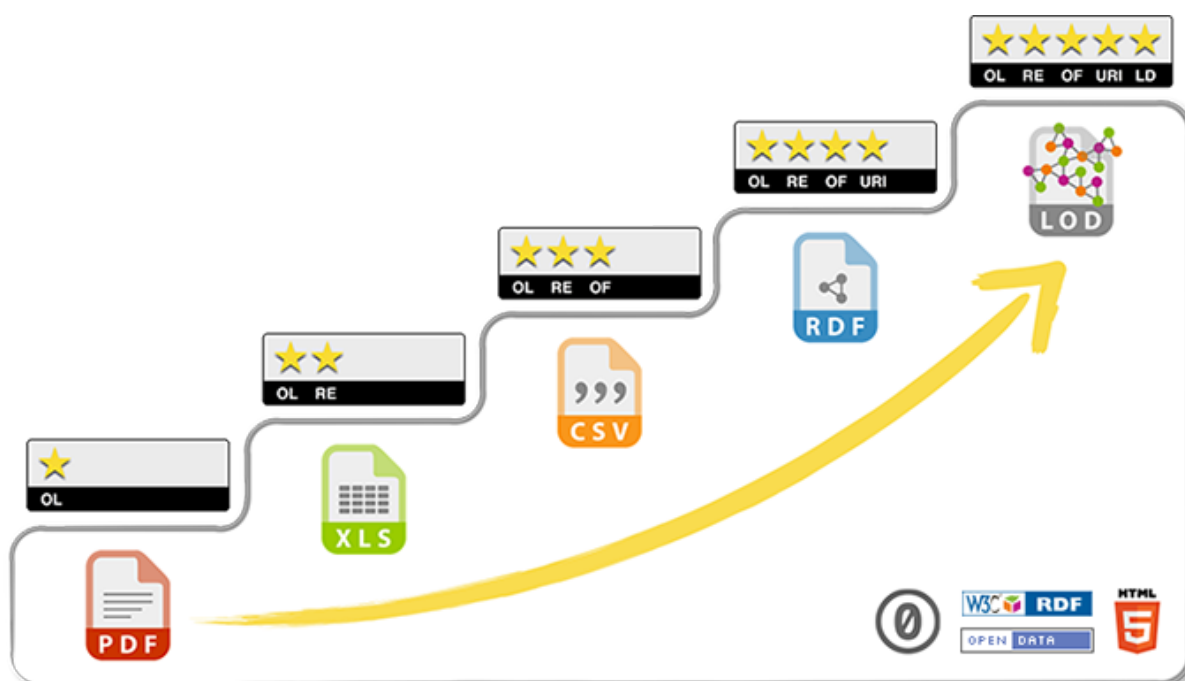


Figure 2.1: The Five Star Linked Open Data Publishing Principles. The PDF, XSL, CSV, RDF, and Linked Open Data (LOD) represent the five levels of data publishing schemes in increasing order of connectivity, accessibility and reusability.

publishing principle or scheme² for open data as illustrated in Figure 2.1³.

In the five star scheme, the Linked open Data (LOD) constitutes of the highest order. In order to promote reusability and embed semantics in linked data, there are four principles proposed by Tim Berners-Lee to adhere to [50]:

- Using URIs as names of the things (i.e. resources);
- Using HTTP URIs for dereferencing such that user can look up for these names easily;
- Looking up a URI should provide useful information, using the standards (RDF, SPARQL);
- Interlinking URIs so that people can discover more things.

The crux of the five star principles is to promote openness and interlinking of information over the Web. Following these principles, more and more data is published on the Web which over the years has lead to the development of *Linked Open Data Cloud* (LOD Cloud) [51], a network of interconnected 5 star datasets. For instance, DBpedia [52] – which is a structured version of Wikipedia⁴; Wikidata [53] – which is an uniform source for openly published Wikipedia articles; YAGO⁵ [54] – which as an open source high-quality knowledge base extracted from Wikipedia

² 5 Star Linked Data Principles https://www.w3.org/2011/gld/wiki/5_Star_Linked_Data

³ Image source https://commons.wikimedia.org/wiki/File:5-star_deployment_scheme_for_Open_Data.png

⁴ <http://www.wikipedia.org>

⁵ YAGO <https://www.mpi-inf.mpg.de/departments/databases-and-information-systems/research/>

and other sources, are a few very popular examples.

2.1.1 RDF Data Model and OWL

The **Resource Description Framework (RDF)** is a well-known W3C standard, which is used for data modeling and encoding machine readable content on the Web [55] and within intranets. RDF provides the technological foundation for expressing the meaning (information, e.g. description) of terms and concepts (resources, e.g. coffee machine) in the Web in a form that computers (machines, e.g. applications) can readily process [49]. Resources, in this context means anything that exists in the real world such as, images, text/documents, multimedia, physical objects, etc⁶. RDF data is represented using a variety of data formats such as Turtle [56]⁷, N-Triples⁸, N3⁹, N-Quads¹⁰, TriG¹¹, amongst others are a few important examples.

Assume the existence of three disjoint infinite sets: the set I of resource identifiers represented as IRIs, the set B of anonymous resources called blank nodes, and the set L of simple atomic values called literals. An *RDF triple* is a triple $(s, p, o) \in (I \cup B) \times I \times (I \cup B \cup L)$ where s is the *subject*, p is the *predicate* and o is the *object*. An *RDF Graph* is a set of RDF triples. To illustrate the notion of RDF triples, consider the following example.

Example 2.1.1. Consider a subset of RDF triples from DBpedia [52] describing the relationship between Elon Musk and Tesla Incorporation represented in the RDF turtle [56] format. Figure 2.2 shows a graphical representation of these RDF triples, i.e. an RDF graph.

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix dbr: <http://dbpedia.org/resource/>.
@prefix dbo: <http://dbpedia.org/ontology/>.
@prefix dbp: <http://dbpedia.org/property/>.
dbr:Elon_Musk rdf:type dbo:Person
dbr:Elon_Musk dbp:birthDate "1971-06-28"
dbr:Elon_Musk dbp:birthName "Elon Reeve Musk"
dbr:Elon_Musk dbp:ceo dbr:Tesla_Inc
dbr:Tesla_Inc rdf:type dbo:Organisation
dbr:Tesla_Inc dbp:name "Tesla, Incorporation"
dbr:Tesla_Inc dbp:date "2003-07-01"
```

In Semantic Web *vocabularies* (or RDF vocabularies) are used to represent the structural information of the resources. RDF Schema [8] (RDFS), also know as the RDF vocabulary

yago-naga/yago/

⁶ The term resources and entities can be considered as synonyms in this context.

⁷ Turtle Specification <https://www.w3.org/TR/turtle/>

⁸ N-Triples Specification <https://www.w3.org/TR/n-triples/>

⁹ N3 Specification <https://www.w3.org/TeamSubmission/n3/>

¹⁰ N-Quads Specification <https://www.w3.org/TR/n-quads/>

¹¹ TriG Specification <https://www.w3.org/TR/trig/>

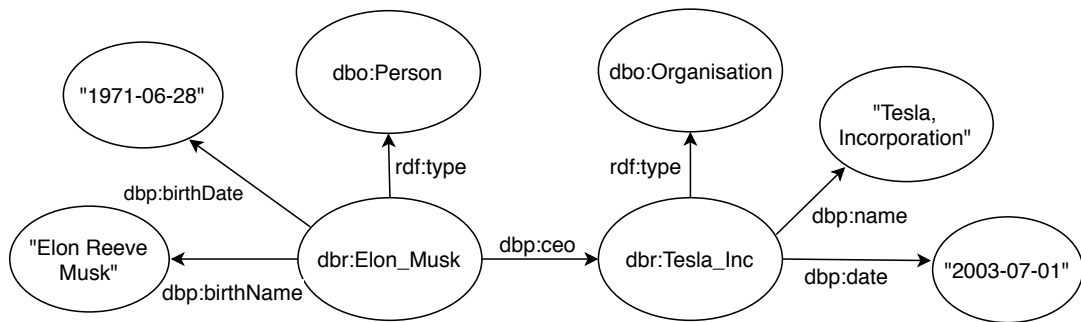


Figure 2.2: A graphical illustration of an RDF graph describing information about Elon Musk and Tesla Incorporation as shown in example 2.1.1

description language¹², is one such standardized light weight vocabulary that allows describing the structure and relationships between real world entities using classes, entities and properties which are defined by the users. The goal behind having such a standardized and controlled environment for defining and describing resources is to promote reuse and ease of integration (linking to other sources).

The **Web Ontology Language (OWL)** is language using which the meaning of terms, their relationships and consistency between them is represented. Such a representation is referred to as an Ontology [57]. OWL extends RDFS by allowing users to represent more complex relationships between RDF resources, i.e. ontologies. As defined in the words of Struder et al. [58] "An ontology is a formal, explicit specification of a shared conceptualization". Typically, ontologies are designed to evolve with the respective domain whose information they capture with time and also as more heterogeneous data sources are added.

Example 2.1.2. The RDF schema information corresponding to the RDF triples in Example 2.1.1, is represented in the turtle [56] format, below:

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix dbr: <http://dbpedia.org/resource/>.
@prefix dbo: <http://dbpedia.org/ontology/>.
@prefix dbp: <http://dbpedia.org/property/>.
dbr:Elon_Musk rdf:type dbo:Person
dbr:Tesla_Inc rdf:type dbo:Organisation
dbo:Person rdf:type rdfs:Class
dbo:Person rdfs:label "Person"
dbo:Organisation rdf:type rdfs:Class
dbo:Organisation rdfs:label "Organisation"
dbp:birthDate rdf:type rdf:Property
dbp:birthDate rdfs:label "birth date"
dbp:birthDate rdfs:domain dbo:Person
dbp:birthDate rdfs:range rdfs:Literal
```

¹² RDF Vocabulary Description Language <https://www.w3.org/2001/sw/RDFCore/Schema/200203/>

```

dbp:birthName rdf:type rdf:Property
dbp:birthName rdfs:label "birth name"
dbp:birthName rdfs:domain dbo:Person
dbp:birthName rdfs:range rdfs:Literal
dbp:name rdf:type rdf:Property
dbp:name rdfs:label "name"
dbp:name rdfs:domain dbo:Organisation
dbp:name rdfs:range rdfs:Literal
dbp:date rdf:type rdf:Property
dbp:date rdfs:domain dbo:Organisation
dbp:date rdfs:range rdfs:Literal

```

While this example is based on the information obtained from Dbpedia, assume that we have same information but from another source, say Wikidata. Here the entity Elon Musk is identified using the identifier Q317521 and Tesla Incorporation using Q478214. Furthermore, the relations *birth name*, *ceo* and *birth date* are identified using properties P1477, P169 and P569 respectively as shown in the Example 2.1.3 below:

Example 2.1.3. Consider a subset of RDF triples from DBpedia [52] describing the relationship between Elon Musk and Tesla Incorporation represented in the RDF turtle [56] format.

```

@prefix : <https://www.wikidata.org/wiki#>
@prefix _: <https://www.wikidata.org/wiki/Property#>
_:Q317521 _:P31 :Q5
_:Q317521 _:P569 "1971-06-28"
_:Q317521 _:P1477 "Elon Reeve Musk"
_:Q317521 _:P169 :Q478214
_:Q478214 _:P31 :Q167037
_:Q478214 _:P1448 "Tesla, Incorporation"
_:Q478214 _:P571 "July 2003"

```

Now, using OWL it is possible to add specific semantics stating that the both the Elon Musk entities in DBpedia and Wikidata are in fact the same person/resource/object, i.e. they point to the same meaning. This is denoted via the `owl:sameAs` property – `:Q317521 owl:sameAs dbr:Elon_Musk`. Similarly, for the Tesla Incorporation – `Q478214 owl:sameAs dbr:Tesla_Inc.`

2.1.2 SPARQL Query Language

SPARQL is a W3C recommendation since 2008 and the defacto standard query language for RDF databases or triplestores (more on this in the next in Section 2.1.3), which is based on graph pattern matching. Like SQL for relational databases, SPARQL is also a declarative query language that relies heavily on joins. SPARQL 1.0 [59] defines basic types of graph patterns, filter conditions (e.g. equalities), solution modifiers, and query forms. SPARQL 1.1 [7] extends the first version with operators for aggregation, sub-queries and path queries. We briefly discuss

the syntax and semantics of the SPARQL query language in the next sections. The syntax of a SPARQL query will be presented by using the formalism presented in [60], but in agreement with the W3C specifications.

SPARQL Syntax

Assume the existence of an infinite set V of variables disjoint from I , B and L . A *filter constraint* is defined recursively as follows: (i) If $?X, ?Y \in V$ and $c \in I \cup L$ then $(?X = c)$ and $(?X = ?Y)$ are atomic filter constraints; (ii) If C_1 and C_2 are filter constraints then $(!C_1)$, $(C_1 \parallel C_2)$ and $(C_1 \ \&\& \ C_2)$ are *complex filter constraints*.

A SPARQL *graph pattern* is defined recursively as follows: A tuple from $(I \cup L \cup V) \times (I \cup V) \times (I \cup L \cup V)$ is a graph pattern called a *triple pattern*. If P_1, P_2 are graph patterns and C is a filter constraint then $\{P_1 \text{ JOIN } P_2\}$, $\{P_1 \text{ UNION } P_2\}$, $\{P_1 \text{ OPTIONAL } P_2\}$, and $\{P_1 \text{ FILTER } C\}$ are graph patterns.

A *solution modifier* is given by any of the following clauses: DISTINCT allows to eliminate duplicates; ORDER BY establishes the order of the solutions; GROUP BY allows to group the solutions according to a given criteria; OFFSET defines a starting point in the list of solutions; and LIMIT defines an upper bound on the number of solutions returned.

A *query form* defines the output of a query, and there are four types: SELECT returns a binding table, ASK returns a boolean value, and CONSTRUCT and DESCRIBE return an RDF graph. We concentrate our interest on SELECT queries.

Assume that $?X_1 \dots ?X_n$ represents a list of variables such that $n > 0$. A *SPARQL query* will be represented as a tuple $Q^S = \{S, W, GB, OB, LO\}$ where: S is an expression SELECT DISTINCT $?X_1 \dots ?X_n$; W is an expression WHERE P where P is a graph pattern; GB is an expression GROUP BY $?X_1 \dots ?X_n$; OB is an expression ORDER BY β where β is $?X$, ASC($?X$) or DESC($?X$); L is an expression LIMIT n OFFSET m , or simply LIMIT n , where n and m are positive integers. The operator DISTINCT and the clauses GB , OB and LO are optional. The Listings 2.1 and 2.2 present examples different types of SPARQL queries. The first query retrieves the answer to the question "What is the birth date of Elon Musk?". Whereas, the second query expects a boolean answer to the question *Is Elon Musk is the Owner of Tesla Incorporation?*.

```
PREFIX dbr: <http://dbpedia.org/resource/>
PREFIX dbp: <http://dbpedia.org/property/>
SELECT DISTINCT ?birthdate WHERE {
dbr:Elon_Musk dbp:birthDate ?birthdate . }
```

Listing 2.1: An Example of SPARQL SELECT query

```
PREFIX dbr: <http://dbpedia.org/resource/>
PREFIX dbp: <http://dbpedia.org/property/>
ASK WHERE { dbr:Elon_Musk dbp:owner dbr:Tesla_Inc . }
```

Listing 2.2: An Example of SPARQL ASK query

{ dbr:Elon_Musk dbp:birthDate ?birthdate }

?birthdate
1971-06-28
"1971-6-28"^^<http://www.w3.org/2001/XMLSchema#date>

Figure 2.3: The evaluation of the SPARQL SELECT query presented in listing 2.1.

Next, we present a short summary of the evaluation of a SPARQL query and its semantics.

SPARQL Semantics

The evaluation of a SPARQL query is defined in terms algebra operations over binding tables. A *binding table* is a tabular structure where the head contains variables, and the body contains bindings for such variables, i.e. IRIs and literals. Each row of the table is called a solution mapping and contains an specific result of pattern matching. A solution mapping is a partial function $\mu : V \rightarrow (I \cup L)$, i.e. a list of variable \rightarrow value assignments. The result of evaluating a triple pattern over an RDF graph is a multiset of solution mappings. For example, the SPARQL query presented in Listing 2.1 looks for the birth date of Elon Musk via the DBpedia property “dbp:birthDate”. Hence, the result of this query is a solution set of all the *distinct* values that exist for the relation “dbp:birthDate” in DBpedia for that particular mapping μ such that $\mu(?birthdate) = “1971-06-28”$, $\mu(?birthdate) = “1971-6-28”$ ^^<http://www.w3.org/2001/XMLSchema#date>. Figure 2.3, presents the binding table of the evaluation of SPARQL query form listing 2.1.

The evaluation (semantics) of a SPARQL query is defined by operations over multisets of solution mappings. Specifically, the JOIN operator merges two multisets based on the “join” of common variables, the UNION operator computes the union of multisets, the OPTIONAL operator extends a multiset with the variables of another multiset when possible (following outer join semantics), the FILTER operator selects the mappings that satisfy a filter constraint, the SELECT operator projects the variables of a multiset, the DISTINCT operator removes duplicated solution mappings, the GROUP BY operator allows to group and aggregate solution mappings. The operators ORDER BY, LIMIT and OFFSET allows to format the final multiset (SQL style).

We show brevity in discussing further about the semantics and expressivity of SPARQL query language and point the interested reader to the works [60–63] that present an extensive evaluation of the same. Next, we discuss about RDF databases and present an overview of their history of development and characteristics.

2.1.3 RDF Databases

RDF databases (or RDF Triplestores) are essentially the purpose-built databases optimized to store and retrieve RDF data or triples. Much like relational databases, RDF triplestores also provide a wide range of functionalities such as large-scale data storage, federated querying, reasoning over the data, etc. RDF triplestores can be queried using the W3C recommendation SPARQL query language [62] (cf. Section 2.1.2). Recently, the increasing amount of RDF data due to the rise of popularity of the Semantic Web has motivated the development of a variety of RDF triplestores depending on the application use case [64]. RDF triplestores are used in a variety of domains, social networking, cryptocurrency and finance, pharmaceutical and biological domain, chemistry, etc., to name a few.

There exist a variety of implementations of RDF triplestores, which can essentially be grouped into three categories – (a) relational-database based, (b) native based, and (c) hybrid. Relational databases based triplestores are typically built by extending the commercial relational database engines. Their main advantage is their highly optimised methods for storing and querying relational data due to the three decades of academic research behind them. Examples of relational based triplestores are Oracle 12c¹³, IBM DB2¹⁴, Redland¹⁵, etc. In these type of triplestores, data storage and querying is supported via a *data and query translation* client or wrapper such as [65–67]. The native based triplestores are built from scratch, requiring far more effort as compared to the relational-based triplestores. The native RDF triplestores are optimised to store and retrieve RDF data efficiently and have shown to have an edge over relational-based triplestores for performance [68]. One of the reason is that they avoid the intermediate data and query conversion process. Examples of native triplestores are Jena TDB¹⁶, 4store¹⁷, Stardog¹⁸, etc. The third type of triplestores is the hybrid-based or multi-model based triplestores. Hybrid-based triplestores support both relational-based and RDF-based architectures for storing and querying RDF data. Both the relational and RDF data is stored either physically or logically (as visualised relational graphs) in the hybrid scheme. Examples of hybrid triplestores are Blazegraph¹⁹, Virtuoso²⁰, MarkLogic²¹, etc.

Figure 2.4 present the ranked list of a variety of existing RDF triplestores as reported on DB-Engines ranking website. Furthermore, a complete list of comparing various RDF triplestores can be referred from https://en.wikipedia.org/wiki/Comparison_of_triplestores.

In the next section we briefly discuss about the Graph databases with respect to their data model, query languages – Gremlin in particular, and their similarities and differences with RDF triplestores.

¹³ <https://www.oracle.com/database/12c-database/>

¹⁴ <https://www.ibm.com/analytics/db2>

¹⁵ <http://librdf.org/>

¹⁶ <https://jena.apache.org/documentation/tdb/>

¹⁷ <https://github.com/4store/4store>

¹⁸ <https://www.stardog.com/>

¹⁹ <https://blazegraph.com/>

²⁰ <http://vos.openlinksw.com/owiki/wiki/VOS>

²¹ <https://www.marklogic.com/>





























Rank			DBMS	Database Model	Score		
Dec 2019	Nov 2019	Dec 2018			Dec 2019	Nov 2019	Dec 2018
1.	1.	1.	MarkLogic	Multi-model 	12.47	-0.36	-1.81
2.	2.	2.	Virtuoso 	Multi-model 	2.64	+0.00	+0.16
3.	3.	3.	Apache Jena - TDB	RDF	2.63	+0.06	+0.57
4.	4.	4.	Amazon Neptune	Multi-model 	1.57	-0.03	+0.35
5.	5.	 6.	GraphDB 	Multi-model 	1.15	+0.01	+0.43
6.	6.	 5.	AllegroGraph 	Multi-model 	0.87	+0.00	-0.04
7.	7.	7.	Stardog 	Multi-model 	0.78	+0.02	+0.25
8.	8.	8.	Blazegraph	Multi-model 	0.62	+0.04	+0.23
9.	9.	 11.	Redland	RDF	0.56	+0.03	+0.29
10.	10.	10.	RDF4J	RDF	0.40	+0.02	+0.13
11.	11.	 12.	4store	RDF	0.37	+0.02	+0.12
12.	 13.	 14.	RedStore	RDF	0.19	0.00	+0.07
13.	 15.	 15.	Mulgara	RDF	0.17	+0.03	+0.09
14.	14.	 13.	CubicWeb	RDF	0.14	0.00	+0.02
15.	 17.	 19.	Strabon	RDF	0.11	+0.01	+0.06
16.	16.	16.	AnzoGraph 	Multi-model 	0.09	-0.01	+0.02
17.	 18.	17.	BrightstarDB	RDF	0.09	+0.00	+0.02
18.	 19.	18.	Dydra	RDF	0.08	+0.00	+0.02
19.	 20.	 20.	SparkleDB	RDF	0.06	+0.01	+0.03

Figure 2.4: The ranking of a variety of RDF triplestores as reported by the db-engines ranking website – <https://db-engines.com/en/ranking> based on their popularity and other technical criteria.

2.2 Graph Databases

In Computer Science, *Graph* databases make use of graph structures to represent, store, and query complex network of heavily interconnected data using nodes (vertices), edges (relationships) and properties (attributes) [69]. Graphs are valued distinctly, when it comes to choosing formalisms for modelling real-world scenarios such as biological, transport, communication, financial and social networks due to their intuitive data model [70]. Nodes represent the entities, whereas edges represent the relationships between them. Properties are used to represent additional meta-data or information regarding a particular nodes, edge or a property. A graph database is a part of the NoSQL databases which have been developed to address the limitations of the relational databases. Graph databases store the data items as collections of entities and relationships, thus querying these relationships within the database is generally fast [42] as compared to relational databases. This is because graph databases allow index-free adjacency, that is every element contains a direct pointer to its adjacent elements, and therefore necessity for index lookups is mitigated. Graph databases are also available for generalised applications and specialised application domains such as RDF triplestores, that can store any graph-like data. Examples of graph databases are Neo4J[71], TinkerGraph²², Amazon Neptune²³, JanusGraph²⁴, InfiniteGraph²⁵, etc. Figure 2.5 presents a ranked list of Graph databases based on their

²² TinkerGraph <http://tinkerpop.apache.org/docs/current/reference/#tinkergraph-gremlin>

²³ Amazon Neptune <https://aws.amazon.com/neptune/>

²⁴ JanusGraph <https://janusgraph.org/>

²⁵ InfiniteGraph <https://www.objectivity.com/products/infinitegraph/>

Dec 2019	Nov 2019	Dec 2018	DBMS	Database Model	Dec 2019	Nov 2019	Dec 2018
1.	1.	1.	Neo4j	Graph	50.56	+0.03	+5.00
2.	2.	2.	Microsoft Azure Cosmos DB	Multi-model	31.43	-0.54	+8.06
3.	3.	3.	OrientDB	Multi-model	4.93	-0.46	-1.15
4.	4.	4.	ArangoDB	Multi-model	4.87	-0.14	+0.60
5.	5.	5.	Virtuoso	Multi-model	2.64	+0.00	+0.16
6.	6.	6.	JanusGraph	Graph	1.75	-0.05	+0.49
7.	7.	7.	Amazon Neptune	Multi-model	1.57	-0.03	+0.35
8.	8.	10.	GraphDB	Multi-model	1.15	+0.01	+0.43
9.	11.	8.	Giraph	Graph	1.04	+0.03	+0.04
10.	10.	14.	TigerGraph	Graph	0.96	-0.05	+0.49
11.	9.	11.	Dgraph	Graph	0.95	-0.09	+0.37
12.	12.	9.	AllegroGraph	Multi-model	0.87	+0.00	-0.04
13.	13.	12.	Stardog	Multi-model	0.78	+0.02	+0.25
14.	14.	18.	FaunaDB	Multi-model	0.69	+0.08	+0.46
15.	15.	15.	Blazegraph	Multi-model	0.62	+0.04	+0.23
16.	16.	13.	Graph Engine	Multi-model	0.58	+0.01	+0.08
17.	18.	17.	InfiniteGraph	Graph	0.39	+0.01	+0.08
18.	19.	20.	FlockDB	Graph	0.28	+0.01	+0.11
19.	21.	23.	GRAKN.AI	Multi-model	0.23	+0.02	+0.10
20.	22.	19.	HyperGraphDB	Graph	0.19	0.00	+0.02
21.	23.	24.	AgensGraph	Multi-model	0.16	+0.00	+0.05
22.	24.	22.	Sparksee	Graph	0.15	0.00	+0.02
23.	25.	26.	TinkerGraph	Graph	0.13	+0.01	+0.04
24.	26.	30.	HGraphDB	Graph	0.11	-0.01	+0.06
25.	28.	29.	GraphBase	Graph	0.11	+0.02	+0.05
26.	27.	28.	AnzoGraph	Multi-model	0.09	-0.01	+0.02
27.	29.		HugeGraph	Graph	0.07	+0.00	
28.	31.	27.	VelocityDB	Multi-model	0.07	0.00	-0.02
29.			Nebula Graph	Graph	0.06		
30.	30.	25.	Memgraph	Graph	0.05	-0.02	-0.06
31.	32.		TerminusDB	Graph, Multi-model	0.02	+0.01	
32.	33.		Weaviate	Graph, Multi-model	0.00	+0.00	
33.	33.		FlureeDB	Graph	0.00	±0.00	

Figure 2.5: The ranking of a variety of Graph databases as reported by the db-engines ranking website – <https://db-engines.com/en/ranking> based on their popularity and other technical criteria.

popularity in the software industry.

The underlying data model of the Graph databases is the graph data model. There exist a variety of graph data models, such as – (i) edge-labeled graph, (ii) property graph (aka labeled property graph or LPG), (iii) RDF, are examples of very popular graph data models widely in use. The main difference between these models is the way data is modeled and the support for semantics and meta-data. The **Edge-labeled graph** data model is based on a directed labeled multigraph, where there is no concept of properties. The **Property Graph** (PG) data model is based on directed labeled multigraphs where nodes and edges can contain a collection of key-value pairs called properties. The notion of schema for a property graph database has not been developed, but some systems use the notions of node types and edge types in order to enforce a schema [72]. The **RDF** data models is a special kind of data model that relies on RDF data (cf. Section 2.1.1). The RDF data model can be interpreted as a edge-labeled multigraph

which allows three types of nodes (web resources, anonymous resources called Blank Nodes, and simple values called Literals) and two types of edges (data properties that relate resource nodes with literals, and object properties that relate two resource nodes). Furthermore, RDF allows supports representing meta-data via the use of RDF reification. Unlike the RDF data model has SPARQL, the PG data model does not have a standardised query language. However, the TinkerPop Gremlin graph traversal query language [16, 17], Cypher query language [15] and GraphQL²⁶, are very popular amongst the existing NoSQL graph database community. Furthermore, there are also on-going efforts to propose standard for the graph query languages such as GQL²⁷ which a SQL-based extension for query property graphs.

Next, we briefly discuss the Property graph data model and the Gremlin traversal language. For supporting interoperability between RDF and PG databases, the scope of this thesis, we have selected Gremlin as target language, since it is more general than, e.g. CYPHER, as it is supported by a wide range of property graph databases (including both OLTP and OLAP processors, more on this in Section 2.2.2). Moreover, Gremlin supports both the imperative (graph traversal) and declarative (graph pattern matching) style of querying. Lastly, the *Apache TinkerPop* framework and Gremlin represent a virtual query machine, thus providing a flexible abstraction layer between graph data storage and querying (analogous to how various Java Virtual Machine languages can be compiled to be executed in the Java Runtime Environment). We talk more about this in Section 2.2.3.

2.2.1 Property Graph Data Model

A *Property Graph* is a directed labeled multigraph whose main characteristic is that nodes (or vertices) and edges can contain zero or more properties. A property is represented as a name:value pair. Nodes, edges and properties could have labels which are used to describe their role or type in the data domain.

Assume two disjoint infinite sets: the set \mathcal{L} of element labels, and the set \mathcal{V} of primitive values. A property graph can be defined as a tuple $G^P = (N, E, P, \lambda, \delta, \rho, \sigma)$ where N is a finite set of nodes, E is a finite set of edges, P is a finite set of properties, λ is a (total) labeling function $(N \cup E \cup P) \rightarrow \mathcal{L}$, δ is a (total) incidence function $E \rightarrow (N \times N)$, ρ is a (total) property-assignment function $P \rightarrow (N \cup E)$, and σ is a (total) value-assignment function $P \rightarrow \mathcal{V}$.

Property graph data is represented in a variety of data serialization formats such as GraphML [73], GraphSON²⁸, kryo²⁹, GML[74], etc. To illustrate the notion of a Property graph, consider the following example.

Example 2.2.1. Consider an example of a Property graph data representation, related to Elon Musk and Tesla Incorporation, represented using the GraphML [73] format below:

```
<?xml version="1.0" encoding="UTF-8"?>
```

²⁶ GraphQL <https://graphql.org/>

²⁷ GQL standard <https://www.gqlstandards.org/>

²⁸ GraphSON serialization <http://tinkerpop.apache.org/docs/3.4.1/dev/io/#graphson>

²⁹ kryo serialization <https://github.com/EsotericSoftware/kryo>

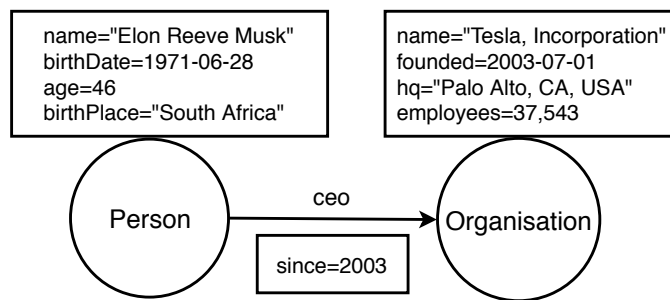


Figure 2.6: An example of a Property graph representing the information about Elon Musk and Tesla Incorporation.

```

<graphml xmlns="http://graphml.graphdrawing.org/xmlns"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
<key id="a0" for="node" attr.name="label" attr.type="string"/>
<key id="a1" for="node" attr.name="name" attr.type="string"/>
<key id="a2" for="node" attr.name="birthDate" attr.type="Date"/>
<key id="a3" for="node" attr.name="age" attr.type="int"/>
<key id="a4" for="node" attr.name="birthPlace" attr.type="string"/>
<key id="a5" for="node" attr.name="founded" attr.type="string"/>
<key id="a6" for="node" attr.name="hq" attr.type="string"/>
<key id="a7" for="node" attr.name="employees" attr.type="int"/>
<key id="a8" for="edge" attr.name="label" attr.type="string"/>
<key id="a9" for="edge" attr.name="since" attr.type="int"/>
<graph id="G" edgedefault="directed">
<node id="n0">
<data key="a0">Person</data>
<data key="a1">Elon Reeve Musk</data>
<data key="a2">1971-06-28</data>
<data key="a3">46</data>
<data key="a4">South Africa</data></node>
<node id="n1">
<data key="a0">Organisation</data>
<data key="a1">Tesla, Incorporation</data>
<data key="a5">2003-07-01</data>
<data key="a6">Palo Alto, CA, USA</data>
<data key="a7">37,543</data></node>
<edge id="e0" source="n0" target="n2">
<data key="a8">ceo</data>
<data key="a9">2003</data>
</edge>
</graph>
</graphml>
  
```

Figure 2.6 shows a graphical representation of a Property graph that describes the information about Elon Musk and Tesla Incorporation as shown in example 2.2.1. Note that each node contains a label which identifies its type (i.e. *Person* and *Organisation*), and one or more properties (i.e. *birthName*, and *age*). On the other hand, each edge contains a label which defines its type (*ceo*), and includes a *since* property.

2.2.2 Gremlin Traversal Language and Machine

Graph analysis tools have turned out to be one of pioneering applications in understanding these natural and man-made networks [3]. Graph analysis is carried out using graph processing techniques which ultimately boil down to efficient graph query processing. Graph Pattern Matching (GPM), also referred to as the sub-graph matching is the foundational problem of graph query processing. Many vendors have proposed a variety of (proprietary) graph query languages to demonstrate the solvability of graph pattern matching problem. These modern graph query languages focus either on *traversal*, where traversers move over vertices and edges of a graph in a user defined fashion or on *pattern-matching*, where graph patterns are matched against the graph database.

Gremlin [17] is one such modern graph query language, with a distinctive advantage over others that allows both pattern matching (declarative) and graph traversal (imperative) style of querying over property graphs. Gremlin is part of the *Apache TinkerPop* graph computing framework³⁰ (more on the TinkerPop graph computing framework in Section 2.2.3). Using Gremlin implies that a user can reap benefits of both declarative and imperative matching style within the same framework. Furthermore, conducting GPM in Gremlin can be of crucial importance in cases such as:

- Querying very large graphs, where a user is not completely aware of certain dataset-specific statistics or structure of the graph (e.g., the number of specific type of edges or nodes existing in the graph);
- Creating optimal query plans, without the user having to dive deep into traversal optimization strategies.
- In application-specific settings such as a question answering [75], users express information needs (e.g., natural language questions) which can be better represented as graph pattern matching problems than path traversals.

Gremlin is based on computing graph traversals over a property graph, i.e. the act of visiting nodes and edges in an alternating manner (in some algorithmic fashion) [70]. In this sense, a graph pattern matching query in Gremlin can be perceived as a path traversal [76].

³⁰ Apache TinkerPop Project (<https://tinkerpop.apache.org/>)

Gremlin Syntax

A Gremlin query is called a path traversal. A *path traversal*, denoted by the symbol Ψ , is composed of an ordered list of steps called the single-step traversals. A *single-step traversal* (SST), denoted by ψ_s , is an atomic operation of function which is applied over the elements in the target graph (i.e. nodes and edges). For example, the Listing 2.3 shows a Gremlin query where `V()`, `.values(...)`, `.has(...)`, `.out(...)` are single-step traversals.

```
g.V().has("name","Elon Reeve Musk").out("ceo").values("name")
```

Listing 2.3: Gremlin query (*imperative*) for the question “Elon Musk is the CEO of which organisation?”

Gremlin includes a large list of traversal operators whose syntax and use is described in the TinkerPop3 documentation³¹. Gremlin supports a variety of operators such as:

- `match` contains a collection of traversal patterns that must hold true, allowing to express pattern matching in a declarative form, i.e. enabling the declarative construct;

```
g.V().match(  
  .as("x").has("name","Elon Reeve Musk"),  
  .as("x").out("ceo").values("name").as("y")).select("y")
```

Listing 2.4: Gremlin query (*declarative*) for the question “Elon Musk is the CEO of which organisation?”

- `union` allows to merge the results of an arbitrary number of traversals;
- `optional` allows to return the result of the specified traversal if it yields a result, else it returns the calling element;
- `where` allows to filter the current object based on either the object itself or the path history of the object, and can include operators like `eq` and `neq` to evaluate equalities or inequalities, and operators `and`, `or` and `not` to introduce boolean conditions;
- `group.by` allows to organize (or group) the objects according to some function of the object (e.g. a property);
- `order.by` allows to sort the objects;
- `range(begin,end)` allows to restrict the number of objects obtained by a traversal;
- `limit` is analogous to `range()`, save that the lower end range is set to 0;
- `select` allows to specify the object returned by the traversal;
- `dedup` allows to remove duplicated objects for the traversal stream.

³¹ <http://tinkerpop.apache.org/docs/current/reference/>

The full list of all the operators provided by the Gremlin traversal language can be referred from <http://tinkerpop.apache.org/docs/current/reference/#graph-traversal-steps>. Next, we discuss the Gremlin traversal machine and the evaluation semantics of the Gremlin traversals.

Gremlin Traversal Machine

Theoretically, a set of traversers in T move (traverse) over a graph G (Property graph, cf. Figure 2.6) according to the instruction in $(\Psi$, i.e. the SST operations), and this computation is said to have completed when there are either:

1. no more existing traversers (t), or
2. no more existing instructions (ψ) that are referenced by the traversers (i.e. program has halted).

Result of the computation being either a null/empty set (i.e. former case) or the multiset union of the graph locations (vertices, edges, labels, properties, etc.) of the halted traversers which they reference. Rodriguez et al. [77] formally define the operation of a traverser t as follows:

$$G \leftarrow \mu \frac{t \in T}{\{\beta, \varsigma\}} \psi \rightarrow \Psi \quad (2.1)$$

where, $\mu: T \rightarrow U$ is a mapping from the traverser to its location in G ; $\psi: T \rightarrow \Psi$ maps a traverser to a step in Ψ ; $\beta: T \rightarrow \mathbb{N}$ maps a traverser to its bulk³²; $\varsigma: T \rightarrow U$ maps a traverser to its sack (local variable of a traverser) value.

Gremlin Semantics

As mentioned before a Gremlin traversal can be evaluated in either an imperative manner, a declarative manner, or a combination of both. An imperative Gremlin traversal tells the traversers how to proceed at each step in the traversal. In Gremlin the declarative construct (graph pattern matching), analogous to SPARQL [60, 63, 78], is provided by the `match()`-step³³. The `match()`-step evaluates each of the input graph patterns over a graph G in a structure preserving manner binding the variables and constants to their respective values [77]. However, the order of execution of each graph pattern is up to the `match()`-step implementation, where the variables and path labels are local only to the current `match()`-step. Due to this uniqueness of the Gremlin `match()`-step it is possible to:

1. treat each graph pattern individually as a single step traversal and thus, construct composite graph patterns by joining (path-joins) each of these single step traversals;
2. combine multiple `match()`-steps for constructing complex navigational traversals (i.e. multi-hop queries), where each composite graph pattern (from a particular `match()`-step)

³² The bulk of a traverser is number of equivalent traversers a particular traverser represents.

³³ <http://tinkerpop.apache.org/docs/3.2.3/reference/#match-step>

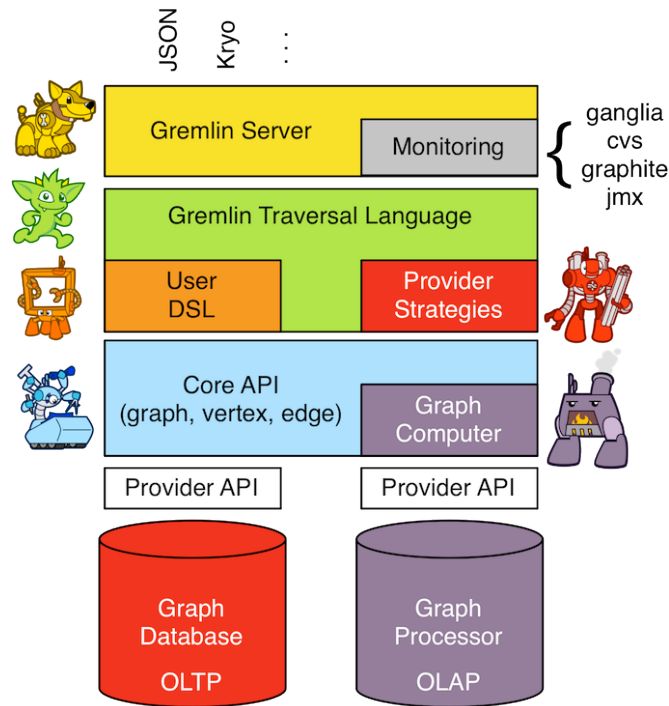


Figure 2.7: The technology stack of the Apache TinkerPop graph computing framework.

can be combined using the join operator.

The evaluation of an input graph pattern/traversal in Gremlin is taken care by two functions:

1. the recursively defined `match()` function- which evaluates each constituting graph pattern and keeps a track of the traversers location in the graph (i.e. path history), and,
2. the `bind()` function- which maps the declared variables (elements and keys) to their respective values.

Example 2.2.2. We illustrate the evaluation of the Gremlin traversals shown in listings 2.3 and 2.4, as (G1) and (G2), over the graph G from Figure 2.6.

(G1) ==>"Tesla, Incorporation" (G2) ==>[y="Tesla, Incorporation"]

2.2.3 Apache TinkerPop Graph Computing Framework

The TinkerPop framework is a part of the top level project at the Apache foundation – the Apache TinkerPop Project [18]. TinkerPop is a graph computing framework for both graph databases (OLTP) and graph analytic systems (graph processors – OLAP). Analogous to a computer which operates using data and a set of rules (algorithm), the TinkerPop graph computing framework consists of two key elements – the *data* i.e. the graph, and the *program* i.e. the traversal. The graph is a set of nodes, edges and properties (cf. Section 2.2.1), and the traversal is a Gremlin

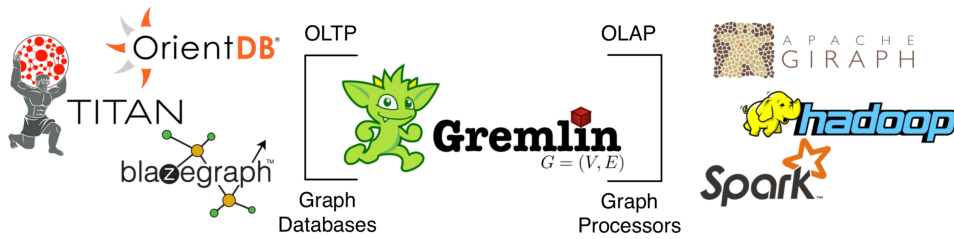


Figure 2.8: The TinkerPop-enabled graph database and graph processor providers.

traversal (cf. Section 2.2.2) which is a functional, data-flow based domain specific language (DSL) for the storing and manipulating the graph data. Together, the TinkerPop framework and Gremlin form a traversal *machine* (execution platform) and a traversal (programming) *language* analogous to Scala³⁴ and Java Virtual Machine³⁵ (JVM). Using Gremlin and the TinkerPop framework it is possible to construct another traversal language and execute it over any TinkerPop-enabled graph system creating a virtual layer over the physical system. This is similar to how a java program can be executed over any other JVM-enabled operating system without the requirement of recompilation or rewriting. Figure 2.7 presents the technology stack of the Apache TinkerPop graph computing framework³⁶.

TinkerPop-enabled Graph Systems

The popularity of the TinkerPop framework has attracted various graph database (OLTP) and graph analytical system (OLAP) vendors to adopt it as depicted in Figure 2.8. In the past few years the list of TinkerPop-enabled vendors has grown drastically which cover the key software industry players such as Amazon, Microsoft, IBM, Neo4j, etc. We list a few of the notable vendors below:

- Alibaba Graph Database [79] - A real-time, reliable, cloud-native graph database service that supports property graph model.
- Amazon Neptune [80] - Fully-managed graph database service.
- ArangoDB [81] - OLTP Provider for ArangoDB.
- Blazegraph [82] - RDF graph database with OLTP support.
- CosmosDB [83] - Microsoft's distributed OLTP graph database.
- ChronoGraph [84] - A versioned graph database.
- DSEGraph [85] - DataStax graph database with OLTP and OLAP support.
- GRAKN.AI [86] - Distributed OLTP/OLAP knowledge graph system.

³⁴ Scala Programming Language <https://www.scala-lang.org/>

³⁵ JVM https://en.wikipedia.org/wiki/Java_virtual_machine

³⁶ TinkerPop stack <http://tinkerpop.apache.org/docs/current/reference/>

- Hadoop (Spark) [87] - OLAP graph processor using Spark.
- Huawei Graph Engine Service [88] - Fully-managed, distributed, at-scale graph query and analysis service that provides a visualized interactive analytics platform.
- IBM Graph [89] - OLTP graph database as a service.
- JanusGraph [90] - Distributed OLTP and OLAP graph database with BerkeleyDB, Apache Cassandra and Apache HBase support.
- Neo4j [71] - OLTP graph database (embedded and high availability).
- OrientDB [91] - OLTP graph database
- Apache S2Graph [92] - OLTP graph database running on Apache HBase.
- Sqlg [93] - OLTP implementation on SQL databases.
- Stardog [94] - RDF graph database with OLTP and OLAP support.
- Titan [95] - Distributed OLTP and OLAP graph database with BerkeleyDB, Apache Cassandra and Apache HBase support.

However, a detailed list of TinkerPop-enabled graph systems can be accessed from <http://tinkerpop.apache.org/providers.html>.

TinkerPop-enabled Query Languages

One of the key benefits of using the TinkerPop-Gremlin technology stack is the leverage to its coverage of third-party query language providers. TinkerPop provides interfaces for accessing a variety of distinct and domain specific query languages such as the NoSQL query languages – SPARQL, Cypher, GraphQL, etc.; Relational query language – SQL. As Gremlin is a Turing-complete language any query written in any other query language can be compiled and executed over the Gremlin traversal machine. Below we present a list of query-interfaces or query interoperability bridges which are available to the TinkerPop users:

- Gremlin-Groovy³⁷ is used to represent Gremlin traversals inside the Groovy language and can be leveraged by any JVM-based project either through gmaven or its JSR-223 ScriptEngine implementation.
- Gremlin-Python³⁸ is used to represent Gremlin traversals inside the Python language and can be used by any Python virtual machine such as CPython and Jython.
- Gremlin-Java³⁹ is used to represent Gremlin traversals inside the Java8 language. Gremlin-Java is considered the canonical, reference implementation of Gremlin traversal language.

³⁷ Gremlin-Groovy <http://tinkerpop.apache.org/docs/current/reference/#gremlin-console>

³⁸ Gremlin-Python <http://tinkerpop.apache.org/docs/current/reference/#gremlin-python>

³⁹ Gremlin-Java http://tinkerpop.apache.org/docs/current/reference/#_on_gremlin_language_variants

- Gremlin-.NET⁴⁰ used to represent Gremlin traversals inside the C# language and can be used by any .NET-based VM.
- Gremlin-Scala⁴¹ is a scala based Gremlin variant for the TinkerPop framework.
- Gremlin-Ogre is a Gremlin language variant for Clojure. It provides an API that enhances the expressivity of Gremlin within Clojure.
- Gremlin-SQL⁴² is a SQL-to-Gremlin traversal compiler which compiles ANSI SQL to Gremlin traversals.
- Gremlin-SPARQL [41, 42] is a SPARQL-to-Gremlin traversal compiler. This is one of the main contributions of this doctoral thesis (discussed in Chapter 5 addressing the RQ2 regarding the *query interoperability* between RDF and Property graph databases).
- Gremlin-Cypher⁴³ is a Cypher-to-Gremlin traversal compiler.

The TinkerPop community is always accepting contributions from motivated developers and vendors. There is a public forum⁴⁴ that allows users, vendors and contributors to raise issues and discuss future development and integration of features and interfaces. It's wide popularity can be observed from the supported graph systems and query languages which are completely open-source and are available under the Apache 2.0 licence⁴⁵ for further development and commercial use. Next, we discuss about the role of both RDF and Property graph data models in the rise and development of Knowledge Graphs (KG). In doing so we present the different types of KGs, their large-scale adoption in the industry and a few use cases.

2.3 Knowledge Graphs

The term “Knowledge Graph” (KG) started gathering wide-spread attention after Google’s announcement on May 16, 2012 [96]. The research on KGs had however been active since the late 1980⁴⁶, when two dutch universities started a project called *Knowledge Graph* which was conceived as a semantic network with some additional restrictions in order to enable algebraic operations over graphs. While there is no formal definition of a KG, in general it is described as a semantic network of real world entities or entities of interest and the connections amongst them. Essentially KGs store factual information about real world entities in a semantically meaningful manner as a structured graph. In doing so a KG also describes the possible types of entities (classes) and the nature of their interactions (relations) as a schema. Apart from Google’s KG, there are various publicly available KGs such as DBpedia [52], Wikidata [53], YAGO [54], Freebase [97], etc. In [98], Heiko Paulheim characteristically defines a Knowledge

⁴⁰ Gremlin.NET <http://tinkerpop.apache.org/docs/current/reference/#gremlin-DotNet>

⁴¹ Gremlin-Scala <https://github.com/mpollmeier/gremlin-scala>

⁴² SQL to Gremlin compiler <https://github.com/twilmes/sql-gremlin>

⁴³ Cypher to Gremlin compiler <https://github.com/opencypher/cypher-for-gremlin>

⁴⁴ TinkerPop dev-mailing-list <https://lists.apache.org/list.html?dev@tinkerpop.apache.org>

⁴⁵ Apache 2.0 License <https://www.apache.org/licenses/LICENSE-2.0>

⁴⁶ source: <https://towardsdatascience.com/knowledge-graph-bb78055a7884>

Graph as follows:

Definition 1 (Knowledge Graph). A Knowledge Graph:

1. mainly describes real world entities and their interrelations, organised in a structured graph;
2. defines possible classes and relations of entities in a schema;
3. allows for potentially interrelating arbitrary entities with each other;
4. covers various topical domains.

For example DBpedia [52] describes the city of Bonn (<http://dbpedia.org/resource/Bonn>, we will use `dbr:Bonn` for short) as shown in the listing 2.5 below:

```
dbr:Bonn dbo:country dbr:Germany .
dbr:Bonn dbp:type "city"^^rdf:langString .
dbr:Bonn dbo:foundingYear "0001-01-01"^^xsd:gYear .
dbr:Bonn dbp:mayor "Ashok-Alexander Sridharan"^^rdf:langString .
dbr:Bonn dbo:populationTotal "311287"^^xsd:nonNegativeInteger .
dbr:Bonn dbo:postalCode "53111-53229" .
```

Listing 2.5: A snippet of the triples describing the city of Bonn (<http://dbpedia.org/resource/Bonn>)

KGs can be domain specific such as a KG of automotives, life sciences, sports, etc., or open domain such as an encyclopedia of information on diverse subjects. Large-scale KGs typical contain millions of entities and billions of facts [98]. A KG can be either completely based on RDF data such as DBpedia or hybrid that gathers data from heterogeneous sources or knowledge bases and formats based on the Property graph data model for instance. KGs have been adopted by a large fraction of the commercial companies and are of utmost importance in AI-based applications such as recommendation engines, questions answering, search engines and link prediction, digital/smart assistants, etc., to name a few. Figure 2.9, obtained from <https://www.slideshare.net/Frank.van.Harmelen/adoption-of-knowledge-graphs-late-2019>, presents the large-scale adoption landscape of Knowledge Graphs by a wide variety of companies for custom use cases.

KGs have emerged to be an extremely profitable means of technology for a variety of companies such as Google, Amazon, Facebook, Yahoo, LinkedIn, etc [99]. This is reflected in the market capital revenue figures as reported from a survey by PwC and Bloomberg⁴⁷, shown in Figure 2.10.

KGs can be constructed around a variety of use cases, a few types of KGs are – Enterprise KGs, Contextual KGs, Scholarly KGs, Personalised KGs, etc. We briefly explain each of these next:

- Enterprise Knowledge Graph (EKG) is typically built by an Enterprise around its product(s). This KG encapsulates the knowledge around the production and consumption of data or a particular product. Example of products are social networks, ecommerce, etc. Examples of EKGs are Facebook’s Social Graph⁴⁸, Bing Knowledge Graph⁴⁹, etc.

⁴⁷ source: <https://www.slideshare.net/AlanMorrison/scaling-the-mirrorworld-with-knowledge-graphs>

⁴⁸ https://en.wikipedia.org/wiki/Social_graph

⁴⁹ <https://blogs.bing.com/search-quality-insights/2017-07/bring-rich-knowledge-of-people-places>



Figure 2.9: The landscape of large-scale adoption of Knowledge Graphs in a variety of industrial domains as consolidated by Frank Van Harmelen in 2019.

- Contextualized Knowledge Graph (CKG) is a KG that contains contextual information in addition to the real world facts as metadata. The context dependent information can be for instance the provenance and trust metadata about the events and facts contained in the KG [100]. This information is extremely essential for answering contextual queries where inference of knowledge becomes critical [99]. Examples of such contextual KG applications are digital assistants (in conversational AI), consider a scenario where a user asks “On which day was the 2014 Football Worldcup played?” followed by the question “What was the weather on that day?”. In such a case the context of the conversation is key to answer the question correctly.
- Personalized Knowledge Graph (PKG) is based on data personalization of a specific user with respect to their particular product or device. The goal here is to enhance the user experience by cataloguing the likes and dislikes of the user and then presenting recommendations based on it. Examples of applications of PKGs are automotive industry, such as in Tesla cars wherein the smart assistant (application) remembers the air conditioner temperature setting and the choice of music of the user based on the past choice, and

	Company name	Location	Industry	Change in market cap 2009 – 2018 (\$bn)	Market cap 2018 (\$bn)
1	Apple	United States	Technology	757	851
2	Amazon.Com	United States	Consumer Services	670	701
3	Alphabet	United States	Technology	609	719
4	Microsoft Corp	United States	Technology	540	703
5	Tencent Holdings	China	Technology	483	496
6	Facebook	United States	Technology	383 ¹	464
7	Berkshire Hathaway	United States	Financial	358	492
8	Alibaba	China	Consumer Services	302 ¹	470
9	JPMorgan Chase	United States	Financials	275	375
10	Bank of America	United States	Financials	263	307

Figure 2.10: The market capital (revenue) generation by various companies leveraging Knowledge Graphs.

is set automatically without the person’s intervention. Another such domain is the pharmaceutical and life sciences industry, where there is increasing use of PKGs [101].

- Scholarly Knowledge Graph is based on the scholar and bibliographic networks. It is graph consisting of authors, their publications, their fields of study and their citation network. Examples of SKGs are the Microsoft Academic Knowledge Graph [102] and the Open Research Knowledge Graph [103].

Thus, it is clear that Knowledge Graphs are a key component for knowledge representation and consumption in a variety of domains and enterprises. While curating, storing and querying them is challenging, it is most certainly a very profitable technology going forward in the booming AI-based industry. Knowledge Graphs are enabling companies to increase their customer interaction by unlocking new opportunities.

2.4 Summary

In this chapter, we introduced the formal concepts and foundation technology stacks that are required in order to understand the work presented in this thesis. We now, in the next chapter 3, discuss the State-of-the-Art with respect to each of the research questions, and summarize their limitations on top of which we build or work, that are the core contributions of this dissertation.

Related Work

Having introduced the formal concepts required to understand the work and contributions presented in this dissertation (in chapters 1 and 2), we now we discuss the related work with respect to the three main research questions (RQ1, RQ2 and RQ3) that contribute to answering the overarching research objective of this dissertation:

Overarching Research Problem: How can we support interoperability between the Semantic Web and Property graph Databases?

In the pursuit of investigating the existing works, we cover a broad range of approaches addressing both *data interoperability* and *query interoperability* amongst Relational, RDF, Graph, Document-based, Hierarchical, etc. database models. We then narrow the focus specifically to those works that allow *data* and *query* interoperability between RDF and Graph databases in sections 3.1 and 3.2 respectively. Furthermore, we also discuss the State-of-the-Art in the database benchmarking domain in chapter 3.3. We also present a consolidated summary in the form of figures and tables, highlight the research gaps which need to be addressed and outline the contributions of this thesis. Finally, in section 3.4 we conclude the chapter with outlining the shortcomings in the State-of-the-Art for the respective proposals and present the step forward.

This chapter is based on the parts of related work from the following publications [39–41, 43, 45, 104–106]:

1. Renzo Angles, **Harsh Thakkar**, and Dominik Tomaszuk. *Mapping RDF Databases to Property Graphs*. In IEEE Access, Vol. 8, 2020.
2. **Harsh Thakkar**, Renzo Angles, Marko Rodriguez, Stephen Mallette, and Jens Lehmann. *Let's build Bridges, not Walls: SPARQL Querying of TinkerPop Graph Databases with Sparql-Gremlin*. In Proceedings of the IEEE 14th International Conference on Semantic Computing (ICSC), pp. 408-415, San Diego, USA, 2020. [**Best Paper Award**]
3. Renzo Angles, **Harsh Thakkar**, and Dominik Tomaszuk. *RDF and Property Graphs*

- Interoperability: Status and Issues*. In Proceedings of the 13th Alberto Mendelzon International Workshop on Foundations of Data Management, Asunción (AMW 2019), Paraguay, June 3-7, 2019.
4. Mohamed Nadjib Mami, Damien Graux, **Harsh Thakkar**, Simon Scerri, Sören Auer, and Jens Lehmann. *The Query Translation Landscape: A Survey*. Pre-print arXiv:1910.03118, pp. 1-25, 2019.
 5. **Harsh Thakkar**, Dharmen Punjani, Yashwant Keswani, Jens Lehmann, and Sören Auer. *A Stitch in Time Saves Nine – SPARQL Querying of Property Graphs using Gremlin Traversals*. Pre-print arXiv:1801.02911, pp. 1-24, 2018.
 6. **Harsh Thakkar**, Yashwant Keswani, Mohnish Dubey, Jens Lehmann, and Sören Auer. *Trying Not to Die Benchmarking – Orchestrating RDF and Graph Data Management Solution Benchmarks using LITMUS*. In Proceedings of the 13th International Conference on Semantic Systems (SEMANTiCS 2017), Amsterdam, Nederland, pages 120-127. ACM, 2017. [**Best Paper Award**]
 7. Saeedeh Shekarpour, Kemele M Endris, Ashwini Jaya Kumar, Denis Lukovnikov, Kuldeep Singh, **Harsh Thakkar**, and Christoph Lange. *Question Answering on Linked Data: Challenges and Future Directions*. In Companion Proceedings of the 25th International Conference Companion on World Wide Web (WWW), pages 693-698. 2016.

3.1 Data Interoperability between Databases

In this section we present the related work that targets the *syntactic* and *semantic* interoperability between the RDF and PG data models. We group the efforts based on mapping of the data model they target, i.e. RDF \rightarrow PG, etc., and summarise their shortcomings. Please note that \rightarrow implies unidirectional and \leftrightarrow implies a bidirectional transformation.

3.1.1 Syntactic Interoperability

The *Syntactic* interoperability is given by the facilities to transform data from one format to another. Therefore, the main requirement to support syntactic interoperability is the existence of data formats (i.e. a syntax for encoding data stored in a database), over which transformation methods can be implemented.

RDF Databases \leftrightarrow Relational Databases

Almost all the existing approaches for supporting *data interoperability* between RDF and Relational databases are *semantic* in nature. This is because the notion of a schema is native to relational databases and it is, for this reason, not possible to have a relational database without a schema. Therefore, all the approaches by default support transforming the relational schema

to an RDF Schema. Though, in some approaches the integrity constraints may or may not be preserved. Hence, we do not list any approaches that support purely a *syntactic* transformation.

RDF Databases ↔ Graph Databases

RDF → Property graphs: Turtle, TriG, RDF/XML, RDF/JSON and JSON-LD are data formats for encoding RDF data. In contrast, there is no standardized data format to encode Property graphs. Given this restriction, some systems use graph data formats (like GraphML, DotML, GEXF, GraphSON), but none of them is able to cover all the features presented by the Property graph data model.

Given a source data format S and a target data format T , the first option to support syntactic interoperability is to define a textual mapping from S to T . Note that the schema of the database is not considered in the transformation. Hence, the structure, semantics and restrictions of the source data could not be preserved by the translated data. The existing syntactic transformation approaches for converting RDF to Property graphs are as follows:

1. Hartig [107] proposes two transformations between RDF^{*} and Property graphs. RDF^{*} is a syntactic extension of RDF which is based on reification. The first transformation maps any RDF triple as an edge in the resulting Property graph. Each node has the “kind” attribute to describe the type of a node (e.g. IRI). The second transformation distinguishes data and object properties. The former are transformed into node properties, and latter into edges of a Property graph. The limitation of the second transformations is that metadata triples cannot be transformed. The shortcoming of this approach is that RDF^{*} isn’t supported by majority of RDF triplestores (except Blazegraph and the most recent addition, AnzoGraph) and requires conversion of existing RDF data beforehand.
2. Schätzle et al. [108] propose a mapping which is native to GraphX (a parallel processing system implemented on Apache Spark). The proposed graph model is an extension of the regular graph, but lacking the concept of attributes. The mapping uses an special attribute *label* to store the node and edge identifiers, i.e. each triple $t = (s, p, o)$ is represented using two vertices v_s, v_o , an edge (v_s, v_o) and properties $v_s.label = s, v_o.label = o, (v_s, v_o).label = p$. The proposed method does not address blank nodes or RDF entailments.
3. Nyugen et al. [109] proposed *LDM-3N* (labeled directed multigraph-three nodes), a graph model for RDF data. It is an extension of the regular graph, without the concept of attributes, and represents each triple element as separate nodes, thus three nodes (3N) . The *LDM-3N* graph model is used to address the Singleton Property (SP) based reified RDF data. The problem with this approach is that – (i) it adds an extra computation step (and $2n$ triples); and (ii) doesn’t cover RDF graph Schema; and misses the concept of properties.
4. Tomaszuk [110] presented an approach that uses the *YARS* serialization for transforming RDF data into Property graphs. This approach basically performs a transformation between encoding schemes and does not consider the RDF schema and its entailments.

This approach has several implementations, eg. neo4j-yars¹ and TTL2YARS².

Property graphs \rightarrow RDF: With respect to the methods to transform Property graphs into RDF graphs, the literature is very restricted. All the existing approaches only provide support for *syntactic* interoperability. The current methods RDF* [107] and Das et al. [4] are based on employing RDF reification. In the simplest case, for each edge in the Property graph there will be a blank node (in the RDF graph) containing at least three nodes (resources or literals) and three edges (properties). Such elements will be necessary to describe all the information of the original edge. Both approaches do not consider the presence of a PG schema.

In a recently published article Matsumoto et al. [111], present the Graph to Graph Mapping Language (*G2GML*) for mapping RDF graphs to Property graphs. This language can be processed by an implementation called G2G Mapper (available at <https://github.com/g2gml>). *G2GML* allows performing syntactic transformation of the RDF data and Graph serialisation such as (kryo, GraphML, etc.) However, there is no formal background or analysis of the proposed mappings and this approach does not support mapping RDF Schema and Blank nodes.

In the context of the scope of this thesis, we will focus on the State-of-the-Art that address the *data interoperability* between RDF and Graph data models. Table 3.1, presents a consolidated summary of related work that addresses the *data interoperability* issue between the RDF and Property graph data model along with the transformation features they offer.

Work	Target	Type	Direct Mapping	Formal Def.	Schema	Reif./ B.N.	I.P.
RDF* [107, 112]	RDF \leftrightarrow PG	syntactic	No	Yes	No	No/No	–
S2X [108]	RDF \rightarrow PG	syntactic	-	No	No	No/No	No
LDM-3N [109]	RDF \rightarrow PG	syntactic	No	No	No	No/No	No
G2GML [111]	RDF \leftrightarrow PG	syntactic	No	No	No	No/No	No
Das et al. [4]	PG \rightarrow RDF	syntactic	No	No	No	No/No	No
YARS [110]	PG \rightarrow RDF	syntactic	No	No	No	No/No	No

Table 3.1: A consolidated summary of related work supporting *data interoperability* between RDF and Property graphs. Here, B.N. refers to whether the approach supports Blank Nodes, Reif. refers to whether the approach supports RDF reification, I.P. refers to whether the approach is Information Preserving, and the “-” refers to the lack of evidence in the respective work. The type of the arrow in the column “**Target**” represents whether the proposed transformation is omni-directional or bi-directional.

3.1.2 Semantic Interoperability

Semantic interoperability between databases means that both, source and target systems, are able to understand the meaning of the data to be exchanged. It implies that both, data and schema must participate of the transformations method.

A common approach to support semantic interoperability is the definition of data and schema transformation methods. The schema transformation method takes as input the schema of

¹ <https://github.com/lszeremeta/neo4j-sparql-extension-yars>

² <https://github.com/lszeremeta/ttl-to-yars>

the source database, and generates a schema for the target database. Similarly, the data transformation method allows to move the data from the source database to the target database, but taking care of the target schema. The transformation methods can be implemented by using data formats or data definition languages.

RDF Databases ↔ Relational Databases

In this section, we discuss the related work addressing the *data interoperability* between RDF and Relational databases. We refrain from dwelling into much detail, as relational databases are not the focus of this thesis, instead list only selected works for the sake of completeness. We point the interested reader to studies such as [113–115], where a detailed report of the State-of-the-Art on this topic can be referred from.

RDF → RDB:

1. In [116] propose *RETRO*, wherein RDF data is exhaustively parsed to extract domain-specific relational schema. The schema corresponds to the so-called vertical partitioning, i.e., one table for every extracted predicate, each table is composed of <subject object> attributes.
2. In [117, 118] Ramanujan et al. propose *R2D*³, a novel approach to create a relational virtual normalized schema (view) on top of RDF data. Schema elements are extracted from RDF schema; if schema is missing or incomplete, schema information is extracted by thoroughly exploring the data. A relational view is created using those schema constructs. This generated schema provides support for executing SQL queries over RDF databases.
3. In *OntoAccess* Hert et al. [119], present a novel approach that allows vocabulary-based write access over relational data. It consists relational database to RDF mapping language called *R3M*, which consists of an RDF format and algorithms for translating queries to SQL.

RDB → RDF:

1. In [65, 120] Chebotko et al. propose two many-to-one mappings: (i) a mapping between the triples and the tables, and (ii) a mapping between pairs of the form (triple, pattern, position) and relational attributes. In addition this approach assumes that the underlying relational database is denormalized, and stores RDF terms. The two semantics deviate in the definition of the OPTIONAL algebra operator.
2. In *Triplify* Auer et al. [121] propose an approach to publish RDF from relational databases, which is based on mapping HTTP-URI requests to database queries expressed in SQL queries. These are used to match subsets of the stored contents and map them to classes

³ R2D source code <https://github.com/michaelbrunbauer/rdf2rdb>

and properties. This approach transforms the resulting relations into RDF statements and publishes the data in various RDF serializations.

3. In *D2RQ* Bizer et al. [122] propose a mapping representation which supports both automatic and manual operation modes for transforming RDF database to Relational database. In the automatic mode, RDF Schema vocabulary is created, in accordance with the reverse engineering method, for the translation of foreign keys to properties. In the manual mode, the contents of the database are exported to an RDF in accordance with mappings stored in RDF.
4. In *AuReLi* (Automatic Relational Database to Linked Data Converter) Polfiet et al. [123] propose a *D2RQ*-based approach which uses several string similarity measures associating attribute names to existing vocabulary entities in order to complete the automation of the transformation of databases. They also propose a low-cost method to automatically link the relevant data from other datasets.
5. In *R2RML* Das et al. [124] propose the popular W3C recommendation, which is a language for specifying mappings from relational to RDF data. In *R2RML* a mapping takes as input a *logical table*, which consists of – (i) a database table, (ii) a database view⁴, or a SQL query. Each *logical table* is mapped to a set of triples by rules called *triples map*. The *R2RML* produces by default, all the triples in the default graph of the output RDF triples dataset
6. In [125] Elliot et al. propose a custom mapping language that allows transforming relational data to RDF. The mapping language they propose is less expressive than *R2RML* [126] and lacks the support for URI templates.
7. In *SquirrelRDF* Seaborne et al. [127] present an approach that extracts data from a number of databases and integrates that data into a business process. Their proposal supports RDF views and allows for the execution of queries against it.
8. In *Ultrawrap* [67] Sequeda et al. propose the *RDF2RDB* mapping on top of existing relational databases. This approach creates an RDF ontology from the SQL schema, based on which it next creates a set of logical RDF views over the relational database. The views, called *Tripleviews*, are an extension of the famous *triple tables* (subject, predicate, object) with two additional columns: subject and object primary keys respectively. This mappings approach by this approach possess desirable properties such as *information preservation* and *soundness*.
9. The work by Priyatna [128] is an extension of the approach by Chebotko et al. [65, 120], wherein the additional feature is the provided support for user defined *R2RML* mappings. This is achieved by incorporating *R2RML* mappings in the α and β mappings as well as *genCondSQL()*, *genPRSQL()* and *trans()* functions.
10. In [126] Rodriguez-Muro et al. propose an *R2RML*-based mapping approach for transforming a relational database into RDF database. This is achieved by obtaining a virtual representation of the RDF graph via the *R2RML* mappings. Their approach uses a set of

⁴ It is called an “R2RML view” as it is like an SQL view without the capability to modify the database

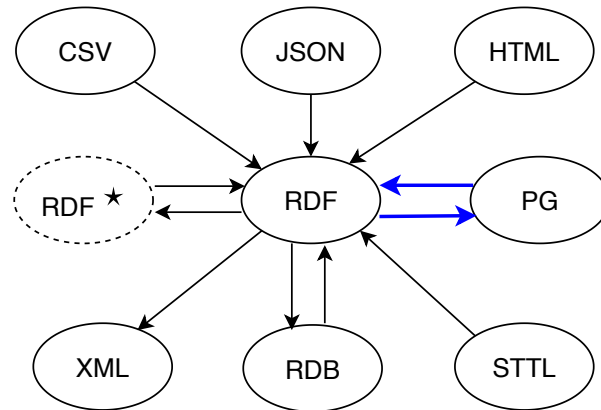


Figure 3.1: A graphical representation of the, existing efforts found in published literature, i.e. the State-of-the-Art addressing the *data interoperability* issue between RDF, Graph, Relational, and Heterogeneous data models respectively. The edges in black (\rightarrow) denote the existing works, whereas the edge in blue (\leftrightarrow) denotes the contribution of this thesis, supporting the transformation between RDF and Graph data model (covering both *syntactic* and *semantic* mappings). The node in dashed-line represents an intermediate approach mapping RDF to RDF*.

Datalog rules, as an intermediate language, in order to preserve the data semantics.

11. In *Ontop* [129] Calvanese et al. present a relational database to RDF database transformation approach which is based on the *R2RML* mappings over general relational schemas. This is achieved by exposing relational databases as virtual RDF graphs by linking the terms (classes and properties) in the ontology to the data sources through mappings.

RDF Databases \leftrightarrow Graph Databases

To the best of our knowledge, there is no existing method that supports data and schema transformations (*semantic* interoperability) between RDF and PGs. However, there exist some proposals for supporting *syntactic* interoperability (as discussed in the previous subsection) between RDF and Property graphs via the use of an intermediate data transformation language.

3.1.3 Other Approaches for Data Interoperability

There exist approaches that aim to support *data interoperability* amongst RDF, XML, JSON, CSV and other heterogeneous data formats at the same time. These approaches cannot be classified in one of the above mentioned categories (i.e. *syntactic* or *semantic*) as the notion and presence of a strict schema is not native to these heterogeneous data formats. In this subsection, we present a summary of a few selected approaches.

RDF \leftrightarrow Heterogeneous formats:

1. *RML* [130] is a generic language which allows to define mappings/rules that allow trans-

forming data from heterogeneous serialisation and structures to RDF. *RML* supports JSON, XML, CSV, and Relational databases. *RML* is defined as a superset of *R2RML*.

2. *SPARQL-Generate* [131] generates RDF from RDF dataset and a set of documents in arbitrary formats. It is designed as an extension of SPARQL 1.1. *SPARQL-Generate* supports XML, JSON, CSV, GeoJSON, HTML, and CBOR⁵.
3. In [132] Bischof et al. propose *XSPARQL* which is a query language based on SPARQL [7, 59] and XQuery [133, 134] for transformations from RDF into XML and back. It is built on top of XQuery in a syntactic and semantic view.
4. In [135] Battle et al. propose *Gloze* which is a tool for bi-directional mapping between XML and RDF. It uses information available in the XML Schema for describing how XML is mapped into RDF and back again.
5. In [136] Connolly et al. propose *GRDDL* is a markup language for for obtaining RDF data from XML and XHTML documents via associated transformation algorithms. These are typically represented in XSLT.
6. SPARQL Template Transformation Language (*STTL*) [137] are languages that allow data transformation between RDF and other languages or formats.

Figure 3.1 presents a graphical summary of the related works covering the State-of-the-Art that are concerned with both the *syntactic* and *semantic data interoperability* issue amongst RDF, Relational, Graph, and other heterogeneous data modelling and serialisation formats.

3.2 Query Interoperability between Databases

In this section we briefly survey the related work with regard to techniques and tools that address the *query interoperability* problem by supporting the translation and execution of query languages. We present the general related work with respect to the query language(s) supported by four database categories: Relational, Graph-based, Hierarchical and Document-based. Furthermore, we will focus *Query interoperability* between RDF and Property graph databases is a current issue due to the lack of a standard query language for Property graphs.

3.2.1 Graph-based ↔ Relational

A research topic closely related to our work is the *query interoperability* between SPARQL and SQL, which was investigated in e.g. [65, 120, 125, 126, 129]. Next, we categorise and list the efforts and proof of concept implementations that target the *query interoperability* gap between the RDF-Relation Database (RDB), and RDB-Property graph query languages respectively.

⁵ CBOR (Concise Binary Object Representation) is a binary representation of JSON.

SPARQL → SQL: There is a substantial amount of work been done for conversion of SPARQL queries to SQL queries [120, 125, 126, 128, 129, 138].

1. Calvanese et al. propose *Ontop* [129]⁶, which exposes relational databases as virtual RDF graphs by linking the terms (classes and properties) in the ontology to the data sources through mappings. This virtual RDF graph can then be queried using SPARQL by dynamically and transparently translating the SPARQL queries into SQL queries over the relational databases.
2. In [126] Rodriguez-Muro et al. present an approach that generates an optimized SQL from SPARQL by using Datalog as an intermediate language. This approach also provides a well-defined specification of the SPARQL semantics used in the translation. In addition, their approach, which is implemented within *Ontop* [129] also supports R2RML mappings over general relational schemas. The authors show, using the Ontology Based Data Access (OBDA), that their implementation can outperform other well known SPARQL-to-SQL systems, as well as commercial triple stores by large margin.
3. In [125], Elliot et al. introduce a SPARQL-to-SQL translation technique that focuses on the generation of efficient SQL queries. They proposes several translation *SQL model*-algorithms implementing different operators of a SPARQL query (algebra). In contrast to many existing works, this work aims to generate *flat/un-nested* SQL queries, instead of multi-level nested-queries, so SQL query optimizers can achieve better performance. However, their mapping language lacks support for URI templates and is less expressive than *R2RML*.
4. In [65, 120] Chebotko et al. proposes a translation function that takes a query and two many-to-one mappings: (i) a mapping between the triples and the tables, and (ii) a mapping between pairs of the form (triple, pattern, position) and relational attributes. A translation function returns a SQL query by fusing and building up the previous primitives given a graph pattern. The translation function generates SQL joins from UNIONs and OPTIONALs between sub-graph patters. The two semantics deviate in the definition of the OPTIONAL algebra operator.
5. In [128] Priyatna et al. propose an approach which is the extension of work by Chebotko et al. [120] to include user-defined *R2RML* mappings, which are incorporated within the α and β mappings as well as the *genCondSQL()*, *genPRSQL()* and *trans()* functions. For each, an algorithm is devised, considering the various situations found in R2RML mappings like the absence of Reference Object Map.
6. In [138] Zemeke et al. propose an approach that makes use of non-standard SQL constructs for SPARQL-to-SQL translation and lacks formal proof that the translation is correct and an empirical evaluation with realistic data is missing.
7. In *Ultrawrap* [67], Sequeda et al. propose an RDF2RDB mapping-based execution of SPARQL queries over relational databases. As mentioned earlier, this approach first transforms the data by creating an RDF ontology from the SQL schema, based on which

⁶ Ontop system (<http://ontop.inf.unibz.it/>)

it next creates a set of logical RDF views (called *Tripleviews*) over the relational database. Given a SPARQL query, each triple pattern maps to a *Tripleview*. The proposed approach has been shown to be information and query preserving respectively.

SQL → SPARQL: In [116] Rachapalli et al. present *RETRO*, a formal semantics preserving the translation from SQL to SPARQL. Their deals only with schema mapping and query mapping rather than to transform the data physically. Schema mapping derives a domain-specific relational schema from RDF data. Query mapping transforms an SQL query over the schema into an equivalent SPARQL query, which in turn is executed against the RDF store.

SQL → CYPHER: CYPHER⁷ is the graph query language used to query the *Neo4j*⁸ graph database. There has been no work yet aiming to convert the SQL to CYPHER. However, there are some examples⁹ that show the equivalent CYPHER queries for certain SQL queries.

CYPHER → SQL: *Cyp2sql* [139] is a tool for the automatic transformation of both data and queries from Neo4j to a relational database. During the transformation, the following tables are created: Nodes, Edges, Labels, Relationship types, plus materialized views to store the adjacency list of the nodes. CYPHER queries are then translated to SQL queries tailored to that data storage scheme.

SQL → Gremlin: There is initial work that shows Gremlin queries that are converted from SQL queries for the Northwind database. The proof-co-concept of this work can be found on a web-page[140]. However, it does not propose any algorithm or approach that can be used to convert SQL from any relational schema to Gremlin queries or traversals. It is limited to showing example SQL queries and corresponding Gremlin queries including the different traversal steps that are used for the mapping.

Gremlin → SQL: In [141], the authors propose a direct mapping approach for translating Gremlin queries (without the side effect step) to SQL queries. They discuss a generic technique to translate a subset of Gremlin queries (queries without side effect steps) into SQL leveraging the relational query optimizers. They propose techniques that make use of a novel schema which exploits both relational and non-relational storage for property graph data by combining relational storage with JSON storage for adjacency information and vertex and edge attributes respectively.

⁷ CYPHER Query Language (<https://neo4j.com/developer/cypher-query-language/>)

⁸ Neo4j (<https://neo4j.com/>)

⁹ SQL to CYPHER (<https://neo4j.com/developer/guide-sql-to-cypher/>)

3.2.2 Graph-based ↔ Document-based

The main motivation behind exploring this path was to enable SQL users and legacy systems to access the new class of NoSQL document databases with their sole SQL knowledge.

SPARQL → Document-based: The rationale here is identical to that of SPARQL-to-SQL, with one extra consideration: scalability. Native triple stores become prone to scalability issues when storing and querying significant amounts of RDF data. Users resorted to more scalable solutions to store and query the data [142]. The most studied database solution by the research community, we found, was MongoDB.

1. In *D-SPARQ* [143] Mutharaju et al. present an approach that focuses on the efficient processing of join operation between triple patterns of a SPARQL query. RDF data is physically materialized in a cluster of MongoDB stores, following a specific graph partitioning scheme. SPARQL queries are converted to MongoDB queries following the same.
2. In *SPARQL-to-X* [144] Michel et al. propose a generic two-step approach is suggested, with a showcase using MongoDB. The article proposes to convert a SPARQL query to a pivot intermediate query language called Abstract Query Language (AQL). The translation uses a set of mappings in xR2RML mapping language, which describe how data in target databases are mapped into RDF model, without converting data to RDF. AQL has a grammar that is similar to SQL both syntactically and semantically.

3.2.3 Graph-based ↔ Hierarchical

This bridge seeks to build interoperability environments between semantic and XML database systems, to enable ontology-based data access to XML data, and to add a semantic layer on top XML data and services for integration purposes.

SPARQL → XPath/XQuery: Enabling XPath traversal or XQuery functional programming styles on top of RDF data can be an interesting feature to equip native RDF stores with, in order to embark adopters from the XML world into the Semantic Web world. The following are a few approaches that address this feature.

1. In [145] Groppe et al. propose an approach that allows an indirect translation of SPARQL by embedding it inside a XQuery. This method involves firstly representing SPARQL in form of *tree of operators*. The translation involves data translation, from RDF to XML, and the translation of the operators to XQuery queries accordingly. The translation from an operator into an XQuery constructs is based on transformation rules, which replace the embedded SPARQL constructs with XQuery constructs.
2. In *SPARQL2XQuery* [146–148] Bikakis et al. propose a translation approach that is based on a mapping model between OWL ontology (existing or user-defined) and XML Schema.

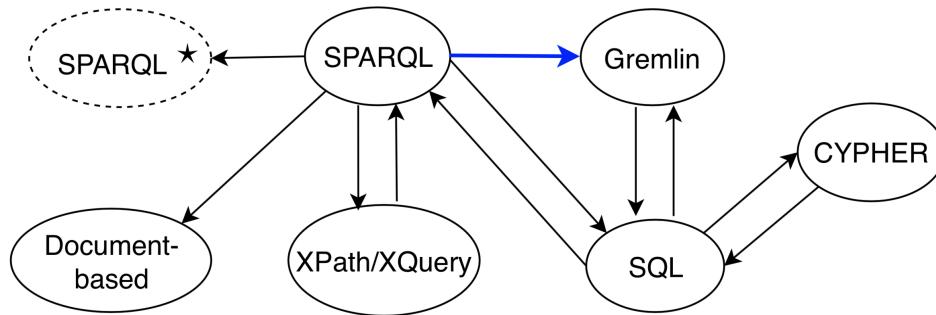


Figure 3.2: A graphical representation of the, existing efforts found in published literature, State-of-the-Art addressing the *query interoperability* issue between RDF, Graph, Relational, Hierarchical and Document-based databases respectively. The edges in black (\rightarrow) denote the existing works supporting the translation between the corresponding query languages (i.e. adjacent nodes). The edge in blue (\rightarrow) denotes the contribution of this thesis, supporting the translation of SPARQL queries to Gremlin traversals. The node in dashed-line represents an intermediate approach mapping SPARQL queries to SPARQL^{*} queries.

Mappings can either be automatically extracted by analyzing the ontology and XML schema, or manually curated by a domain expert. SPARQL queries are posed against the ontology without knowledge of the XML schema.

3. In *XQL2Xquery* [149] Fischer et al. propose an approach wherein the variables of the basic graph patter (BGP) are mapped to XQuery values. A *for* loop and a path expression is used to retrieve subjects and bind any variables encountered, then nested under every variable, iterate over the predicates and bind their variables.

XPath/XQuery \rightarrow SPARQL: In [150] Droop et al. present a translation approach that includes data transformation from XML to RDF. During the data transformation process, XML nodes are annotated with information used to support all XPath axes. For example, type information, attributes, namespaces, etc. The above annotations conform to the structure of the generated RDF and are used to generate the final SPARQL query.

3.2.4 Other Intermediate Graph-based Approaches

SPARQL \rightarrow SPARQL^{*}: In [112, 151] Hartig et al. have defined extensions of the SPARQL query language that capture an alternative approach to represent statement-level metadata that can be used in Property graphs (see [107]). This proposal, called SPARQL^{*} is an RDF^{*}-aware extension that introduces new features that enable users to directly access metadata triples in queries. However, this approach requires transforming the existing RDF data to the RDF^{*} data model, thereby adding an intermediate step, which is a conceptual extension of the RDF data model (cf. Section 3.1.2). Furthermore, it is not possible to query native Property graph databases using SPARQL^{*}. It is for this reason not a suitable candidate for directly addressing the *query interoperability* issue between the RDF and Property graph databases.

Figure 3.2 presents a graphical summary of the related works that are concerned with the *query*

interoperability issue between various database query languages. In the context of this thesis, we will focus on the State-of-the-Art that address the *query interoperability* between RDF and Graph databases.

3.2.5 Commercial Database Approaches

To the best of our knowledge, there is no formally published work or openly available software that addresses the *query interoperability* issue between SPARQL and any Property graph query language on a broader scale. On the other hand, Commercial graph databases, such as AWS Neptune, BlazeGraph, and Stardog do provide some form of *data interoperability* between the RDF and Property graph data model, however they are black-box systems as the technology is proprietary. On the other hand, they do not provide *query interoperability* between RDF and Property graph data models by translating SPARQL queries to any other Property graph query language, that is publicly known. Some of them (such as AWS Neptune and Azure Cosmos DB) provide support for querying using both Gremlin traversals and SPARQL directly, *without any form of translation to or from SPARQL*. It is for this reason, these commercial databases cannot be considered as related work. A similar argument holds for other TinkerPop-enabled Graph databases¹⁰.

¹⁰ TinkerPop-enabled Graph Databases <http://tinkerpop.apache.org/providers.html>

Work	DISTINCT /REDUCED	FILTER/ regex()	OPTIONAL	UNION	ORDER BY	LIMIT/ OFFSET	Blank nodes	datatype() /lang()	isURI() isLiteral()	DESCRIBE /bound()	CONSTRUCT /ASK	Others
<i>SPARQL → SQL</i>												
[152]	✓/	?	?	✓	✗	✓/	?	?	?	?	?	
[128]	?	?	?	?	?	?	?	?	?	?	?	
[125]	✓/	?	✓	✓	✓	✓/	✓	✓/	✓	✓/	✓/✓	GRAPH, FROM NAMED, isBlank()
[153]	?/	?	✓	?	✓	?	?	?	?	?	?	
[154]	?	✓	✓	✓	?	?	?	✓/	/✓	/✓	?	
[67]	✓/	✓/✓	✓	✓	✓	✓/✓	?	/✓	?	✓/✓	?	BIND
[66]	✓/	✓/	✓	✓	✓	✓/✓	?	/✓	?	?	?	
[65, 120]	?	?	✓	?	?	?	?	?	?	?	?	
<i>SQL → SPARQL</i>												
[116]	?	✓/	✓	✓		?	?	?	?	?	?	EXCEPT
[117, 118]	?	✓/✓	?	?	✓/	?	?	?	?	?	?	
<i>SPARQL → Document-based</i>												
[155]	?	✓/	?	?	?	?	?	?	?	?	?	
[156]	✓/	✓/✓	✓	✓	✓	✓/✓	?	/✓	?	✓/✓	✓/	
[143]	?	✗	✗	?	✗	?	?	?	?	?	?	
[157]	?	✓/	✗	?	✗	?	?	?	?	?	?	
<i>SPARQL → XPath/XQuery</i>												
[146–148]	✓/✓	✓/✓	✓	✓	✓	✓/✓	✓	?	?	✓/	✓/✓	DELETE, INSERT
[149]	✓/✓	✓/	✓	✓	✓	✓/✓	?	?	?	?	?	
[145]	?	✓/✓	✓	✓	✓	?	?	?	?	?	?	
<i>SQL ↔ Gremlin</i>												
[158]	✓	✓/✓	?	✓	✓/	✓	?	?	?	✓	?	
[140]		✓	✓	✓	✓	✓		✓/✓	✓	✓	?	WITH (recur.)

Table 3.2: A consolidated summary of the SPARQL language features supported in various SPARQL ↔ X query translation approaches. Here, ✓ refers to features that are supported, ✗ implies features that are not supported, and ? implies features that have not been (clearly) mentioned in the respective study. The *Others* column reports the features provided only by individual works.

3.3 Benchmarking Frameworks for RDF and Property graph Databases

Benchmarking is widely used for evaluating databases (i.e. Data Management Systems). Benchmarks exist for a variety of levels of abstraction from simple data models to graphs and triple stores, to entire enterprise information systems. We describe the current state of the art in benchmarking, in particular for: (a) Relational databases, (b) Graph databases, (c) RDF stores, and (d) Cross-domain (multi-modal) databases benchmarking efforts. We identify the scope and shortcomings of existing benchmarking efforts, to determine the gaps that need to be taken into consideration.

1. In *Relational* Data Management Systems, the benchmarks of the Transaction Processing Performance Council (**TPC**) [159] are well established. TPC uses discrete metrics for measuring the performance of the relational DMS. The online transaction processing benchmarks **TPC-C** and **TPC-E** use a transactions per minute metric. The analytics **TPC-H** and decision support **TPC-DS** benchmarks use the queries per hour and cost per performance metrics, respectively.
2. For *Graph* Data Management Systems, there exist benchmarks, some of which are in their early stages (such as **HPC Scalable Graph Analysis Benchmark** [160], **Graph 500** [161], **XGDBench** [162]) that deal with graph suitability transformations and graph analysis. However, they do not succeed to define standards for graph modelling and query languages.
3. Benchmarking *RDF* Data Management Systems. The substantial increase in the number of applications that use *RDF* data has encouraged the need for large-scale benchmarking efforts on all aspects of the Linked Data life cycle, mostly focusing on query processing [163]. RDF DMS benchmarks use real (i.e., DBpedia or Wikidata) and synthetic (i.e., Berlin SPARQL Benchmark or WAT-DIV) datasets to evaluate DMS performance over custom stress-loads and setup environments.¹¹ DBpedia SPARQL Benchmark (**DBPSB**) [164] assesses RDF Data Management Systems performance over DBpedia by creating a query workload derived from the DBpedia query logs. The aim of the Lehigh University Benchmark (**LUBM** [165]) is to evaluate the performance of Semantic Web triple stores over a large synthetic dataset that complies to a university domain ontology. The Waterloo SPARQL Diversity TEST Suite (**WatDiv** [166]) provides data and query generators to enable benchmarking of RDF Data Management Systems against a varying query structure (also complexity) to understand correlation of query typology with the variance in DMS performance. **SP2Bench** [167], one of the most commonly used synthetic data based benchmarks, uses the schema of the DBLP bibliographic dataset¹² to generate arbitrarily large datasets.
4. Benchmarking *Cross-domain* Data Management Systems. There are only a few efforts that benchmark cross-domain DMS so far. The Berlin SPARQL Benchmark (**BSBM** [168]) is a synthetic data benchmark, based on an e-commerce use cases built around a set of products offered by different vendors. It provides the dataset and queries for both RDF

¹¹ <https://www.w3.org/wiki/RdfStoreBenchmarking>

¹² <http://dblp.uni-trier.de/db/>

and Relational DMS benchmarking. **Pandora**¹³, uses the Berlin SPARQL Benchmark data to benchmark RDF stores against relational stores (Jena-TDB, MonetDB, GH-RDF-3X, PostgreSQL, 4Store). **Graphium** [169] is a similar study benchmarking RDF stores against Graph stores (Neo4J, Sparksee/DEX, HypergraphDB, RDF-3X) on graph datasets including a 10M triple graph data generated using the Berlin SPARQL Benchmark data generator. More recently, the **LDBC** [170] focused on combining industry-strength benchmarks for graph and RDF data management systems. The LDBC introduces a new choke-point analysis methodology for developing benchmark workloads, which tries to combine user input with feedback from system experts.

Table 3.3 (see next page) presents a consolidated summary of the existing approaches with respect to the State of the Art in benchmarking Relational, Graph, RDF, and other cross-domain Data Management Systems (databases).

¹³ <http://pandora.ldc.usb.ve/>

3.3 Benchmarking Frameworks for RDF and Property graph Databases

Type	Benchmark	RDB	RDF	Graph	Description
<i>Single-domain</i>	TPC [159]	✓			The Transaction Processing Performance Council (TPC) [159] is well established for benchmarking relational Data Management Systems. TPC provides a range of benchmarks such as the online transaction processing benchmarks TPC-C and TPC-E (which employ transactions per minute metric), the analytics TPC-H and decision support TPC-DS (which employ the queries per hour and cost per performance metrics).
	XGDBench [162]			✓	is a graph database benchmarking platform for cloud computing systems, which is an extension of the famous Yahoo! Cloud Serving Benchmark. The authors benchmarked AllegroGraph, Fuseki, Neo4j, and OrientDB using XGDBench on Tsubame 2.0 HPC cloud environment .
	HPC [160]			✓	The HPC Scalable Graph Analysis Benchmark consists of a range of tests for examining a variety of independent attributes of the hardware of High Performance Computing systems. HPC addresses graph-specific tasks such as graph suitability transformations and graph analysis over graph Data Management Systems in a distributed environment.
	Graph500 [161]			✓	is a benchmark for data intensive supercomputing systems and its applications. It does not consider benchmarking typical graph databases.
	DBPSB [164]		✓		The DBpedia SPARQL Benchmark (DBPSB) benchmarks RDF Data Management Systems using DBpedia by creating a query workload derived from the DBpedia query logs.
	LUBM [165]		✓		The Lehigh University Benchmark (LUBM) benchmarks RDF Data Management Systems over a large synthetic dataset that complies to a university domain ontology.
	WatDiv [166]		✓		The Waterloo SPARQL Diversity TEST Suite (WatDiv) benchmarks RDF Data Management Systems using their synthetic data and query generators in order to analyze the correlation between DMS performance against varying query structures and complexities (query typology).
	SP2Bench [167]		✓		is one of the most commonly used synthetic data-based RDF DMS benchmarks, which uses the schema of the DBLP bibliographic dataset (http://dblp.uni-trier.de/db/) to generate custom sized datasets.
	IGUANA [171]		✓		is a generic SPARQL benchmark execution framework focused on benchmarking RDF Data Management Systems and federated querying.
	FEASIBLE [172]		✓		is a feature-based (data-driven and structural) SPARQL benchmarking framework for RDF Data Management Systems. It employs an automatic approach for the generation of benchmarks using query logs.
	LSQ [173]		✓		consists of real world SPARQL queries extracted from the logs of public SPARQL endpoints. These queries are extracted from four public endpoints: DBpedia (logs 232 million triples), Linked GeoData (LGD) (1 billion triples), Semantic Web Dog Food (SWDF) (300 thousand triples) and the British Museum (BM) 1.4 million triples).
HOBBIT [163]		✓		is an end-to-end benchmarking platform (in early stage) focused towards large-scale benchmarking on all aspects of the Linked Data life cycle. It will enable data, query and task generation functionalities (in later stages) for benchmarking of RDF Data Management Systems under custom stress loads for the querying of RDF graphs using industrial use-case queries.	
<i>Cross-domain</i>	BSBM [168]	✓	✓		The Berlin SPARQL Benchmark (BSBM) is a synthetic data-based e-commerce use case scenario for benchmarking RDF and Relational Data Management Systems. It provides custom generators for creating datasets and queries of custom size and typology.
	Pandora	✓	✓		Pandora (http://pandora.ldc.usb.ve/) is a benchmark which uses the BSBM data to assess the performance of RDF stores against relational stores (i.e. Jena-TDB, MonetDB, GH-RDF-3X, PostgreSQL, 4Store).
	Quertzal-RDF [174]	✓	✓		is a RDF and Graph DMS benchmarking framework, which offers a novel SPARQL to SQL translation engine for multiple backends. Its current version supports benchmarking DB2, PostgreSQL and Apache Spark. It offers custom query loads for both DBpedia (real) and LUBM (synthetic) datasets.
	Graphium [169]		✓	✓	Is a benchmarking plus result visualization effort, comparing RDF stores against Graph stores (i.e. Neo4J, Sparksee/DEX, HypergraphDB, RDF-3X) on custom graph datasets including a 10M triple dataset (using the BSBM data generator).
	LDBC [170]		✓	✓	The Linked Data Benchmark Council (LDBC) is focused on curating industry-strength benchmarks for both graph and RDF Data Management Systems. It introduces a choke-point driven analysis methodology for analyzing and developing benchmark workloads.

Table 3.3: A consolidated summary of the State-of-the-Art in benchmarking frameworks for Relational, RDF and Graph Data Management Systems.

3.4 Summary

In this chapter, we presented the existing approaches in the respective State-of-the-Art for – *data interoperability*, *query interoperability* between RDF and Graph databases, and the frameworks for benchmarking them. Next, we summarize their shortcomings, highlight the research gaps with respect to each of these and outline the challenges that may be faced in overcoming them.

- **Data Interoperability between RDF and Graph databases:** It is evident from the State-of-the-Art for supporting *data interoperability* between various data models, that there is a severe lack of efforts targeting the RDF and Property graph data models specifically. While the existing RDF \leftrightarrow Property graph transformation approaches focus merely on enabling syntactic interoperability, some of them also do not provide a direct mapping (as shown in Table 3.1). Furthermore, the existing approaches also do not support transforming RDF Schema, Blank nodes, RDF reification and its entailments (inferencing) into a Property graph. Thus, there is a need for an approach that allows both syntactic and semantic *query interoperability* between RDF and Property graph data models covering RDF Schema, Blank nodes, RDF reification and its entailments.
- **Query Interoperability between RDF and Graph databases:** A majority of the existing works in the State-of-the-Art support addressing the *query interoperability* are focused on SQL \leftrightarrow SPARQL translation (as shown in Table 3.2). Both SPARQL and SQL are declarative query languages and operate over the same relational algebra. Gremlin is a functional property graph query language which offers both declarative and imperative constructs. Furthermore, the Property graph domain lacks a standardized query language, unlike the RDF and RDB data models and Gremlin has not been thoroughly studied with respect to its execution semantics and expressivity. Therefore, in contrast to these existing *query interoperability* translation efforts, we have to overcome the challenge of mediating between two very different execution paradigms. More specifically, those efforts applied query rewriting techniques between languages, which are rooted in relational algebra operations, whereas we had to bridge more disparate query paradigms. While this poses a significant challenge, it is also the reason why substantial performance differences can be observed depending on the different query characteristics.
- **Benchmarking RDF and Graph databases:** The existing benchmarks, have various domain-specific strengths. However, they also display limitations regarding the need of having an integrated generalized benchmarking framework. The existing efforts, for instance, (i) do not offer the capability of benchmarking both RDF and Property graphs in a single environment; (ii) do not offer a complete automation or a common framework for reproducing existing benchmarks; (iii) with the exception of HOBBIT, do not offer an end-to-end benchmarking and result visualization solution of cross domain Data Management Systems; and (iv) do not allow easy integration of existing benchmarks in an user-driven fashion. Therefore, it is necessary to develop an automated open, extensible and reusable benchmarking framework that address all the above mentioned limitations for RDF and Property graph Data Management Systems (databases).

Directly Mapping RDF Databases to Property graph Databases

RDF [6] and Graph databases [69] are two approaches for data management that are based on modeling, storing and querying graph-like data. The database systems based on these models are gaining relevance in the industry due to their use in various application domains where complex data analytics is required [175]. Both, RDF and Graph database systems are tightly connected as they are based on graph-oriented database models. RDF databases are based on the RDF data model [6], their standard query language is SPARQL [7], and RDF Schema [8] allows to describe classes of resources and properties (i.e. the data schema). On the other hand, most graph databases are based on the Property Graph (PG) data model, there is no standard query language, and there is no standard notion of property graph schema [14]. Therefore, RDF and PG databases are dissimilar in the data model, schema, query language, meaning, and content. Given the heterogeneity between RDF triplestores and Property graph databases, and considering their graph-based data models, it becomes fair and necessary to develop methods to allow interoperability among these systems. To the best of our knowledge, the research about the interoperability between RDF and PG databases is very restricted (as discussed in Section 3.1 of Chapter 3). While there exist some system-specific (i.e. indirect) approaches, most of them are oriented to data transformation (focusing mostly on syntactic interoperability), lack solid formal foundations and even overall compatibility in some cases.

In this chapter we address the first research question (RQ1) that is concerned with addressing *data interoperability* issue between the RDF and Property graph databases.

RQ1: Data Interoperability – How can we directly map RDF Databases to Property Graph Databases in an information preserving manner?

The main contributions of this chapter are the following: (a) we define three database mappings [106]: (i) a simple mapping which allows transforming an RDF graph into a PG without considering schema restrictions (in both sides); (ii) a generic mapping which allows transforming an RDF graph (without RDF schema) into a PG that follows the restrictions defined by a generic PG schema; (iii) a complete mapping which allows transforming a complete RDF

database into a complete PG database (i.e. schema and instance); and (b) We also study three desirable properties of the above database mappings: computability, semantics preservation, and information preservation. Based on such analysis and our comprehensive evaluations by implementing the mappings in RDF2PG, we formally prove that two of the proposed mappings are in fact satisfying these three properties and argue that it is indeed possible to transform any RDF database into a PG database. In terms of data modeling, we can conclude that the PG data model subsumes the information capacity of the RDF data model.

This chapter is based on the following publications [38, 39, 106]:

1. Renzo Angles, **Harsh Thakkar**, and Dominik Tomaszuk. *Mapping RDF Databases to Property Graphs*. In IEEE Access, Volume 8, 2020.
2. Dominik Tomaszuk, Renzo Angles, and **Harsh Thakkar**. *PGO: Describing Property Graphs in RDF*. In IEEE Access, Vol. 8, 2020.
3. Renzo Angles, **Harsh Thakkar**, and Dominik Tomaszuk. *RDF and Property Graphs Interoperability: Status and Issues*. In Proceedings of the 13th Alberto Mendelzon International Workshop on Foundations of Data Management, Asunción (AMW 2019), Paraguay, June 3-7, 2019.

The remainder of this chapter is structured into five sections, starting with Section 4.1, we provide formal graph-based definitions of the notions of RDF graph, RDF graph schema, Valid RDF graph; Section 4.2, provides formal definitions of the notions of a Property Graph, Property Graph Schema, and Valid Property Graph; Section 4.3, presents and discusses the novel direct RDF to Property graph mappings (and their properties) addressing both schema-dependent and schema-independent data with concrete illustrations; Section 4.4, elaborates on our implementation, experimental setup, methodology of the evaluation. It also presents the findings from the results of the conducted experiments demonstrating the validity and scalability of the proposed mappings and also its limitations; Finally, Section 4.5 concludes the chapter outlining future work.

4.1 RDF Database (as an edge-labeled graph)

In this section, we formally introduce the main elements concerning RDF databases. Specifically, we define the concepts of RDF graph, RDF graph Schema, and introduce the notion of a valid RDF graph.

The Resource Description Framework (RDF) is a well-known W3C standard, which is used for data modeling and encoding machine readable content on the Web [6] and within intranets. An RDF graph can be seen as a set of triples, roughly analogous to nodes and edges in a graph database. However, RDF is more specific in defining disjoint vertex-sets of Blank nodes, literals and IRIs. In the rudimentary form, an RDF graph is a directed, edge-labeled, multigraph or simply an edge-labeled graph.

Assume that I , B and L are three disjoint infinite sets, corresponding to IRIs, blank nodes

and literals respectively. An IRI identifies a concrete web resource, a blank node identifies an anonymous resource, and a literal is a basic value (e.g. a string, a number or a date). We will use the term RDF resource to indicate any element in the set $I \cup B$.

4.1.1 RDF Graph

Informally, an RDF graph is a set of RDF triples. An RDF triple is a tuple $t = (v_1, v_2, v_3)$ where $v_1 \in I \cup B$ is called the *subject*, $v_2 \in I$ is called the *predicate* and $v_3 \in I \cup B \cup L$ is called the *object*. Here, the subject represents a resource, the predicate represents a relationship of the resource, and the object represents the value of such relationship. Given a set of RDF triples S , we will use $\text{sub}(S)$, $\text{pred}(S)$ and $\text{obj}(S)$ to denote the sets of subjects, predicates, and objects in S respectively.

There are different data formats to encode a set of RDF triples, including Notation3 (N3) [176], RDF/XML [177], N-Triples [178], Turtle [179] and N-Quads [180]. The following example shows a set of RDF triples encoded using the Turtle data format.

Example 4.1.1. Consider a subset of RDF triples from DBpedia related to Elon Musk represented in the turtle [56] format.

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix voc: <http://www.example.org/voc/> .
@prefix ex: <http://www.example.org/data/> .

ex:Tesla_Inc rdf:type voc:Organisation .
ex:Tesla_Inc voc:name "Tesla, Inc." .
ex:Tesla_Inc voc:creation "2003-07-01"^^xsd:date .
ex:Tesla_Inc voc:ceo ex:Elon_Musk .
ex:Tesla_Inc voc:location _:b1 .
ex:Elon_Musk rdf:type voc:Person .
ex:Elon_Musk voc:birthName "Elon Musk" .
ex:Elon_Musk voc:age "46"^^xsd:int .
_:b1 rdf:type voc:City .
_:b1 voc:name "Palo Alto" .
_:b1 voc:country _:b2 .
_:b2 rdf:type voc:Country .
_:b2 voc:name "US" .
_:b2 voc:is_location_of ex:Tesla_Inc .
```

The lines beginning with `@prefix` are prefix definitions and the rest are RDF triples. A prefix definition associates a prefix (e.g. `voc`) with an IRI (e.g. `http://www.example.org/voc/`). Hence, a full IRI like `http://www.example.org/voc/Person` can be abbreviated as a prefixed name `voc:Person`. We will use `prefix(r)` and `name(r)` to extract the prefix and the name of an IRI r respectively.

In order to facilitate readability, we will use prefixed names instead of full URIs. Moreover, we will assume that there exists a standard way to transform a full URI into a prefixed name, and vice versa (e.g. by using an internal index or an external service like DRPD [181]). A blank node is usually represented as `_:` followed by a blank node label which is a series of name characters (e.g. `_:b1`). There are other ways to encode blank nodes (e.g. `[]`), but we will use the above for simplicity. Given a blank node b , the function $\text{lab}(b)$ returns the label of b .

We will consider two types of literals: a simple literal which is a Unicode string (e.g. "Elon Musk"), and a typed literal which consists of a string and a datatype IRI (e.g. "46"^^`xsd:int`). Numbers can be unquoted and boolean values may be written as either `true` or `false`. Given a literal l , the function $\text{val}(l)$ returns the string of l .

The example shows the six types of valid RDF triples: (iri, iri, iri) in line 5, $(iri, iri, literal)$ in line 6, $(iri, iri, bnode)$ in line 9, $(bnode, iri, iri)$ in line 13, $(bnode, iri, literal)$ in line 14, and $(bnode, iri, bnode)$ in line 15. Any other combination is considered invalid.

A set of RDF triples can be visualized as a graph where the nodes represent the resources, and the edges represent properties and values. However, the RDF model has a particular feature: an IRI can be used as an object and predicate in an RDF graph. For instance, the triple $(\text{voc:ceo}, \text{rdfs:label}, \text{"Chief Executive Officer"})$ can be added to the graph shown in Example 4.1.1 to include metadata about the property `voc:ceo`. It implies that an RDF graph is not a traditional graph because it allows edges between edges, and consequently an RDF graph cannot be visualized in a traditional way. Next, we introduce a formal definition of the RDF data model which is able to support the above feature. Next, we formally define the notion of an RDF graph.

Definition 2 (RDF Graph). An RDF graph is defined as a tuple $G^R = (N_R, N_L, E_O, E_D, \alpha_R, \alpha_L, \beta_O, \beta_D, \delta)$ where:

- N_R is a finite set of nodes representing RDF resources (i.e. resource nodes divided in IRI nodes and blank nodes);
- N_L is a finite set of nodes representing RDF literals (i.e. literal nodes), satisfying that $N_R \cap N_L = \emptyset$;
- E_O is a finite set of edges called object property edges;
- E_D is a finite set of edges called datatype property edges¹, satisfying that $E_O \cap E_D = \emptyset$;
- $\alpha_R : N_R \rightarrow I \cup B$ is a total one-to-one function that associates each resource node with a resource identifier (i.e. either a IRI or a blank node identifier);
- $\alpha_L : N_L \rightarrow L$ is a total one-to-one function that associates each literal node with a single literal;
- $\beta_O : E_O \rightarrow (N_R \times N_R)$ is a total function that associates each object property edge with a

¹ The terms “object property” and “datatype property” have been taken from the Web Ontology Language (OWL) [182]

pair of resource nodes;

- $\beta_D : E_D \rightarrow (N_R \times N_L)$ is a total function that associates each datatype property edge with a resource node and a literal node;
- $\delta : (N_R \cup N_L \cup E_O \cup E_D) \rightarrow I$ is a partial function that assigns a resource class label to each node or edge.

Note that the function δ has been defined as being partial in order to support a partial connection between schema and data (which is usual in real RDF datasets). However, it is possible to define the following simple procedure to make the function δ total: For each resource $r \in N_R$, if $r \notin \text{dom}(\delta)$ then assign $\delta(r) = \text{rdfs:Resource}$. Therefore, we will assume that every resource in an RDF graph defines its resource class.

Concerning the issue about an IRI u occurring as both resource and property, note that u will occur as resource and property separately. In such a case, we will have a bipartite graph. The same applies for blank nodes.

Given a set of RDF triples S , the procedure to create a formal RDF graph $G^R = (N_R, N_L, E_O, E_D, \alpha_R, \alpha_L, \beta_O, \beta_D, \delta)$ from S is defined as follow:

- For every resource $r \in \text{sub}(S)$, there is a node $n \in N_R$ with $\alpha_R(n) = r$;
 - If $(r, \text{rdf:type}, c) \in S$ then $\delta(n) = c$, else $\delta(n) = \text{rdfs:Resource}$;
- For every literal $l \in \text{obj}(S) \cap L$, there is a node $n \in N_L$;
 - If l is a simple literal then $\alpha_L(n) = l$ and $\delta(n) = \text{xsd:string}$;
 - If l is a typed literal of the form $\text{value}^{\text{datatype}}$ then $\alpha_L(n) = \text{value}$ and $\delta(n) = \text{datatype}$;
- For every triple $(s, p, o) \in S$ where $o \in I \cup B$, there is an edge $e \in E_O$ with $\delta(e) = p$ and $\beta_O(e) = (n, n')$, such that $\alpha_R(n) = s$ and $\alpha_R(n') = o$;
- For every triple $(s, p, o) \in S$ where $o \in L$, there is an edge $e \in E_D$ with $\delta(e) = p$ and $\beta_D(e) = (n, n')$, such that $\alpha_R(n) = s$ and $\alpha_L(n') = o$.

The RDF triples shown in example 4.1.1 can be graphically represented as an edge-labeled graph, referred to as an RDF graph, is shown in Figure 4.1. We can represent the sample RDF from Figure 4.1, using the formal definition mentioned above as follows:

- 1 $N_R = \{n_1, n_2, n_3, n_4\}$,
- 2 $N_L = \{n_5, n_6, n_7, n_8, n_9, n_{10}\}$,
- 3 $E_O = \{e_1, e_2, e_3, e_4\}$,
- 4 $E_D = \{e_5, e_6, e_7, e_8, e_9, e_{10}\}$,
- 5 $\alpha_R(n_1) = \text{ex:Tesla_Inc}$, $\alpha_R(n_2) = \text{ex:Elon_Musk}$, $\alpha_R(n_3) = _:\text{b1}$, $\alpha_R(n_4) = _:\text{b2}$,
- 6 $\alpha_L(n_5) = \text{"Tesla, Inc."}$, $\alpha_L(n_6) = \text{"2003-07-01"}$, $\alpha_L(n_7) = \text{"Elon Musk"}$, $\alpha_L(n_8) = \text{"46"}$,
 $\alpha_L(n_9) = \text{"Palo Alto"}$, $\alpha_L(n_{10}) = \text{"US"}$,
- 7 $\beta_O(e_1) = (n_1, n_2)$, $\beta_O(e_2) = (n_1, n_3)$, $\beta_O(e_3) = (n_3, n_4)$, $\beta_O(e_4) = (n_4, n_1)$,

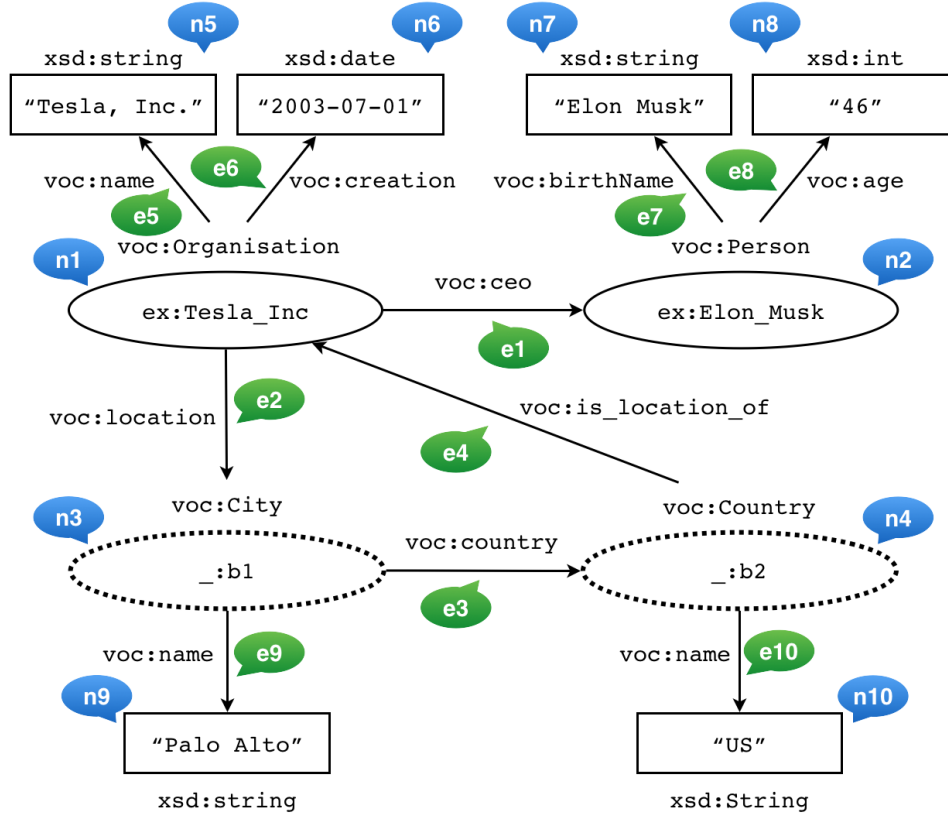


Figure 4.1: A graphical illustration of an RDF graph describing information about Elon Musk and Tesla Incorporation. We use abbreviated IRIs throughout this chapter.

- 8 $\beta_D(e_5) = (n_1, n_5)$, $\beta_D(e_6) = (n_1, n_6)$, $\beta_D(e_7) = (n_2, n_7)$, $\beta_D(e_8) = (n_2, n_8)$,
- 9 $\beta_D(e_9) = (n_3, n_9)$, $\beta_D(e_{10}) = (n_4, n_{10})$,
- 10 $\delta(n_1) = \text{voc:Organisation}$, $\delta(n_2) = \text{voc:Person}$,
- 11 $\delta(n_3) = \text{voc:City}$, $\delta(n_4) = \text{voc:Country}$
- 12 $\delta(n_5) = \text{xsd:string}$, $\delta(n_6) = \text{xsd:date}$,
- 13 $\delta(n_7) = \text{xsd:string}$, $\delta(n_8) = \text{xsd:int}$,
- 14 $\delta(n_9) = \text{xsd:string}$, $\delta(n_{10}) = \text{xsd:string}$,
- 15 $\delta(e_1) = \text{voc:ceo}$, $\delta(e_2) = \text{voc:location}$, $\delta(e_3) = \text{voc:country}$, $\delta(e_4) = \text{voc:is_location_of}$,
- 16 $\delta(e_5) = \text{voc:name}$, $\delta(e_6) = \text{voc:creation}$, $\delta(e_7) = \text{voc:birthName}$, $\delta(e_8) = \text{voc:age}$,
- 17 $\delta(e_9) = \text{voc:name}$, $\delta(e_{10}) = \text{voc:name}$.

Additionally, Figure 4.1 shows a graphical representation of the RDF graph described above. The IRI nodes are represented as ellipses, the blank nodes are represented as dotted ellipses and literal nodes are presented as rectangles. Each node is labeled with two IRIs: the inner IRI indicates the resource identifier, and the outer IRI indicates the resource class of the node. Each edge is labeled with an IRI that indicates its property class. We use balloons to indicate the object identifiers.

4.1.2 RDF Graph Schema

RDF Schema (RDFS) [8] defines a standard vocabulary (i.e., a set of terms, each having a well-defined meaning) which enables the description of resource classes and property classes. From a database perspective, RDF Schema allows to define the structure of the data in an RDF graph, i.e. a schema for RDF data.

In order to describe classes of resources and properties, the RDF Schema vocabulary defines the following terms: `rdfs:Class` and `rdf:Property` represent the classes of resources, and properties respectively; `rdf:type` can be used (as property) to state that a resource is an instance of a class; `rdfs:domain` and `rdfs:range` allow to define domain resource classes and range domain classes for a property, respectively. Note that `rdf:` and `rdfs:` are the prefixes for RDF and RDFS respectively.

An RDF Schema description consists into a set of RDF triples, so it can be encoded using RDF data formats. The following example shows an RDF Schema document which describes the structure of the data shown in Example 4.1.1, using the Turtle data format.

Example 4.1.2. The schema of the RDF graph (cf. Figure 4.1) extracted from DBpedia related to Elon Musk represented in the turtle [56] format, below:

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix voc: <http://www.example.org/voc/> .

voc:Organisation rdf:type rdfs:Class .
voc:Person rdf:type rdfs:Class .
voc:City rdf:type rdfs:Class .
voc:Country rdf:type rdfs:Class .
xsd:string rdf:type rdfs:Class .
xsd:date rdf:type rdfs:Class .
xsd:int rdf:type rdfs:Class .
voc:ceo rdf:type rdf:Property .
voc:ceo rdfs:domain voc:Organisation .
voc:ceo rdfs:range voc:Person .
voc:location rdfs:domain voc:Organisation .
voc:location rdfs:range voc:City .
voc:country rdf:type rdf:Property .
voc:country rdfs:domain voc:City .
voc:country rdfs:range voc:Country .
voc:is_location_of rdf:type rdf:Property .
voc:is_location_of rdfs:domain voc:City .
voc:is_location_of rdfs:range voc:Organisation .
voc:name rdf:type rdf:Property .
voc:name rdfs:domain voc:Organisation .
```

```

voc:name rdfs:domain voc:City .
voc:name rdfs:domain voc:Country .
voc:name rdfs:range xsd:string .
voc:creation rdf:type rdf:Property .
voc:creation rdfs:domain voc:Organisation .
voc:creation rdfs:range xsd:date .
voc:location rdf:type rdf:Property .
voc:birthName rdf:type rdf:Property .
voc:birthName rdfs:domain voc:Person .
voc:birthName rdfs:range xsd:string .
voc:age rdf:type rdf:Property .
voc:age rdfs:domain voc:Person .
voc:age rdfs:range xsd:int .

```

Note that: a resource class rc is defined by a triple of the form $(rc \text{ rdf:type rdfs:Class})$; a property class pc is defined by a triple of the form $(pc \text{ rdf:type rdf:Property})$; a triple $(pc \text{ rdfs:domain } rc_1)$ indicates that the resource class rc_1 is part of the domain of pc (i.e. a resource of class rc_1 could have an outgoing property pc); a triple $(pc \text{ rdfs:range } rc_2)$ indicates that the resource class rc_2 is part of the range of pc (i.e. a resource of class rc_1 could have an incoming property pc).

If the range of a property class pc is a resource class (defined by the user), then pc is called an object property (e.g. `voc:ceo`). If the range is a datatype class, defined by RDF Schema or another vocabulary, then pc is called a datatype property (e.g. `age`). The IRIs `xsd:string`, `xsd:integer` and `xsd:dateTime` are examples of datatypes defined by XML Schema [183]. Let $I_{DT} \subset \mathbf{I}$ be the set of RDF datatypes.

Note that the RDF schema presented in Example 4.1.2 provides a complete description of resource classes and property classes. However, in practice, it is possible to find incomplete or partial RDF schema descriptions. In particular, a datatype could not be defined as a resource class, and a property could not define its domain or its range.

We will assume that a partial schema can be “normalized” to be a total schema. In this sense, we will use the term `rdfs:Resource`² to complete the definition of properties without domain or range. For instance, suppose that our sample RDF Schema does not define the range of the property class `voc:ceo`. In such case, we will include the triple $(\text{voc:ceo}, \text{rdfs:range}, \text{rdfs:Resource})$ to complete the definition of `voc:ceo`.

Now, we introduce the notion of RDF graph schema as a formal way to represent an RDF schema description. Assume that $I_V \subset \mathbf{I}$ is the set that includes the RDF Schema terms `rdf:type`, `rdfs:Class`, `rdfs:Property`, `rdfs:domain` and `rdfs:range`.

Definition 3 (RDF Graph Schema). An RDF graph schema is defined as a tuple $S^R = (N_S, E_S, \phi, \varphi)$ where:

- N_S is a finite set of nodes representing resource classes;

² According to the RDF Schema specification [8], `rdfs:Resource` denotes the class of everything.

- E_S is a finite set of edges representing property classes;
- $\phi : (N_S \cup E_S) \rightarrow I \setminus I_V$ is a total function that associates each node or edge with an IRI representing a class identifier;
- $\varphi : E_S \rightarrow (N_S \times N_S)$ is a total function that associates each property class with a pair of resource classes. \square

Recall that I_{DT} denotes the set of RDF datatypes. Given an RDF Schema description D , the procedure to create an RDF graph schema $S^R = (N_S, E_S, \phi, \varphi)$ from D is given as follows:

1. Let $C = \{rc \mid (rc, \text{rdf:type}, \text{rdfs:Class}) \in DV(pc, \text{rdfs:domain}, rc) \in DV(pc, \text{rdfs:range}, rc) \in D\}$
2. For each $rc \in C$, we create $n \in N_S$ with $\phi(n) = rc$
3. For each pair of triples $(pc, \text{rdfs:domain}, rc_1)$ and $(pc, \text{rdfs:range}, rc_2)$ in D , we create $e \in E_S$ with $\phi(e) = pc$ and $\varphi(e) = (n_1, n_2)$, satisfying that $n_1, n_2 \in N_S$, $\phi(n_1) = rc_1$ and $\phi(n_2) = rc_2$.

According to the above definition, the RDF graph schema shown in Example 4.1.2 can be formally represented as follows:

- 1 $N_S = \{n_1, n_2, n_3, n_4, n_5, n_6, n_7\}$,
- 2 $E_S = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9, e_{10}\}$;
- 3 $\phi(n_1) = \{\text{voc:Organisation}\}$, $\phi(n_2) = \{\text{voc:Person}\}$, $\phi(n_3) = \{\text{voc:City}\}$,
 $\phi(n_4) = \{\text{voc:Country}\}$, $\phi(n_5) = \{\text{xsd:string}\}$, $\phi(n_6) = \{\text{xsd:date}\}$, $\phi(n_7) = \{\text{xsd:int}\}$
- 4 $\phi(e_1) = \{\text{voc:ceo}\}$, $\phi(e_2) = \{\text{voc:location}\}$, $\phi(e_3) = \{\text{voc:country}\}$,
 $\phi(e_4) = \{\text{voc:is_location_of}\}$, $\phi(e_5) = \{\text{voc:name}\}$, $\phi(e_6) = \{\text{voc:creation}\}$,
 $\phi(e_7) = \{\text{voc:birthName}\}$, $\phi(e_8) = \{\text{voc:age}\}$, $\phi(e_9) = \{\text{voc:name}\}$,
 $\phi(e_{10}) = \{\text{voc:name}\}$,
- 5 $\varphi(e_1) = (n_1, n_2)$, $\varphi(e_2) = (n_1, n_3)$, $\varphi(e_3) = (n_3, n_4)$, $\varphi(e_4) = (n_4, n_1)$, $\varphi(e_5) = (n_1, n_5)$,
 $\varphi(e_6) = (n_1, n_6)$, $\varphi(e_7) = (n_2, n_5)$, $\varphi(e_8) = (n_2, n_7)$, $\varphi(e_9) = (n_3, n_5)$,
 $\varphi(e_{10}) = (n_4, n_5)$.

Figure 4.2 shows a graphical representation of an RDF schema description from example 4.1.2.

4.1.3 Valid RDF Graph

Given the definitions of RDF graph and RDF graph schema, we introduce the notion of *Valid RDF graph* as the procedure to verify that an RDF graph satisfies the data and structure restrictions established by an RDF graph schema.

Definition 4 (Valid RDF graph). Given an RDF graph schema $S^R = (N_S, E_S, \phi, \varphi)$ and an RDF graph $G^R = (N_R, N_L, E_O, E_D, \alpha_R, \alpha_L, \beta_O, \beta_D, \delta)$, we say that G^R is valid with respect to S^R , denoted as $G^R \models S^R$, iff:

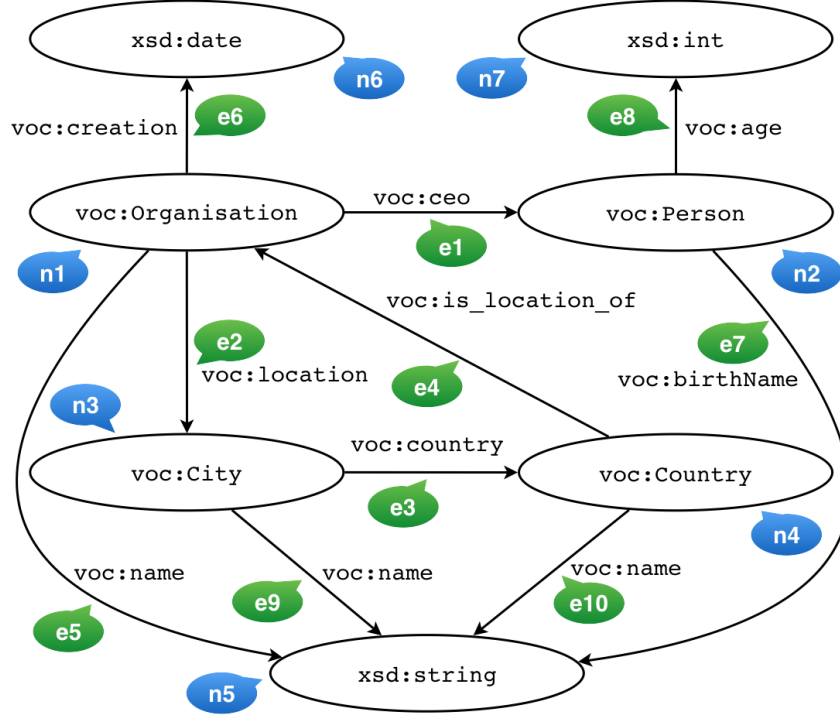


Figure 4.2: The Schema of the RDF graph shown in Figure 4.1.

1. for each $r \in N_R \cup N_L$, it applies that there is $rc \in N_S$ where $\delta(r) = \phi(rc)$;
2. for each $e \in E_O$ with $\beta_O(e) = (n, n')$, it applies that there is $pc \in E_S$ where $\delta(e) = \phi(pc)$, $\varphi(pc) = (rc, rc')$, $\delta(n) = \phi(rc)$ and $\delta(n') = \phi(rc')$.
3. for each $e \in E_D$ with $\beta_D(e) = (n, n')$, it applies that there is $pc \in E_S$ where $\delta(e) = \phi(pc)$, $\varphi(pc) = (rc, rc')$, $\delta(n) = \phi(rc)$ and $\delta(n') = \phi(rc')$. \square

Here, condition (1) validates that every resource node is labeled with a resource class defined by the schema; condition (2) verifies that each object property edge, and the pairs of resource nodes that it connects, are labeled with the corresponding resource classes; and condition (3) verifies that each datatype property edge, and the pairs of nodes that it connects (i.e. a resource node and a literal node), are labeled with the corresponding resource classes.

Finally, we present the notion of RDF database.

Definition 5 (RDF Database). An RDF database is a pair (S^R, G^R) where S^R is an RDF graph schema and G^R is an RDF graph satisfying that $G^R \models S^R$.

4.2 Property Graph Database

In this section, we introduce the main elements comprising of Property graph (PG) databases (or simply graph databases). Specifically, we formally define the concepts of a Property graph, Property graph Schema, and a valid Property graph. Furthermore, we illustrate these concepts via corresponding examples.

4.2.1 Property Graph

A Property Graph is a labeled directed multigraph whose main characteristic is that nodes and edges can contain a set (possibly empty) of *name-value* pairs referred to as *properties*. From the point of view of data modeling, each node represents an entity, each edge represents a relationship (between two entities), and each property represents a specific characteristic (of an entity or a relationship). Currently, there are no standard definitions for the notions of PG and PG Schema. We therefore present formal definitions that cover most of the features provided by current PG database systems.

Example 4.2.1. Consider an example of a Property graph data representation, related to Elon Musk and Tesla Incorporation, represented using the GraphML [73] format below:

```
<?xml version="1.0" encoding="UTF-8"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">

<key id="a0" for="node" attr.name="label" attr.type="string"/>
<key id="a1" for="node" attr.name="birthName" attr.type="string"/>
<key id="a2" for="node" attr.name="name" attr.type="string"/>
<key id="a3" for="node" attr.name="age" attr.type="int"/>
<key id="a4" for="node" attr.name="creation" attr.type="string"/>
<key id="a5" for="edge" attr.name="label" attr.type="string"/>
<key id="a6" for="edge" attr.name="since" attr.type="int"/>

<graph id="G" edgedefault="directed">
<node id="n0">
<data key="a0">Person</data>
<data key="a1">Elon Reeve Musk</data>
<data key="a3">46</data>
</node>
<node id="n1">
<data key="a0">Organisation</data>
<data key="a2">Tesla, Incorporation</data>
<data key="a4">2003-07-01</data>
```

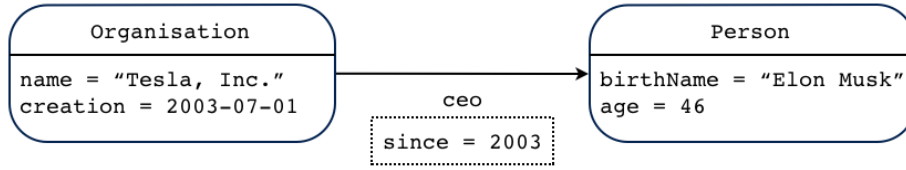


Figure 4.3: A graphical representation of a Property Graph.

```

<edge id="e0" source="n0" target="n2">
<data key="a5">ceo</data>
<data key="a6">2003</data>
</edge>
</graph>
</graphml>
    
```

Figure 4.3 presents the GraphML data from example 4.2.1 in a graphical representation of a Property Graph. The circles represent nodes, the arrows represent edges, and the boxes contain the properties for nodes and edges. Next we introduce a formal definition for this graph-based structure.

Assume that \mathbb{L} is an infinite set of labels (for nodes, edges and properties), \mathbb{V} is an infinite set of (atomic or complex) values, and \mathbb{T} is a finite set of data types (e.g. *string*, *integer*, *date*, etc.). A value in \mathbb{V} will be distinguished as a quoted string. Given a value $v \in \mathbb{V}$, the function $\text{type}(v)$ returns the datatype of v . Given a set S , $\mathcal{P}^+(S)$ denotes the set of non-empty subsets of S .

Definition 6 (Property Graph). A Property Graph is defined as a tuple $G^P = (\mathbb{N}, \mathbb{E}, \mathbb{P}, \Gamma, \Upsilon, \Sigma, \Delta)$ where:

- \mathbb{N} is a finite set of nodes, \mathbb{E} is a finite set of edges, \mathbb{P} is a finite set of properties, and $\mathbb{N}, \mathbb{E}, \mathbb{P}$ are mutually disjoint sets;
- $\Gamma : (\mathbb{N} \cup \mathbb{E}) \rightarrow \mathbb{L}$ is a total function that associates each node or edge with a label;
- $\Upsilon : \mathbb{P} \rightarrow (\mathbb{L} \times \mathbb{V})$ is a total function that assigns a label-value pair to each property.
- $\Sigma : \mathbb{E} \rightarrow (\mathbb{N} \times \mathbb{N})$ is a total function that associates each edge with a pair of nodes;
- $\Delta : (\mathbb{N} \cup \mathbb{E}) \rightarrow \mathcal{P}^+(\mathbb{P})$ is a partial function that associates a node or edge with a non-empty set of properties, satisfying that $\Delta(o_1) \cap \Delta(o_2) = \emptyset$ for each pair of objects $o_1, o_2 \in \text{dom}(\Delta)$ \square

The above definition supports PGs with the following features: a pair of nodes can have zero or more edges; each node or edge has a single label; each node or edge can have zero or more properties; and a node or edge can have the same label-value pair one or more times.

On the other side, the above definition does not support multiple labels for nodes or edges. We

have two reasons to justify this restriction. First, this feature is not supported by all graph database systems. Second, it complicates the definition of schema-instance consistency.

Given two nodes $n_1, n_2 \in N$ and an edge $e \in E$, satisfying that $\Sigma(e) = (n_1, n_2)$, we will use $e = (n_1, n_2)$ as a shorthand representation for e , where n_1 and n_2 are called the “source node” and the “target node” of e respectively. Furthermore, it is possible for two nodes/edges to have the same properties values, but not the same property objects (i.e. $\sigma(o1) \cap \sigma(o2) = \emptyset$). Given the above definition, the sample property graph presented in Figure 4.3 can be formally described as follows:

```

1  $\mathbb{N} = \{n_1, n_2\}$ ,
2  $\mathbb{E} = \{e_1\}$ ,
3  $\mathbb{P} = \{p_1, p_2, p_3, p_4, p_5\}$ ,
4  $\Gamma(n_1) = \{\text{Organisation}\}$ ,  $\Gamma(n_2) = \{\text{Person}\}$ ,
5  $\Gamma(e_1) = \{\text{ceo}\}$ ,
6  $\Upsilon(p_1) = (\text{name}, \text{"Tesla, Inc."})$ ,  $\Upsilon(p_2) = (\text{creation}, \text{2003-07-01})$ ,
    $\Upsilon(p_3) = (\text{birthName}, \text{"Elon Musk"})$ ,  $\Upsilon(p_4) = (\text{age}, 46)$ ,  $\Upsilon(p_5) = (\text{since}, 2003)$ 
7  $\Sigma(e_1) = \{n_1, n_2\}$ ,
8  $\Delta(n_1) = \{p_1, p_2\}$ ,  $\Delta(n_2) = \{p_3, p_4\}$ ,  $\Delta(e_1) = \{p_5\}$ .

```

4.2.2 Property Graph Schema

A Property graph schema defines the types of nodes, edges and properties allowed in a given Property graph database. As with relational databases, the schema allows to define and validate the structure of a property graph. In GraphML [73], the schema information is embedded within the data itself, which is defined by the XML attribute-tags. For the GraphML document to be valid, it is necessary to have the headers defined either in a DTD (document type definition) or an XML schema.

Example 4.2.2. Consider an example of a Property graph schema data, corresponding to the example 4.2.1, represented in GraphML [73] format below:

```

<?xml version="1.0" encoding="UTF-8"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
<key id="a0" for="node" attr.name="label" attr.type="string"/>
<key id="a1" for="node" attr.name="birthName" attr.type="string"/>
<key id="a2" for="node" attr.name="age" attr.type="int"/>
<key id="a3" for="node" attr.name="creation" attr.type="string"/>
<key id="a4" for="edge" attr.name="label" attr.type="string"/>
<key id="a5" for="edge" attr.name="since" attr.type="int"/>
<graph id="G" edgedefault="directed">
</graph>

```

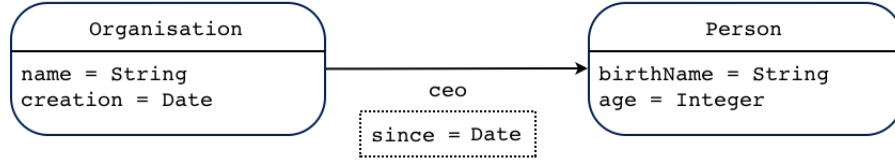


Figure 4.4: The Schema of the Property Graph as shown in Fig. 4.3.

</graphml>

Figure 4.4 shows a graphical representation of a Property graph schema data from example 4.2.2.

Next, we present the formal definitions of Property Graph Schema and introduce the notion of Valid Property Graph.

Definition 7 (Property Graph Schema). A property graph schema is defined as a tuple $S^P = (\mathbb{N}_S, \mathbb{E}_S, \mathbb{P}_S, \Theta, \Pi, \Phi, \Psi)$ where:

- \mathbb{N}_S is a finite set of node types;
- \mathbb{E}_S is a finite set of edge types;
- \mathbb{P}_S is a finite set of property types;
- $\Theta : (\mathbb{N}_S \cup \mathbb{E}_S) \rightarrow \mathbb{L}$ is a total function that assigns a label to each node or edge;
- $\Pi : \mathbb{P}_S \rightarrow (\mathbb{L} \times \mathbb{T})$ is a total function that associates each property type with a property label and a data type;
- $\Phi : \mathbb{E}_S \rightarrow (\mathbb{N}_S \times \mathbb{N}_S)$ is a total function that associates each edge type with a pair of node types;
- $\Psi : (\mathbb{N}_S \cup \mathbb{E}_S) \rightarrow \mathcal{P}^+(\mathbb{P}_S)$ is a partial function that associates a node or edge type with a non-empty set of property types, satisfying that $\Psi(o_1) \cap \Psi(o_2) = \emptyset$, for each pair of objects $o_1, o_2 \in \text{dom}(\Psi)$ □

The property graph schema shown in Figure 4.4 can be formally described using our formal definition as follows:

- 1 $\mathbb{N}_S = \{n_1, n_2\}$,
- 2 $\mathbb{E}_S = \{e_1\}$,
- 3 $\mathbb{P}_S = \{p_1, p_2, p_3, p_4, p_5\}$,
- 4 $\Theta(n_1) = \{\text{Organisation}\}$, $\Theta(n_2) = \{\text{Person}\}$, $\Theta(e_1) = \{\text{ceo}\}$,
- 5 $\Pi(p_1) = (\text{name}, \text{String})$, $\Pi(p_2) = (\text{creation}, \text{Date})$, $\Pi(p_3) = (\text{birthName}, \text{String})$,
 $\Pi(p_4) = (\text{age}, \text{Integer})$, $\Pi(p_5) = (\text{since}, \text{Date})$,
- 6 $\Phi(e_1) = (n_1, n_2)$,
- 7 $\Psi(n_1) = \{p_1, p_2\}$, $\Psi(n_2) = \{p_3, p_4\}$, $\Psi(e_1) = \{p_5\}$

4.2.3 Valid Property Graph

Given the definitions of Property graph and Property graph schema, we introduce the notion of *Valid Property graph* as the procedure to verify that an Property graph satisfies the data and structure restrictions established by an Property graph schema.

Definition 8 (Valid Property Graph). Given a PG schema $S^P = (\mathbb{N}_S, \mathbb{E}_S, \mathbb{P}_S, \Theta, \Pi, \Phi, \Psi)$ and a PG $G^P = (\mathbb{N}, \mathbb{E}, \mathbb{P}, \Gamma, \Upsilon, \Sigma, \Delta)$, we say that G^P is valid with respect to S^P , denoted as $G^P \models S^P$, iff:

1. for each $n \in \mathbb{N}$, it applies that there is $nt \in \mathbb{N}_S$ satisfying that:
 - (a) $\Gamma(n) = \Theta(nt)$;
 - (b) for each $p \in \Delta(n)$, there is $pt \in \Psi(nt)$ satisfying that $\Upsilon(p) = (l, v)$ and $\Pi(pt) = (l, \text{type}(v))$.
2. for each $e = (n, n') \in \mathbb{E}$, it applies that there is $et \in \mathbb{E}_S$ with $\Phi(et) = (nt, nt')$ satisfying that:
 - (a) $\Gamma(e) = \Theta(et)$, $\Gamma(n) = \Theta(nt)$, $\Gamma(n') = \Theta(nt')$;
 - (b) for each $p' \in \Delta(e)$, there is $pt' \in \Psi(et)$ satisfying that $\Upsilon(p') = (l', v')$ and $\Pi(pt') = (l', \text{type}(v'))$.

Here, condition (1a) validates that every node is labeled with a node type defined by the schema; condition (1b) verifies that each node contains the properties defined by its node type; condition (2a) verifies that each edge, and the pairs of nodes that it connects, are labeled with an edge type, and the corresponding node types; and condition (2b) verifies that each edge contains the properties defined by the schema.

Finally, we present the notion of PG database.

Definition 9 (Property Graph Database). A property graph database D^P is a pair (S^P, G^P) where S^P is a PG schema and G^P is a PG satisfying that $G^P \models S^P$.

4.3 Direct Mappings for Data Transformation

In this section, we formally define the mappings for transforming RDF graphs (source model) into the Property graphs (PGs) (target model).

Upon comparison of RDF graphs and PGs, we see that both share the main characteristics of a traditional labeled directed graph, that is, nodes and edges contain labels, the edges are directed, and multiple edges are possible between a given pair of nodes. However, there are also some differences between them:

- An RDF graph allows three types of nodes (IRIs, blank nodes and literals) whereas a PG allows a single type of node;
- Each node or edge in an RDF graph contains just a single value (i.e. a label), whereas each node or edge in a PG could contain multiple labels and properties respectively;
- An RDF graph supports multi-value properties, whereas a PG usually just support mono-value properties;
- An RDF graph allows to have edges between edges, a feature which isn't supported in a PG (by definition);
- A node in an RDF graph could be associated with zero or more classes or resources, while a node in a PG usually has a single node type.

In addition to the above structural differences, RDF Schema gives special semantics to the terms in its vocabulary. For example, the terms `rdf:Statement`, `rdf:subject`, `rdf:predicate` and `rdf:object` can be used to describe explicitly RDF statements. This feature, called “reification”, is not studied in this article as it is rarely used in practice.

A very interesting feature of both, RDF and PG databases, is the support for schema-less databases, i.e. the databases could not have a fixed data structure. In the particular case of RDF, it is possible to find three types of datasets: datasets without schema definitions, datasets that merge data and schema; and datasets that separate schema and instance.

Depending on whether or not the input RDF dataset has schema, the database mappings can be classified into two types: **(i) *schema-dependent***: one that generates a target PG schema from the input RDF graph schema, and then transforms the RDF graph into a PG; and **(ii) *schema-independent***: one that creates a generic PG schema (based on a predefined structure) and then transforms the RDF graph into a PG. Additionally, we maintain a *dictionary* (e.g. DRPD [181]) which preserves the prefix form of IRIs (e.g. `dbo:Person`) with their extended form (`http://dbpedia.org/ontology/Person`) and the class label ("`Person`") associated with them. We now formally define and discuss each of the two types of direct transformations.

4.3.1 Simple Database Mapping (SDM)

This section describes the schema-independent database mapping \mathcal{DM}_1 which allows to transform an schema-less RDF database into a schema-less PG database. \mathcal{DM}_1 is just composed of an instance mapping which allows to transform the input RDF graph into a PG graph.

Given an RDF database $D^R = (\emptyset, G^R)$, we define the database mapping $\mathcal{DM}_1 = (\emptyset, \mathcal{IM}_1)$ such that $\mathcal{DM}_1(D^R) = (\emptyset, G^P)$ where $G^P = \mathcal{IM}_1(G^R)$. The instance mapping \mathcal{IM}_1 is defined next.

Instance Mapping \mathcal{IM}_1

The instance mapping \mathcal{IM}_1 is defined as follows:

Definition 10 (Instance mapping \mathcal{IM}_1). Let $G^R = (N_R, N_L, E_O, E_D, \alpha_R, \alpha_L, \beta_O, \beta_D, \delta)$ be an RDF graph and $G^P = (\mathbb{N}, \mathbb{E}, \mathbb{P}, \Gamma, \Upsilon, \Sigma, \Delta)$ be a PG. The instance mapping $\mathcal{IM}_1(G^R) = G^P$ is defined as follows:

1. For each $r \in N_R$
 - There will be $n \in \mathbb{N}$ with $\Gamma(n) = \text{name}(\delta(r))$
2. For each $op \in E_O$ satisfying that $\beta_O(op) = (r, r')$ where $r, r' \in N_R$
 - There will be $e \in \mathbb{E}$ with $\Gamma(e) = \text{name}(\delta(op))$ and $\Sigma(e) = (n_1, n_2)$ where $n_1, n_2 \in \mathbb{N}$ correspond to r, r' respectively.
3. For each $dp \in E_D$ satisfying that $\beta_D(dp) = (r, l)$ where $r \in N_R$ and $l \in N_L$
 - There will be $p \in \mathbb{P}$ with $\Upsilon(p) = (\text{name}(\delta(dp)), \alpha_L(l))$
 - $\Delta(n) = \Delta(n) \cup p$ such that $n \in \mathbb{N}$ corresponds to r .

In general terms, the instance mapping \mathcal{IM}_1 creates PG nodes from resource nodes, PG properties from datatype properties, and PG edges from object properties. Nodes, edges and properties are labeled with the name of the corresponding resource class label (defined by the function δ) or the name of the resource identifier (when function δ is undefined).

For example, the PG obtained after applying \mathcal{IM}_1 over the RDF graph shown in Figure 4.1 is shown in the listing 4.3.1 below:

```

1  $\mathbb{N} = \{n_1, n_2, n_3, n_4\}$ ,
2  $\mathbb{E} = \{e_1, e_2, e_3, e_4\}$ ,
3  $\mathbb{P} = \{p_5, p_6, p_7, p_8, p_9, p_{10}\}$ ,
4  $\Gamma(n_1) = \text{Organisation}$ ,  $\Gamma(n_2) = \text{Person}$ ,  $\Gamma(n_3) = \text{City}$ ,  $\Gamma(n_4) = \text{Country}$ ,  $\Gamma(e_1) = \text{ceo}$ ,
    $\Gamma(e_2) = \text{location}$ ,  $\Gamma(e_3) = \text{country}$ ,  $\Gamma(e_4) = \text{is\_location\_of}$ 
5  $\Upsilon(p_5) = (\text{name}, \text{"Tesla, Inc."})$ ,  $\Upsilon(p_6) = (\text{creation}, \text{"2003-07-01"})$ ,
    $\Upsilon(p_7) = (\text{birthName}, \text{"Elon\_Musk"})$ ,  $\Upsilon(p_8) = (\text{age}, \text{"46"})$ ,  $\Upsilon(p_9) = (\text{name}, \text{"Palo Alto"})$ ,
    $\Upsilon(p_{10}) = (\text{name}, \text{"US"})$ 
6  $\Sigma(e_1) = \{n_1, n_2\}$ ,  $\Sigma(e_2) = \{n_1, n_3\}$ ,  $\Sigma(e_3) = \{n_3, n_4\}$ ,  $\Sigma(e_4) = \{n_4, n_1\}$ ,
7  $\Delta(n_1) = \{p_5, p_6\}$ ,  $\Delta(n_2) = \{p_7, p_8\}$ ,  $\Delta(n_3) = \{p_9\}$ ,  $\Delta(n_4) = \{p_{10}\}$ .
    
```

Figure 4.5 shows a graphical representation of the PG described above.

Properties of \mathcal{DM}_1

In this section we evaluate the properties of the database mapping \mathcal{DM}_1 , i.e. computability, semantics preservation and information preservation. Recall that \mathcal{DM}_1 just contains the instance mapping \mathcal{IM}_1 , and the output is a PG database without RDF graph schema.

Proposition 1. The database mapping \mathcal{DM}_1 is computable.

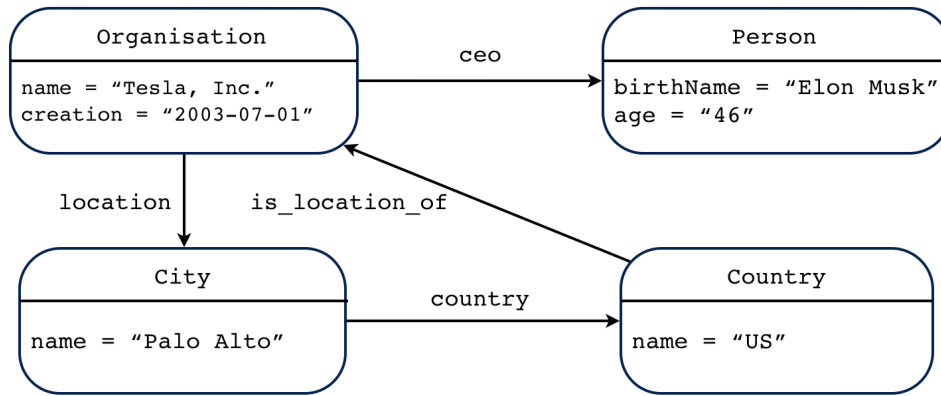


Figure 4.5: Property graph obtained after applying the instance mapping \mathcal{IM}_1 to the RDF graph shown in Figure 4.1.

It is easy to see that the procedure presented in Definition 10 can be implemented as an algorithm.

Proposition 2. The database mapping \mathcal{DM}_1 is semantic preserving.

Note that \mathcal{DM}_1 assumes that there is no RDF graph schema, i.e. no schema restrictions are considered. Moreover, the output PG database does not contain a PG schema. Hence, it is straightforward to see that \mathcal{DM}_1 is semantic preserving.

Proposition 3. The database mapping \mathcal{DM}_1 is not information preserving.

Note that the instance mapping \mathcal{IM}_1 loses multiple pieces of information from the input RDF graph. In particular, it extracts simple labels from IRIs and blank nodes (e.g. by removing the namespace part of a IRI). Hence, it is not possible to define an inverse mapping which is able to reconstruct all the original information.

Although the database mapping \mathcal{DM}_1 does not satisfy the information preservation property, it is a simple method to transform RDF datasets that contains a merge of data and schema. In particular, it works well with RDF graphs where each resource defines its resource class by means of the `rdf:type` term.

4.3.2 Generic Database Mapping (GDM)

This section describes the schema-independent database mapping \mathcal{DM}_2 which allows to transform a schema-less RDF database into a complete PG database. \mathcal{DM}_2 is composed of a schema mapping \mathcal{SM}_2 and an instance mapping \mathcal{IM}_2 such that \mathcal{SM}_2 generates a “generic” PG schema (always the same) and \mathcal{IM}_2 allows to generate a PG graph from the input RDF graph.

Given an RDF database $D^R = (\emptyset, G^R)$, we define the database mapping $\mathcal{DM}_2 = (\mathcal{SM}_2, \mathcal{IM}_2)$

such that $\mathcal{DM}_2(D^R) = (S^P, G^P)$ where S^P is a generic PG schema and $G^P = \mathcal{IM}_2(G^R)$. The schema mapping \mathcal{SM}_2 and the instance mapping \mathcal{IM}_2 are defined next.

Generic Property Graph Schema

First we introduce a property graph schema which is able to model any RDF graph.

Definition 11 (Generic Property Graph Schema). Let $S^* = (\mathbb{N}_S, \mathbb{E}_S, \mathbb{P}_S, \Theta, \Pi, \Phi, \Psi)$ be the PG schema defined as follows:

- 1 $\mathbb{N}_S = \{n_1, n_2, n_3\}$,
- 2 $\mathbb{E}_S = \{e_1, e_2, e_3, e_4, e_5, e_6\}$,
- 3 $\mathbb{P}_S = \{p_1, p_2, p_3, p_4, p_5, p_6\}$,
- 4 $\Theta(n_1) = \{\text{Resource}\}$, $\Theta(n_2) = \{\text{BlankNode}\}$, $\Theta(n_3) = \{\text{Literal}\}$,
- 5 $\Theta(e_1) = \{\text{ObjectProperty}\}$, $\Theta(e_2) = \{\text{ObjectProperty}\}$, $\Theta(e_3) = \{\text{ObjectProperty}\}$,
 $\Theta(e_4) = \{\text{ObjectProperty}\}$, $\Theta(e_5) = \{\text{DatatypeProperty}\}$,
 $\Theta(e_6) = \{\text{DatatypeProperty}\}$,
- 6 $\Pi(p_1) = (\text{iri}, \text{String})$, $\Pi(p_2) = (\text{type}, \text{String})$, $\Pi(p_3) = (\text{id}, \text{String})$, $\Pi(p_4) = (\text{type}, \text{String})$,
 $\Pi(p_5) = (\text{value}, \text{String})$, $\Pi(p_6) = (\text{type}, \text{String})$, $\Pi(p_7) = (\text{type}, \text{String})$,
 $\Pi(p_8) = (\text{type}, \text{String})$, $\Pi(p_9) = (\text{type}, \text{String})$,
- 7 $\Phi(e_1) = (n_1, n_1)$, $\Phi(e_2) = (n_2, n_2)$, $\Phi(e_3) = (n_1, n_2)$, $\Phi(e_4) = (n_2, n_1)$, $\Phi(e_5) = (n_1, n_3)$,
 $\Phi(e_6) = (n_2, n_3)$,
- 8 $\Psi(n_1) = \{p_1, p_2\}$, $\Psi(n_2) = \{p_3, p_4\}$, $\Psi(n_3) = \{p_5, p_6\}$,
- 9 $\Psi(e_1) = \{p_7\}$, $\Psi(e_2) = \{p_8\}$, $\Psi(e_3) = \{p_9\}$.

In the above definition: the node type `Resource` will be used to represent RDF resources, the node type `Literal` will be used to represent RDF literals, the edge type `ObjectProperty` allows to represent object properties (i.e. relationships between RDF resources), and the edge type `DatatypeProperty` allows representing datatype properties (i.e. relationships between an RDF resource and a literal). Figure 4.6 shows a graphical representation of the generic PG schema.

Instance Mapping \mathcal{IM}_2

Now, we define the instance mapping \mathcal{IM}_2 which takes an RDF graph and produces a PG following the restrictions established by the generic PG schema defined above.

Definition 12 (Instance mapping \mathcal{IM}_2). Let $G^R = (N_R, N_L, E_O, E_D, \alpha_R, \alpha_L, \beta_O, \beta_D, \delta)$ be an RDF graph and $G^P = (\mathbb{N}, \mathbb{E}, \mathbb{P}, \Gamma, \Upsilon, \Sigma, \Delta)$ be a PG. The instance mapping $\mathcal{IM}_2(G^R) = G^P$ is defined as follows:

1. For each $r \in N_R$
 - There will be $n \in \mathbb{N}$ with $\Gamma(n) = \text{Resource}$

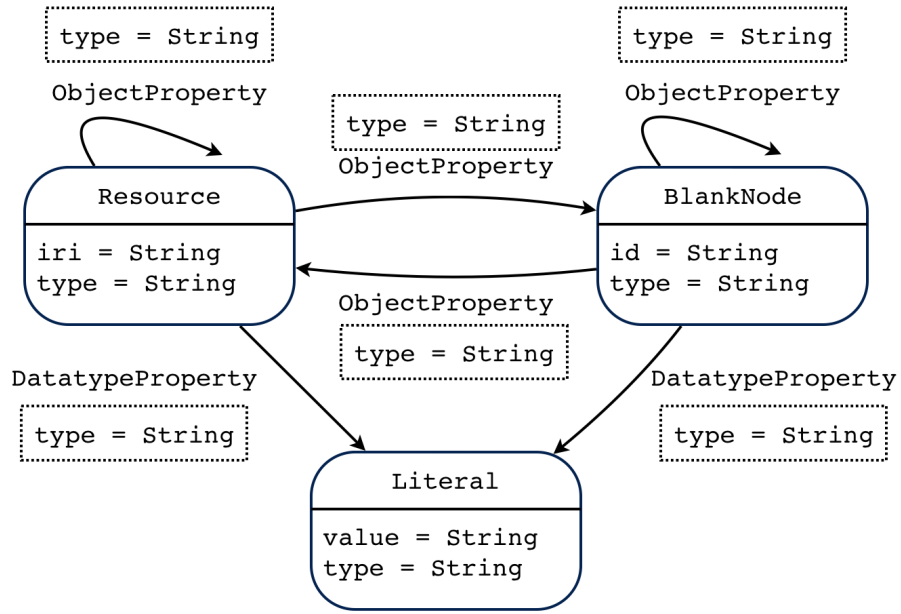


Figure 4.6: A generic Property graph schema.

- There will be $p \in \mathbb{P}$
 - If $\alpha_R(r) \in I$ then $\Upsilon(p) = (\text{iri}, \alpha_R(r))$
 - If $\alpha_R(r) \in B$ then $\Upsilon(p) = (\text{id}, \alpha_R(r))$
 - There will be $p' \in \mathbb{P}$ with $\Upsilon(p') = (\text{type}, \delta(r))$
 - $\Delta(n) = \{p, p'\}$
2. For each $l \in N_L$
- There will be $n \in \mathbb{N}$ with $\Gamma(n) = \text{Literal}$
 - There will be $p \in \mathbb{P}$ with $\Upsilon(p) = (\text{value}, \alpha_L(l))$
 - There will be $p' \in \mathbb{P}$ with $\Upsilon(p') = (\text{type}, \delta(l))$
 - $\Delta(n) = \{p, p'\}$
3. For each $op \in E_O$ satisfying that $\beta_O(op) = (r_1, r_2)$ where $r_1, r_2 \in N_R$
- There will be $e \in \mathbb{E}$ with $\Gamma(e) = \text{ObjectProperty}$, and $\Sigma(e) = (n_1, n_2)$ where $n_1, n_2 \in \mathbb{N}$ correspond to $r_1, r_2 \in N_R$ respectively
 - There will be $p \in \mathbb{P}$ with $\Upsilon(p) = (\text{type}, \delta(op))$
 - $\Delta(e) = \{p\}$

4. For each $dp \in E_D$ satisfying that $\beta_D(dp) = (r, l)$ where $r \in N_R$ and $l \in N_L$
 - There will be $e \in \mathbb{E}$ with $\Gamma(e) = \text{DatatypeProperty}$, and $\Sigma(e) = (n_1, n_2)$ where $n_1, n_2 \in \mathbb{N}$ correspond to r and l respectively
 - There will be $p \in \mathbb{P}$ with $\Upsilon(p) = (\text{type}, \delta(dp))$
 - $\Delta(e) = \{p\}$

According to the above definition, the instance mapping \mathcal{IM}_2 creates PG nodes from resource nodes and literal nodes, and PG edges from datatype properties and object properties. The property `type` is used to maintain resource class identifiers and RDF datatypes. The property `iri` is used to store the IRI of RDF resources and properties. The property `value` is used to maintain a literal value.

For example, the PG obtained after applying \mathcal{IM}_2 over the RDF graph shown in Figure 4.1 is shown in the listing 4.3.2 below:

```

1  $\mathbb{N} = \{n_1, \dots, n_{10}\}$ ,
2  $\mathbb{E} = \{e_1, \dots, e_{10}\}$ ,
3  $\mathbb{P} = \{p_1, \dots, p_{30}\}$ ,
4  $\Gamma(n_1) = \text{Resource}$ ,  $\Upsilon(p_1) = (\text{iri}, \text{"ex:Tesla\_Inc"})$ ,  $\Upsilon(p_2) = (\text{type}, \text{"voc:Organisation"})$ ,
    $\Delta(n_1) = \{p_1, p_2\}$ ,
5  $\Gamma(n_2) = \text{Resource}$ ,  $\Upsilon(p_3) = (\text{iri}, \text{"ex:Elon\_Musk"})$ ,  $\Upsilon(p_4) = (\text{type}, \text{"voc:Person"})$ ,
    $\Delta(n_2) = \{p_3, p_4\}$ ,
6  $\Gamma(n_3) = \text{BlankNode}$ ,  $\Upsilon(p_5) = (\text{id}, \text{"_:b1"})$ ,  $\Upsilon(p_6) = (\text{type}, \text{"voc:City"})$ ,  $\Delta(n_3) = \{p_4, p_5\}$ ,
7  $\Gamma(n_4) = \text{BlankNode}$ ,  $\Upsilon(p_7) = (\text{id}, \text{"_:b2"})$ ,  $\Upsilon(p_8) = (\text{type}, \text{"voc:City"})$ ,  $\Delta(n_4) = \{p_6, p_7\}$ ,
8  $\Gamma(n_5) = \text{Literal}$ ,  $\Upsilon(p_9) = (\text{value}, \text{"Tesla, Inc."})$ ,  $\Upsilon(p_{10}) = (\text{type}, \text{"xsd:string"})$ ,
    $\Delta(n_5) = \{p_9, p_{10}\}$ ,
9  $\Gamma(n_6) = \text{Literal}$ ,  $\Upsilon(p_{11}) = (\text{value}, \text{"2003-07-01"})$ ,  $\Upsilon(p_{12}) = (\text{type}, \text{"xsd:date"})$ ,
    $\Delta(n_6) = \{p_{11}, p_{12}\}$ ,
10  $\Gamma(n_7) = \{\text{Literal}\}$ ,  $\Upsilon(p_{13}) = (\text{value}, \text{"Elon Musk"})$ ,  $\Upsilon(p_{14}) = (\text{type}, \text{"xsd:string"})$ ,
    $\Delta(n_7) = \{p_{13}, p_{14}\}$ ,
11  $\Gamma(n_8) = \{\text{Literal}\}$ ,  $\Upsilon(p_{15}) = (\text{value}, \text{"46"})$ ,  $\Upsilon(p_{16}) = (\text{type}, \text{"xsd:int"})$ ,  $\Delta(n_8) = \{p_{15}, p_{16}\}$ ,
12  $\Gamma(n_9) = \{\text{Literal}\}$ ,  $\Upsilon(p_{17}) = (\text{value}, \text{"Palo Alto"})$ ,  $\Upsilon(p_{18}) = (\text{type}, \text{"xsd:string"})$ ,
    $\Delta(n_9) = \{p_{17}, p_{18}\}$ ,
13  $\Gamma(n_{10}) = \{\text{Literal}\}$ ,  $\Upsilon(p_{19}) = (\text{value}, \text{"US"})$ ,  $\Upsilon(p_{20}) = (\text{type}, \text{"xsd:string"})$ ,
    $\Delta(n_{10}) = \{p_{19}, p_{20}\}$ ,
14  $\Gamma(e_1) = \text{ObjectProperty}$ ,  $\Sigma(e_1) = \{n_1, n_2\}$ ,  $\Upsilon(p_{21}) = (\text{type}, \text{"voc:ceo"})$ ,  $\Delta(e_1) = \{p_{21}\}$ ,
15  $\Gamma(e_2) = \text{ObjectProperty}$ ,  $\Sigma(e_2) = \{n_1, n_3\}$ ,  $\Upsilon(p_{22}) = (\text{type}, \text{"voc:location"})$ ,  $\Delta(e_2) = \{p_{22}\}$ ,
   ,
16  $\Gamma(e_3) = \text{ObjectProperty}$ ,  $\Sigma(e_3) = \{n_3, n_4\}$ ,  $\Upsilon(p_{23}) = (\text{type}, \text{"voc:country"})$ ,  $\Delta(e_3) = \{p_{23}\}$ ,
   ,
17  $\Gamma(e_4) = \text{ObjectProperty}$ ,  $\Sigma(e_4) = \{n_3, n_4\}$ ,  $\Upsilon(p_{24}) = (\text{type}, \text{"voc:is\_location\_of"})$ ,
    $\Delta(e_4) = \{p_{24}\}$ ,
18  $\Gamma(e_5) = \text{DatatypeProperty}$ ,  $\Sigma(e_5) = \{n_3, n_5\}$ ,  $\Upsilon(p_{25}) = (\text{type}, \text{"voc:name"})$ ,  $\Delta(e_5) = \{p_{25}\}$ ,
   ,
19  $\Gamma(e_6) = \text{DatatypeProperty}$ ,  $\Sigma(e_6) = \{n_1, n_6\}$ ,  $\Upsilon(p_{26}) = (\text{type}, \text{"voc:creation"})$ ,

```

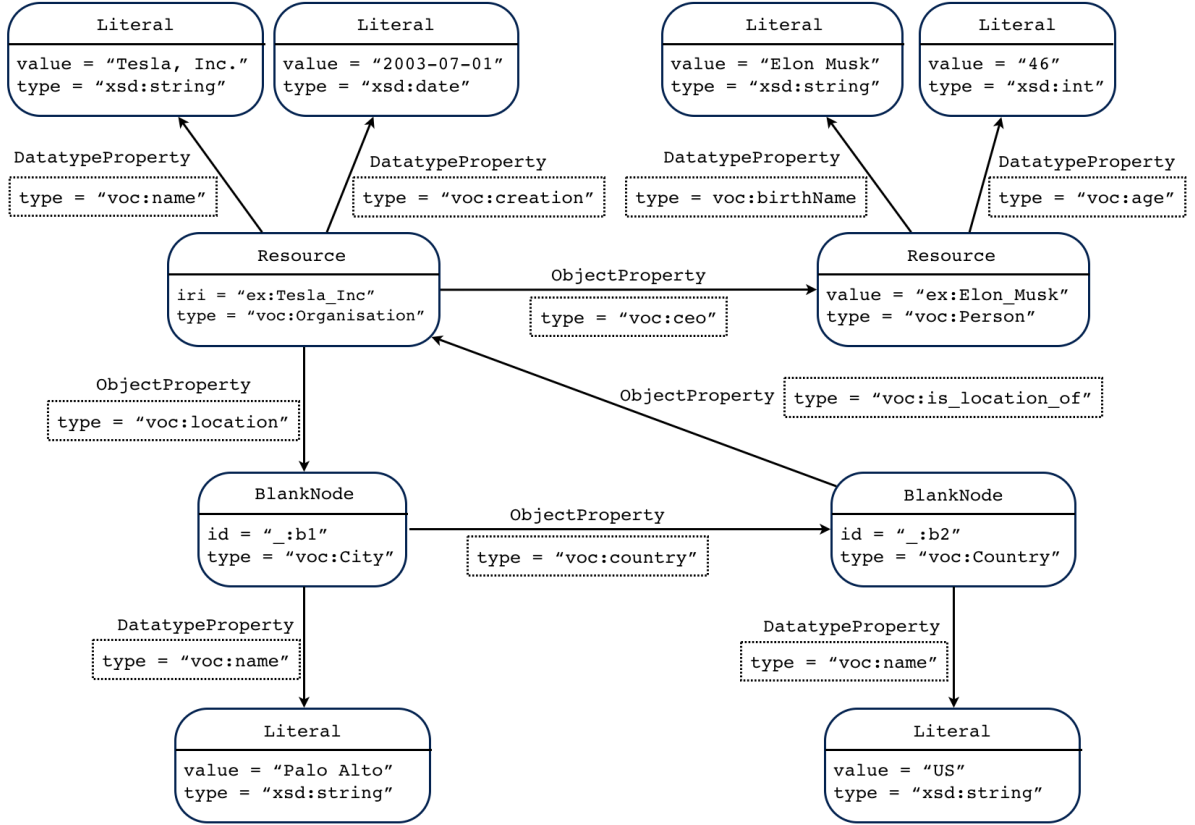


Figure 4.7: Property graph obtained after applying the instance mapping \mathcal{IM}_2 to the RDF graph shown in Figure 4.1.

$$\begin{aligned}
 & \Delta(e_6) = \{p_{26}\}, \\
 20 & \Gamma(e_7) = \text{DatatypeProperty}, \quad \Sigma(e_7) = \{n_2, n_7\}, \quad \Upsilon(p_{27}) = (\text{type}, \text{"voc:birthName"}), \\
 & \quad \Delta(e_7) = \{p_{27}\}, \\
 21 & \Gamma(e_8) = \text{DatatypeProperty}, \quad \Sigma(e_8) = \{n_2, n_8\}, \quad \Upsilon(p_{28}) = (\text{type}, \text{"voc:age"}), \quad \Delta(e_8) = \{p_{28}\}, \\
 22 & \Gamma(e_9) = \text{DatatypeProperty}, \quad \Sigma(e_9) = \{n_3, n_9\}, \quad \Upsilon(p_{29}) = (\text{type}, \text{"voc:name"}), \quad \Delta(e_9) = \{p_{29}\} \\
 & \quad , \\
 23 & \Gamma(e_{10}) = \text{DatatypeProperty}, \quad \Sigma(e_{10}) = \{n_4, n_{10}\}, \quad \Upsilon(p_{30}) = (\text{type}, \text{"voc:name"}), \\
 & \quad \Delta(e_{10}) = \{p_{30}\}.
 \end{aligned}$$

Figure 4.7 shows a graphical representation of the PG described above.

Properties of \mathcal{DM}_2

In this section we evaluate the properties of the database mapping \mathcal{DM}_2 . Recall that \mathcal{DM}_2 is formed by the schema mapping \mathcal{SM}_2 and the instance mapping \mathcal{IM}_2 , where \mathcal{SM}_2 always creates a generic PG schema S^* .

Proposition 4. The database mapping \mathcal{DM}_2 is *computable*.

It is not difficult to see that an algorithm can be created from the description of the instance mapping \mathcal{IM}_2 , presented in Definition 12.

Lemma 1. The database mapping \mathcal{DM}_2 is *semantics preserving*.

It is straightforward to see (by definition) that any PG graph created with the instance mapping \mathcal{IM}_2 will be valid with respect to the generic PG schema S^* .

Theorem 1. The database mapping \mathcal{DM}_2 is *information preserving*.

In order to prove that \mathcal{DM}_2 is information preserving, we need to provide a database mapping \mathcal{DM}_2^{-1} which allows to transform a PG database into an RDF database, and show that for every RDF database D^R , it applies that $D^R = \mathcal{DM}_2^{-1}(\mathcal{DM}_2(D^R))$.

Recalling that the objective of this section is to provide a schema-independent database mapping, we will assume that for any RDF database $D^R = (S^R, G^R)$, the RDF graph schema S^R is null or irrelevant to validate G^R . Hence, we just define an instance mapping \mathcal{IM}_2^{-1} which allows to transform a PG graph into an RDF database, such that for every RDF graph G^R , it must satisfy that $G^R = \mathcal{IM}_2^{-1}(\mathcal{IM}_2(G^R))$.

Definition 13 (Instance mapping \mathcal{IM}_2^{-1}). Let $G^P = (\mathbb{N}, \mathbb{E}, \mathbb{P}, \Gamma, \Upsilon, \Sigma, \Delta)$ be a property graph and $G^R = (N_R, N_L, E_O, E_D, \alpha_R, \alpha_L, \beta_O, \beta_D, \delta)$ be an RDF graph. The instance mapping $\mathcal{IM}_2^{-1}(G^P) = G^R$ is defined as follows:

1. For each $n \in \mathbb{N}$ satisfying that $\Gamma(n) = \text{Resource}$, $p_1, p_2 \in \Delta(n)$, $\Upsilon(p_1) = (\text{iri}, v_1)$ and $\Upsilon(p_2) = (\text{type}, v_2)$, then there will be $r \in N_R$ with $\alpha_R(r) = v_1$ and $\delta(r) = v_2$
2. For each $n \in \mathbb{N}$ satisfying that $\Gamma(n) = \text{BlankNode}$, $p_1, p_2 \in \Delta(n)$, $\Upsilon(p_1) = (\text{id}, v_1)$ and $\Upsilon(p_2) = (\text{type}, v_2)$, then there will be $r \in N_R$ with $\alpha_R(r) = v_1$ and $\delta(r) = v_2$
3. For each $n \in \mathbb{N}$ satisfying that $\Gamma(n) = \text{Literal}$, $p_1, p_2 \in \Delta(n)$, $\Upsilon(p_1) = (\text{value}, v_1)$ and $\Upsilon(p_2) = (\text{type}, v_2)$, then there will be $r \in N_L$ with $\alpha_L(r) = v_1$ and $\delta(r) = v_2$
4. For each $e \in \mathbb{E}$ satisfying that $\Gamma(e) = \text{ObjectProperty}$, $p \in \Delta(e)$, $\Upsilon(p) = (\text{type}, v)$, $\Sigma(e) = (n_1, n_2)$, then there will be $op \in E_O$ with $\delta(op) = v$, $\beta_O(op) = (r_1, r_2)$ where $r_1 \in N_R$ corresponds to $n_1 \in \mathbb{N}$, and $r_2 \in N_R$ corresponds to $n_2 \in \mathbb{N}$
5. For each $e \in \mathbb{E}$ satisfying that $\Gamma(e) = \text{DatatypeProperty}$, $p \in \Delta(e)$, $\Upsilon(p) = (\text{type}, v)$, $\Sigma(e) = (n_1, n_2)$, then there will be $dp \in E_D$ with $\delta(dp) = v$, $\beta_D(dp) = (r_1, r_2)$ where $r_1 \in N_R$ corresponds to $n_1 \in \mathbb{N}$, and $r_2 \in N_L$ corresponds to $n_2 \in \mathbb{N}$

Hence, the above method defines that for each node labeled with `Resource` or `BlankNode` is transformed into a resource node, each node labeled with `Literal` is transformed into a literal node, each edge labeled with `ObjectProperty` is transformed into a resource-resource edge, and each edge labeled with `DatatypeProperty` is transformed into a resource-literal edge. Additionally, the property `iri` is used to recover the original IRI identifier for `Resource` nodes, the property

id is used to recover the original identifier for BlankNode nodes, and the property type allows us to recover the IRI identifier of the resource class associated to each node or edge.

It is not difficult to verify that for any RDF graph G^R , we can produce a PG graph $G^P = \mathcal{SM}_3(G^R)$, and then recover G^R by using $\mathcal{IM}_3^{-1}(G^P)$.

4.3.3 Complete Database Mapping (CDM)

This section describes the schema-dependent database mapping \mathcal{DM}_3 which allows to transform a complete RDF database into a complete PG database. \mathcal{DM}_3 is composed of a schema mapping \mathcal{SM}_3 and an instance mapping \mathcal{IM}_3 such that \mathcal{SM}_3 generates a PG schema from the input RDF graph schema, and \mathcal{IM}_3 generates a PG graph from the input RDF graph.

Recall that I_{DT} is the set of IRIs referencing RDF datatypes, and \mathbb{T} is the set of PG datatypes. Assume that there is a total function $f : I_{DT} \rightarrow \mathbb{T}$ which maps RDF datatypes into PG datatypes. Additionally, assume that f^{-1} is the inverse function of f , i.e. f^{-1} maps PG datatypes into RDF datatypes.

Given an RDF database $D^R = (S^R, G^R)$, we define the database mapping $\mathcal{DM}_3 = (\mathcal{SM}_3, \mathcal{IM}_3)$ such that $\mathcal{DM}_3(D^R) = (S^P, G^P)$ where $S^P = \mathcal{SM}_3(S^R)$ and $G^P = \mathcal{IM}_3(G^R)$. The schema mapping \mathcal{SM}_3 and the instance mapping \mathcal{IM}_3 are defined next.

Schema mapping \mathcal{SM}_3

We define a schema mapping \mathcal{SM}_3 which takes an RDF graph schema as input and returns a PG Schema as output.

Definition 14 (Schema Mapping \mathcal{SM}_3). Let $S^R = (N_S, E_S, \phi, \varphi)$ be an RDF schema and $S^P = (\mathbb{N}_S, \mathbb{E}_S, \mathbb{P}_S, \Theta, \Pi, \Phi, \Psi)$ be a PG schema. The schema mapping $\mathcal{SM}_3(S^R) = S^P$ is defined as follows:

1. For each $rc \in N_S$ satisfying that $\phi(rc) \notin I_{DT}$
 - There will be $nt \in \mathbb{N}_S$ with $\Theta(nt) = \phi(rc)$
2. For each $pc \in E_S$ satisfying that $\varphi(pc) = (rc_1, rc_2)$
 - If $\phi(rc_2) \in I_{DT}$ then
 - There will be $pt \in \mathbb{P}_S$ with $\Pi(pt) = (\phi(pc), f(\phi(rc_2)))$, $\Psi(nt) = \Psi(nt) \cup pt$ where $nt \in \mathbb{N}_S$ corresponds to $rc_1 \in N_S$.
 - If $\phi(rc_2) \notin I_{DT}$ then
 - There will be $et \in \mathbb{E}_S$ with $\Theta(et) = \phi(pc)$, $\Phi(et) = (nt_1, nt_2)$ where $nt_1, nt_2 \in \mathbb{N}_S$ correspond to $rc_1, rc_2 \in N_S$ respectively.

Hence, the schema mapping \mathcal{SM}_3 creates a node type for each resource type (with exception of RDF data types);, creates a property type for each object property, and creates an edge type for each value property.

Assume that the function f is defined by the following datatype assignments: $f(\text{xsd:string}) = \text{String}$, $f(\text{xsd:int}) = \text{Integer}$ and $f(\text{xsd:date}) = \text{Date}$. Hence, the PG schema obtained from the RDF graph schema shown in Figure 4.2 is shown in listing 4.3.3 below:

```

1  $\mathbb{N}_S = \{n_1, n_2, n_3, n_4\}$ ,
2  $\mathbb{E}_S = \{e_1, e_2, e_3, e_4\}$ ,
3  $\mathbb{P}_S = \{p_5, p_6, p_7, p_8, p_9, p_{10}\}$ ,
4  $\Theta(n_1) = \text{voc:Organisation}$ ,  $\Theta(n_2) = \text{voc:Person}$ ,  $\Theta(n_3) = \text{voc:City}$ ,  $\Theta(n_4) = \text{voc:Country}$ ,
5  $\Theta(e_1) = \text{voc:ceo}$ ,  $\Theta(e_2) = \text{voc:location}$ ,  $\Theta(e_3) = \text{voc:country}$ ,  $\Theta(e_4) = \text{voc:is\_location\_of}$ ,
6  $\Pi(p_5) = (\text{voc:name}, \text{String})$ ,  $\Pi(p_6) = (\text{voc:creation}, \text{Date})$ ,  $\Pi(p_7) = (\text{voc:birthName}, \text{String})$ ,
    $\Pi(p_8) = (\text{voc:age}, \text{Integer})$ ,  $\Pi(p_9) = (\text{voc:name}, \text{String})$ ,  $\Pi(p_{10}) = (\text{voc:name}, \text{String})$ ,
7  $\Phi(e_1) = (n_1, n_2)$ ,  $\Phi(e_2) = (n_1, n_3)$ ,  $\Phi(e_3) = (n_3, n_4)$ ,
8  $\Phi(e_4) = (n_4, n_1)$ ,
9  $\Psi(n_1) = \{p_5, p_6\}$ ,  $\Psi(n_2) = \{p_7, p_8\}$ ,  $\Psi(n_3) = \{p_9\}$ ,  $\Psi(n_4) = \{p_{10}\}$ .
    
```

Instance Mapping \mathcal{IM}_3

Now, we define the instance mapping \mathcal{IM}_3 which takes an RDF graph as input and returns a PG as output.

Definition 15 (Instance Mapping \mathcal{IM}_3). Let $G^R = (N_R, N_L, E_O, E_D, \alpha_R, \alpha_L, \beta_O, \beta_D, \delta)$ be an RDF graph and $G^P = (\mathbb{N}, \mathbb{E}, \mathbb{P}, \Gamma, \Upsilon, \Sigma, \Delta)$ be a PG. The instance mapping $\mathcal{IM}_3(G^R) = G^P$ is defined as follows:

1. For each $r \in N_R$
 - There will be $n \in \mathbb{N}$ with $\Gamma(n) = \delta(r)$
 - There will be $p \in \mathbb{P}$
 - If $\alpha_R(r) \in I$ then $\Upsilon(p) = (\text{iri}, \alpha_R(r))$
 - If $\alpha_R(r) \in B$ then $\Upsilon(p) = (\text{id}, \alpha_R(r))$
 - $\Delta(n) = \{p\}$.
2. For each $op \in E_O$ satisfying that $\beta_O(op) = (r_1, r_2)$
 - There will be $e \in \mathbb{E}$ with $\Gamma(e) = \delta(op)$, $\Sigma(e) = (n_1, n_2)$ where $n_1, n_2 \in \mathbb{N}$ correspond to $r_1, r_2 \in N_R$ respectively.
3. For each $dp \in E_D$ satisfying that $\beta_D(dp) = (r_1, r_2)$

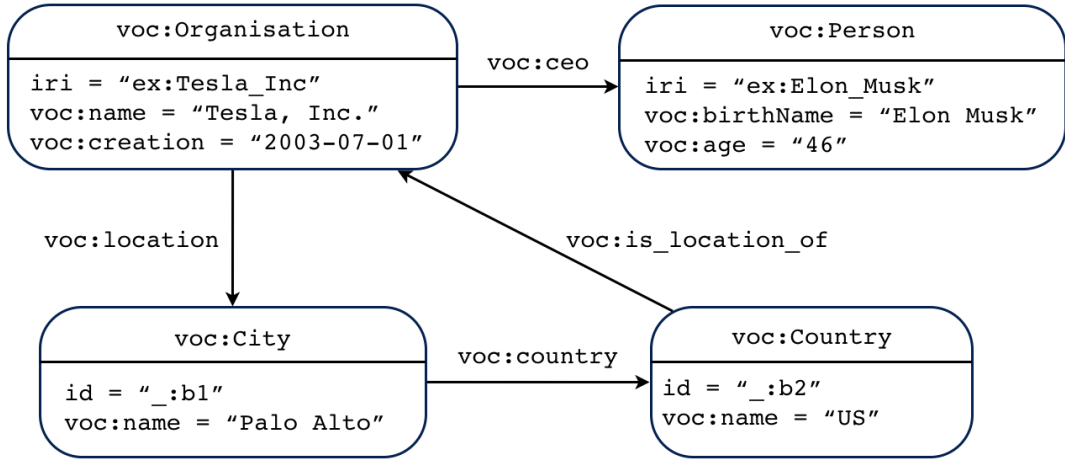


Figure 4.8: Property graph obtained after applying the instance mapping \mathcal{IM}_3 to the RDF graph shown in Figure 4.1.

- There will be $p \in \mathbb{P}$ with $\Upsilon(p) = (\delta(dp), \alpha_L(r_2))$, $\Delta(n) = \Delta(n) \cup \{p\}$ where $n \in \mathbb{N}$ corresponds to $r_1 \in N_R$.

According to the above definition, the instance mapping \mathcal{IM}_3 creates a node in G_R for each resource node, creates a property in G_R for each datatype property, and creates an edge in G_R for each object property.

For example, the PG obtained after applying \mathcal{IM}_3 over the RDF graph shown in Figure 4.1 is shown in the listing 4.3.3 below:

```

1  $\mathbb{N} = \{n_1, n_2, n_3, n_4\}$  ,
2  $\mathbb{E} = \{e_1, e_2, e_3, e_4\}$  ,
3  $\mathbb{P} = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}\}$  ,
4  $\Gamma(n_1) = \text{voc:Organisation}$  ,  $\Gamma(n_2) = \text{voc:Person}$  ,
5  $\Gamma(n_3) = \text{voc:City}$  ,  $\Gamma(n_4) = \text{voc:Country}$  ,
6  $\Gamma(e_1) = \text{voc:ceo}$  ,  $\Gamma(e_2) = \text{voc:location}$  ,  $\Gamma(e_3) = \text{voc:country}$  ,  $\Gamma(e_4) = \text{voc:is\_location\_of}$  ,
7  $\Upsilon(p_1) = (\text{iri}, \text{"ex:Tesla\_Inc"})$  ,  $\Upsilon(p_2) = (\text{iri}, \text{"ex:Elon\_Musk"})$  ,  $\Upsilon(p_3) = (\text{id}, \text{"_:b1"})$  ,
    $\Upsilon(p_4) = (\text{id}, \text{"_:b2"})$  ,  $\Upsilon(p_5) = (\text{voc:name}, \text{"Tesla, Inc."})$  ,
    $\Upsilon(p_6) = (\text{voc:creation}, \text{"2003-07-01"})$  ,  $\Upsilon(p_7) = (\text{voc:birthName}, \text{"Elon Musk"})$  ,
    $\Upsilon(p_8) = (\text{voc:age}, \text{"46"})$  ,  $\Upsilon(p_9) = (\text{voc:name}, \text{"Palo Alto"})$  ,  $\Upsilon(p_{10}) = (\text{voc:name}, \text{"US"})$ 
8  $\Sigma(e_1) = \{n_1, n_2\}$  ,  $\Sigma(e_2) = \{n_1, n_3\}$  ,  $\Sigma(e_3) = \{n_3, n_4\}$  ,  $\Sigma(e_4) = \{n_4, n_1\}$  ,
9  $\Delta(n_1) = \{p_1, p_5, p_6\}$  ,  $\Delta(n_2) = \{p_2, p_7, p_8\}$  ,  $\Delta(n_3) = \{p_3, p_9\}$  ,  $\Delta(n_4) = \{p_4, p_{10}\}$  .
    
```

Figure 4.8 shows a graphical representation of the PG described above.

Properties of \mathcal{DM}_3

In this section we will evaluate the properties of the database mapping \mathcal{DM}_3 . Recall that \mathcal{DM}_3 is formed by the schema mapping \mathcal{SM}_3 and the instance mapping \mathcal{IM}_3 .

Proposition 5. The database mapping \mathcal{DM}_3 is computable.

It is straightforward to see that Definition 14 and Definition 15 can be transformed into algorithms to compute \mathcal{SM}_3 and \mathcal{IM}_3 respectively.

Lemma 2. The database mapping \mathcal{DM}_3 is *semantics preserving*.

Note that the schema mapping \mathcal{SM}_3 and the instance mapping \mathcal{IM}_3 have been designed to create a PG database that maintains the restrictions defined by the source RDF database. On one side, the schema mapping \mathcal{SM}_3 allows to transform the structural and semantic restrictions from the RDF graph schema to the PG schema. On the other side, any PG generated by the instance mapping \mathcal{IM}_3 will be valid with respect to the generated PG schema.

The semantics preservation property of \mathcal{DM}_3 is supported by the following facts:

- We provide a procedure to create a complete RDF graph schema S^R from a set of RDF triples describing an RDF schema, i.e. each property defines its domain and range resource classes.
- We provide a procedure to create an RDF graph G^R from a set of RDF triples, satisfying that every node and edge in G^R is associated with a resource class; it allows a complete connection between the RDF instance and the RDF schema.
- The schema mapping \mathcal{SM}_3 creates a node type for each user-defined resource type, a property type for each datatype property, and an edge for each object property.
- Similarly, the instance mapping \mathcal{IM}_3 creates a node for each resource, a property for each resource-literal edge, and an edge for each resource-resource edge.

Theorem 2. The database mapping \mathcal{DM}_3 is *information preserving*.

In order to prove that \mathcal{DM}_3 is information preserving, we will define a database mapping $\mathcal{DM}_3^{-1} = (\mathcal{SM}_3^{-1}, \mathcal{IM}_3^{-1})$ which allows to transform a PG database into an RDF database. The inverse mapping \mathcal{DM}_3^{-1} must satisfy that $D = \mathcal{DM}_3^{-1}(\mathcal{DM}_3(D))$ for any RDF database D . Next we define the schema mapping \mathcal{SM}_3^{-1} and the instance mapping \mathcal{IM}_3^{-1} .

Definition 16 (Schema mapping \mathcal{SM}_3^{-1}). Let $S^P = (\mathbb{N}_S, \mathbb{E}_S, \mathbb{P}_S, \Theta, \Pi, \Phi, \Psi)$ be a PG schema and $S^R = (N_S, E_S, \phi, \varphi)$ be an RDF schema. The schema mapping $\mathcal{SM}_3^{-1}(S^R) = S^P$ is defined as follows:

- Let $C = \{n \mid n \in \text{range}(\Theta)\} \cup \{n = f^{-1}(t) \mid (u, t) \in \text{range}(\Pi)\}$
- Let $\omega : C \rightarrow N_S$ be a function that maps IRIs to resource classes
- For each $n \in C$
 - There will be $rc \in N_S$ with $\phi(rc) = n$

- $\omega(n) = rc$
- For each $et \in \mathbb{E}_S$ with $\Phi(et) = (nt_1, nt_2)$
 - There will be $pc \in E_S$ with $\phi(pc) = \Theta(et)$ and $\varphi(pc) = (\omega(nt_1), \omega(nt_2))$
- For each $nt \in \mathbb{N}_S$
 - For each $pt \in \Psi(nt)$ with $\Phi(pt) = (n, t)$
 - * There will be $pc \in E_S$ with $\phi(pc) = n$ and $\varphi(pc) = (\omega(nt), \omega(f^{-1}(t)))$

In general terms, the schema mapping \mathcal{SM}_3^{-1} creates a resource class for each node type, an object property for each edge type, and a datatype property for each property type. Given a PG schema $S^P = \mathcal{SM}_3(S^R)$, the schema mapping \mathcal{SM}_3^{-1} allows to “recover” all the schema constraints defined by S^R , i.e $\mathcal{SM}_3^{-1}(S^P) = S^R$.

An issue of \mathcal{SM}_3^{-1} , is the existence of RDF datatypes which are not supported by PG databases. For example, `rdfs:Literal` has no equivalent datatype in PG database systems. The solution to this issue is to find a one-to-one correspondence between RDF datatypes and PG datatypes.

Definition 17 (Instance mapping \mathcal{IM}_3^{-1}). Let $G^P = (\mathbb{N}, \mathbb{E}, \mathbb{P}, \Gamma, \Upsilon, \Sigma, \Delta)$ be a property graph and $G^R = (N_R, N_L, E_O, E_D, \alpha_R, \alpha_L, \beta_O, \beta_D, \delta)$ be an RDF graph. The instance mapping $\mathcal{IM}_3^{-1}(G^P) = G^R$ is defined as follows:

1. For each $n \in \mathbb{N}$, there will be $r \in N_R$ where
 - a) $\alpha_R(r) = v$ such that $p \in \Delta(n)$ and $\Upsilon(p) = (\text{iri}, v)$ or $\Upsilon(p) = (\text{id}, v)$
 - b) $\delta(r) = \Gamma(n)$
 - c) For each $p \in \Delta(n)$ satisfying that $\Upsilon(p) = (\text{lab}, \text{val})$ and $\text{lab} \notin \{\text{iri}, \text{id}\}$, there will be $l \in N_L$ and $dp \in E_D$ with $\alpha_L(l) = \text{val}$, $\delta(l) = f^{-1}(\text{type}(\text{val}))$, $\delta(dp) = \text{lab}$ and $\beta(dp) = (r, l)$
2. For each $e \in \mathbb{E}$ where $\Sigma(e) = (n_1, n_2)$, there will be $op \in E_O$ with $\delta(op) = \Gamma(e)$ and $\beta(op) = (r_1, r_2)$ such that r_1, r_2 correspond to n_1, n_2 respectively.

Hence, the above method defines that each node in G^P is transformed into a resource node in G^R , each property in G^P is transformed into a datatype property in G^R , and each edge in G^P is transformed into an object property in G^R . Given a PG $G^P = \mathcal{IM}_3(G^R)$, the instance mapping \mathcal{IM}_3^{-1} allows to “recover” all the data in G^R , i.e $\mathcal{IM}_3^{-1}(G^P) = G^R$.

Note that each RDF graph produced by the instance mapping \mathcal{IM}_3^{-1} will be valid with respect to the schema produced with the corresponding schema mapping \mathcal{SM}_3^{-1} . Hence, any RDF database D^R can be transformed into a PG database by using the database mapping $\mathcal{DM}(D^R)$, and D^R could be recovered by using the database mapping \mathcal{DM}_3^{-1} .

4.4 Experimental Evaluation

The objective of this experimental evaluation is to examine the performance and scalability of the database mappings presented in this work. This section includes a description of the implementation, the evaluation methodology, the experimental results, and the corresponding discussion.

4.4.1 Implementation

We have developed a java application called `rdf2pg` which implements the mappings described in this article. The source code and the executable jar file of `rdf2pg` can be downloaded from Github (<https://github.com/renzoar/rdf2pg>). The tool can be executed in command line by using an expression with the structure

```
java -jar rdf2pg.jar <m> <i> <s>
```

where `<m>` indicates the database mapping (`-sdm` = simple database mapping, `-gdm` = generic database mapping, `-cdm` = complete database mapping), `<i>` indicates the input instance RDF graph file, and `<s>` indicates the input RDF schema file (in case of using `-gdm` or `-cdm`).

The output of the simple database mapping is a file encoding a PG. In addition, the generic and the complete instance mappings produce a second file containing the PG schema. The current implementation uses the YARS-PG [184] data format for both output files.

The `rdf2pg` API includes an interface named `PGWriter` which can be implemented to support other data formats. The use of `PGWriter` is very simple as it provides the methods `WriteNode(PGNode node)` and `WriteEdge(PGEdge edge)` which should be implemented with the corresponding instructions to write nodes and edges in the output data format.

In order to support the processing of large RDF data files, `rdf2pg` uses the `StreamRDF` class provided by Apache Jena. Additionally, `rdf2pg` implements two methods for writing the output file: a memory-based method which creates a PG object (which follows the definition presented in Section 4.2); and a disk-based method which writes the output by using a minimal set of structures.

Additionally, we have developed a Java application called `rdfs-processor` which provides three functionalities: analysis of an RDF Schema file to obtain basic information (i.e. the number of resource classes, number of property classes, and number of datatypes); normalization of an RDF Schema, in the case of incomplete definitions (e.g. empty domains); and schema discovery from an RDF data file.

The functionality of schema discovery is very relevant for this paper because most of the available RDF datasets do not provide an RDF Schema file. Our method for schema discovery follows the approach described in [185]. In general terms, the method reads the set of RDF triples two times: in the first pass, it identifies resource classes and property classes; in the second pass, it determines the domain and range for each property class. The output is an RDF file

Dataset	Type	Nature	Structure
SP2B [187]	Bibliographic	Synthetic	Regular
Linked GeoData [188]	Spatial	Real	Regular
WatDiv [166]	E-commerce	Synthetic	Irregular
BSBM [168]	E-commerce	Synthetic	Regular
Wikidata [189]	Open Knowledge Base	Real	Irregular

Table 4.1: Datasets used in the experimental evaluation.

Graph ID	Dataset	#Triples	File Size (*.nt)
G1	SP2B	328	42 KB
G2	SP2B	1,285	206 KB
G3	SP2B	10,303	1.6 MB
G4	SP2B	100,073	16.2 MB
G5	SP2B	1,000,009	165 MB
G6	GeoData	4,914,217	740.7 MB
G7	WatDiv	7,159,355	1.01 GB
G8	BSBM	38,333,972	10.01 GB
G9	Wikidata	41,191,235	6.27 GB

Table 4.2: RDF Graphs used in the experimental evaluation.

containing a basic description of the RDF Schema by means of the terms `rdf:type`, `rdfs:Class`, `rdf:Property`, `rdfs:domain` and `rdfs:range`. The source code of the `rdfs-processor` is available in Github (<https://github.com/renzoar/rdfs-processor>).

4.4.2 Methodology and Experimental Setup

The experimental evaluation consists of a series of experiments that combine three variables: database mapping, data source, RDF graph size, and processing power. We evaluate the three database mappings defined in this paper: simple mapping, generic mapping, and complete mapping. We consider four sources of RDF data whose characteristics (domain, nature, and structure) are shown in Table 4.1. We use nine RDF graphs (obtained from the data sources) whose size³ goes from 328 triples to 41,191,235 triples, as shown in Table 4.2.

The processing power variable indicates the use of machines with different characteristics in terms of hardware. In this case, we used four virtual machines hosted in the Google Cloud Platform, having a varying number of CPUs (Intel Skylake), main/primary memory size (RAM), and secondary memory size (SSD). The technical specification of each machine is shown in Table 4.3. All the machines worked with Debian GNU/Linux 9 (amd64 built on 20200309) as the operating system and Java OpenJDK 1.8.0_242 (64-Bit) without a graphic environment.

Based on the above variables, we evaluated the database mappings in terms of performance and scalability. The performance is measured as the running time (or runtime) required to execute a

³ The size of an RDF graph is expressed in terms of the number of triples. Note that the disk space occupied by a set of triples depends on the RDF data format used [186].

Machine ID	Type	#vCPUs	RAM	SSD
VM1	n1-standard-2	2	7.5GB	100GB
VM2	n1-standard-4	4	15GB	100GB
VM3	n1-standard-8	8	30GB	100GB
VM4	n1-standard-16	16	60GB	100GB

Table 4.3: Virtual Machines (Google Cloud Platform) used in the experimental evaluation.

Graph ID	#Classes	#Properties	#Datatypes
G1	5	19	2
G2	6	20	2
G3	8	54	2
G4	9	55	2
G5	12	64	2
G6	393	60	3
G7	40	85	1
G8	1292	36	3
G9	24	44	2

Table 4.4: RDF Schemas used in the experimental evaluation. This table shows the number of resource classes, property classes, and datatype definitions.

mapping and construct the corresponding output database (schema graph and instance graph). To do this, the `rdf2pg` application uses the built-in Java function `System.currentTimeMillis` to register the runtime. The objective is to determine the computational complexity of the mappings in practice.

Each mapping is evaluated under two notions of scalability. Former, we measure the scalability with respect to the size of the input data (i.e. the number of triples). The objective is to determine the behavior of the mappings with RDF graphs of different sizes. Later, we analyze the scalability with respect to the computational resources. The objective is to determine the dependency of each mapping with respect to the hardware.

4.4.3 Experimental Results

Our experimental evaluation begins with the extraction of the RDF Schema for each RDF graph. This task was performed by using the `rdfs-processor` tool described in Section 4.4.1. Table 4.4 shows information about the corresponding RDF Schemas. We can observe that: the SP2B graphs do not change too much in terms of the number of classes and properties; the number of classes in GeoData and BSBM is larger than the number of properties; a small number of datatypes are defined in the graphs.

Once we had the RDF Schema files for each dataset, we executed the experiments in the virtual machines. Every execution of the `rdf2pg` application was configured to use the maximum amount of primary memory allowed by the machine. To do this, we use the `-Xmx` parameter defined by Java. Table 4.5, Table 4.6 and Table 4.7 show the runtimes for the simple mapping, the generic

Graph	VM1	VM2	VM3	VM4
G1	3544	1247	1226	1263
G2	1644	1153	1057	1072
G3	2203	1516	1282	1364
G4	4245	3154	2428	2524
G5	16714	14133	11578	12079
G6	89930	69582	66290	67458
G7	193389	156963	148793	149272
G8	?	?	782718	741916
G9	?	593914	616252	625932

Table 4.5: Runtimes (in milliseconds) for the **simple data mapping**. Undefined runtimes are represented with “?”.

Graph	VM1	VM2	VM3	VM4
G1	1359	1015	960	1099
G2	1642	1101	1030	1093
G3	1953	1411	1218	1260
G4	3545	2564	1904	2170
G5	14636	10772	8566	8483
G6	68390	50471	49977	47064
G7	96449	77464	81579	83639
G8	?	550476	486106	487969
G9	?	413090	427542	381572

Table 4.6: Runtimes (in milliseconds) for the **generic data mapping**. Undefined runtimes are represented with “?”.

mapping and the complete mapping respectively.

In general terms, we can observe that the mappings worked well with most of the graphs (i.e. G1 to G7). However, there were problems to complete the task for graphs G8 and G9, running on virtual machines VM1 and VM2. Specifically, the execution of `rdf2pg` produced an error of “insufficient memory for the Java Runtime Environment”. Hence, our current implementation has a restriction to process large input graphs with small-memory machines.

The above problem is related to the main memory (RAM) required to manage the intermediate objects used by the mappings. Being more specific, the mappings create a `HashMap` to store all the nodes and their properties, and such a structure could be very large for some graphs. Note that the number of nodes is not directly related to the number of triples. For example, we observed that the number of nodes generated for G8 was higher than G9, even when G9 has more triples than G8. It explains why the simple mapping was able to process G9, but it was not able to process G8, both using VM2.

In order to analyze the scalability of the mappings with respect to the size of the input data, we selected the runtimes obtained with VM4. As shown in Figure 4.9, the execution time of all the mappings grows up in concordance with the size of the input graphs, i.e., the larger the size of the graph, the larger the runtime. Note also that the runtimes of the mappings are under

Graph	VM1	VM2	VM3	VM4
G1	1659	1083	1004	1062
G2	1642	1194	1111	1117
G3	1984	1515	1318	1403
G4	4269	2920	2442	2505
G5	20685	1524	12629	12737
G6	97101	78374	74898	69639
G7	200952	173607	161105	145027
G8	?	?	817471	800303
G9	?	?	609577	603040

Table 4.7: Runtimes (in milliseconds) for the **complete data mapping**. Undefined runtimes are represented with “?”.

Graph	SDM	GDM		CDM	
	PG	PG	PGS	PG	PGS
G1	12K	39K	352	15K	553
G2	43K	190K	352	61K	710
G3	320K	1.5M	352	466K	2.3K
G4	3.1M	15M	352	4.5M	3.1K
G5	32M	149M	352	46M	4.0K
G6	125M	731M	352	214M	22K
G7	224M	711M	352	276M	19K
G8	2.6G	7.7G	352	3.3G	872K
G9	1.1G	5.0G	352	1.7G	2.8K

Table 4.8: Size (in bytes) of the output files produced during the experimental evaluation of the database mappings (SDM, GDM and CDM). PG and PGS mean property graph and property graph schema respectively.

the baseline defined by the graph sizes. Hence, we can conclude that the complexity of the mappings is linear with respect to the size of the input.

In order to analyze the scalability of the mappings with respect to the computational power, we prepare a plot for each mapping showing the runtimes for all the virtual machines (see Figures 4.10, 4.11 and 4.12). The plots show that the runtimes decrease for VM1, VM2, and VM3; however, the runtimes for VM3 and VM4 are not so different. The latter implies that there is a threshold in which the computational power does not reduce the execution time of the mapping.

As a general conclusion, we can say that the three database mappings presented in this work have an efficient implementation to process large datasets and work under mid-size computational resources. All the input and output files described in the above experiments are available in Figshare [37].

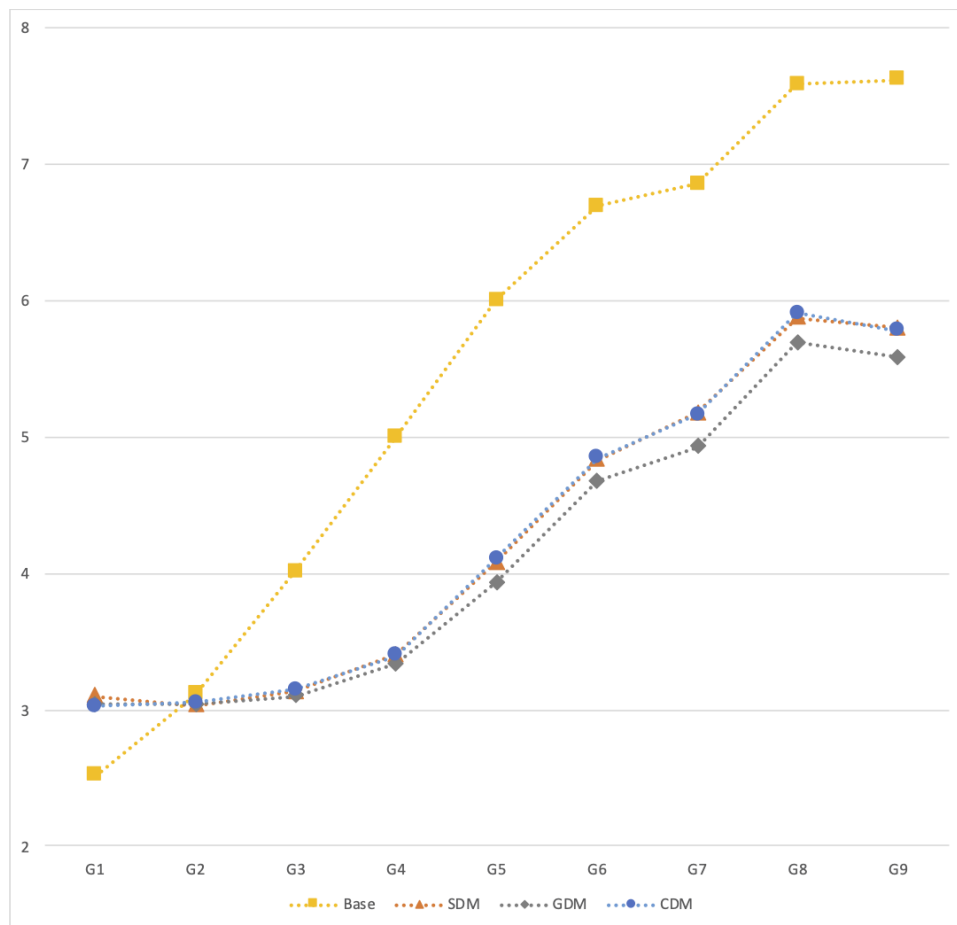


Figure 4.9: Scalability of the database mappings with respect to the size of the input data. The X-axis has graph sizes and the Y-axis has runtimes (in log scale). The “Base” line indicates the sizes of the input graphs in the log scale.

4.4.4 Interoperability in Practice

In order to show the practical use of the database mappings proposed in this article, we conducted a complete ETL process that involves: Extracting RDF data from an RDF dataset; Transforming the extracted RDF data to PG data, and Loading the transformed data into a property graph database system. Due to its popularity and availability of features for data loading, we selected Neo4j as the target database system.

The main issue in this experiment is the configuration of `rdf2pg` to generate and encode property graphs into a data format that can be consumed by the Neo4j system. To do this, we created the `Neo4jWriter` class as an implementation of the `PGWriter` interface provided by `rdf2pg`. The `Neo4jWriter` class allows exporting a property graph as a set of Cypher instructions to create nodes and edges. For example, the property graph shown in Figure 4.5 will be exported as follows:

```

1 CREATE (n15:City {name:'Palo Alto'})
2 CREATE (n12:Person {birthName:'Elon Musk', age:'46'})

```

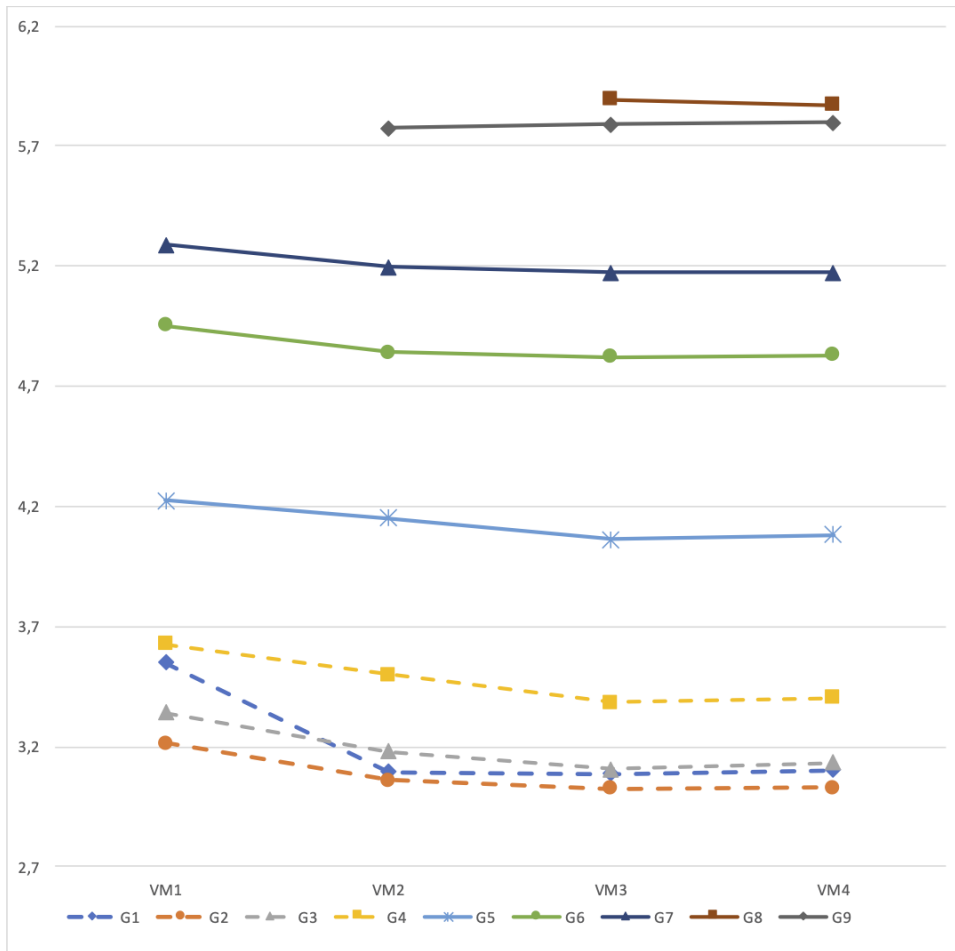


Figure 4.10: Scalability of the simple database mapping with respect to the hardware. The X-axis has virtual machines and the Y-axis has runtimes (in log scale).

```

3 CREATE (n33:Country {name:'US'})
4 CREATE (n1:Organisation {name:'Tesla , Inc.',creation:'2003 07 01'})
5 CREATE (n1) [e5:ceo] >(n12)
6 CREATE (n1) [e6:location] >(n15)
7 CREATE (n15) [e7:country] >(n33)
8 CREATE (n33) [e8:is_location_of] >(n1)

```

To demonstrate the validity of our mappings, we used the property graph obtained by applying the simple database mapping over the RDF graph G2, i.e. the SP2B file containing 1,285 triples. The output file containing Cypher instructions was loaded in Neo4j Desktop 1.2.3 by using the browser-based user interface. The loading process took 7 ms, resulting in a property graph with 270 nodes, 348 relationships, 677 properties, and 260 labels. A graphical representation of the loaded property graph, obtained from the Neo4j browser, is shown in Figure 4.13.

The above experiment was repeated for the generic and the complete database mappings. The generic mapping produced, after 55 ms, a property graph containing 949 nodes, 1285

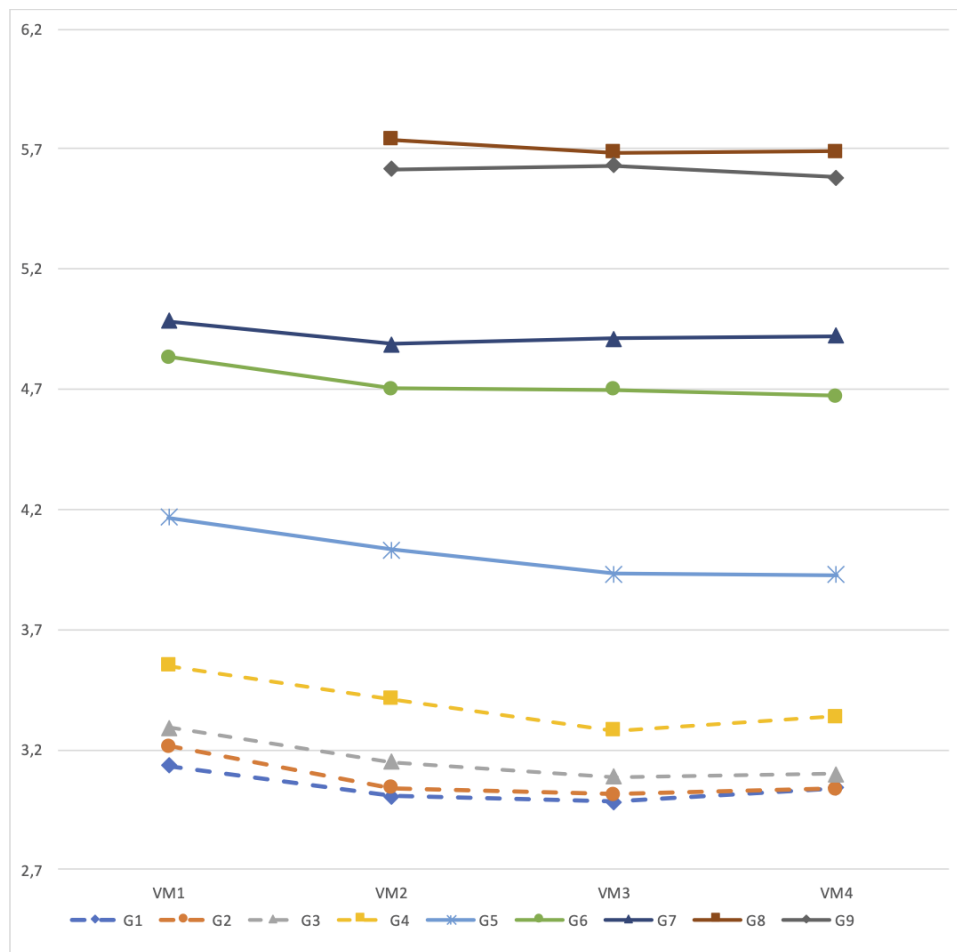


Figure 4.11: Scalability of the generic database mapping with respect to the hardware. The X-axis has virtual machines and the Y-axis has runtimes (in log scale).

relationships, 2911 properties, and 949 labels. The complete mapping took 8 ms, producing a property graph with the same number of elements produced by the simple mapping.

All the information related to this data loading experiment, including the output files and the charts of the property graphs, are available in Figshare [37].

4.4.5 Limitations

The limitations of the mappings presented in this paper, we highlight the following: the simple mapping is not suitable for RDF datasets with complex vocabularies as the common names will be merged in the resulting property graph; the generic mapping works with any RDF dataset, but the size of the output property graph will be bigger than the other two mappings; the complete mapping is suitable for any RDF dataset, but in practice, it requires a special directory to map prefixes to namespaces. A general limitation of the three mappings is that they are not able to deal with the special semantics defined by the RDF model (e.g. reification) and the

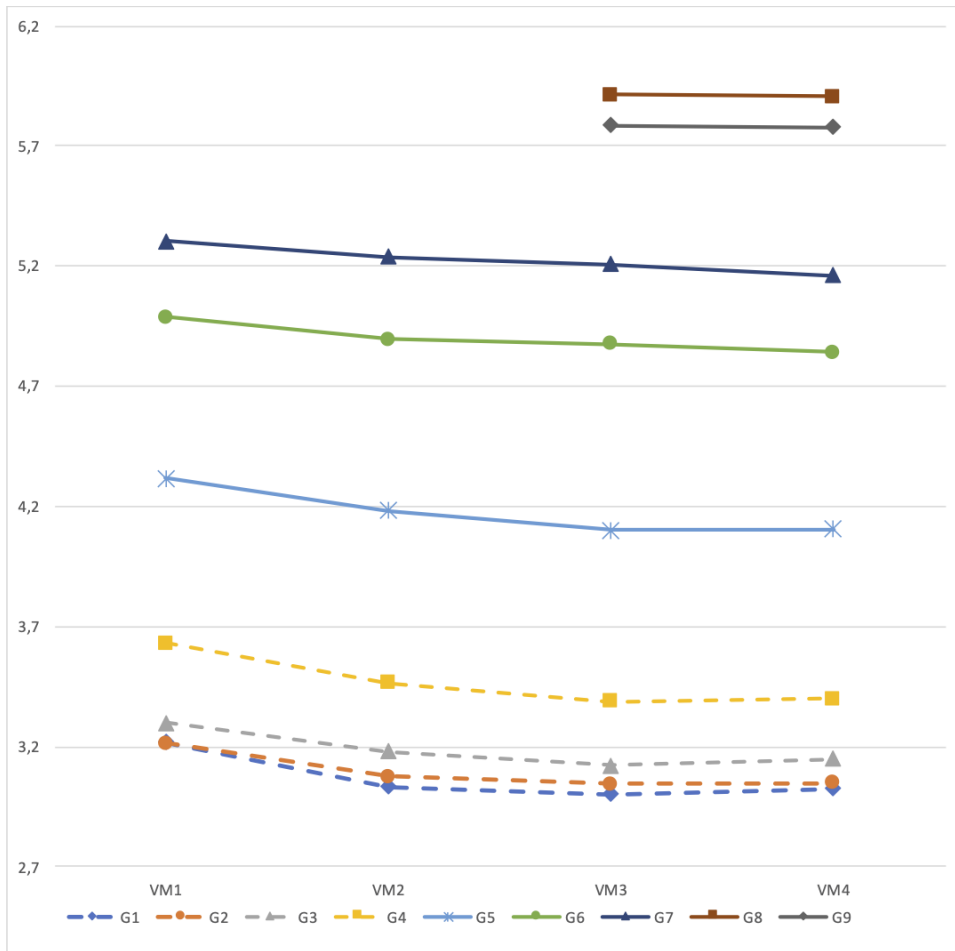


Figure 4.12: Scalability of the complete database mapping with respect to the hardware. The X-axis has virtual machines and the Y-axis has runtimes (in log scale).

inference rules supported by RDF Schema (e.g. sub-class, sub-property). We plan to study these features in the future.

4.5 Summary

In this chapter, we have proposed a novel approach, which consists of three direct mappings, to transform RDF databases into PG databases. We demonstrate, empirically and formally, that the mappings have an efficient implementation to process large datasets. We showed that two of the proposed mappings satisfy the property of information preservation, i.e. there exist inverse mappings that allow recovering the original databases without losing information. These results allow us to present the following conclusion about the information capacity of the PG data model with respect to the RDF data model.

Corollary 1. The property graph data model subsumes the information capacity of the RDF

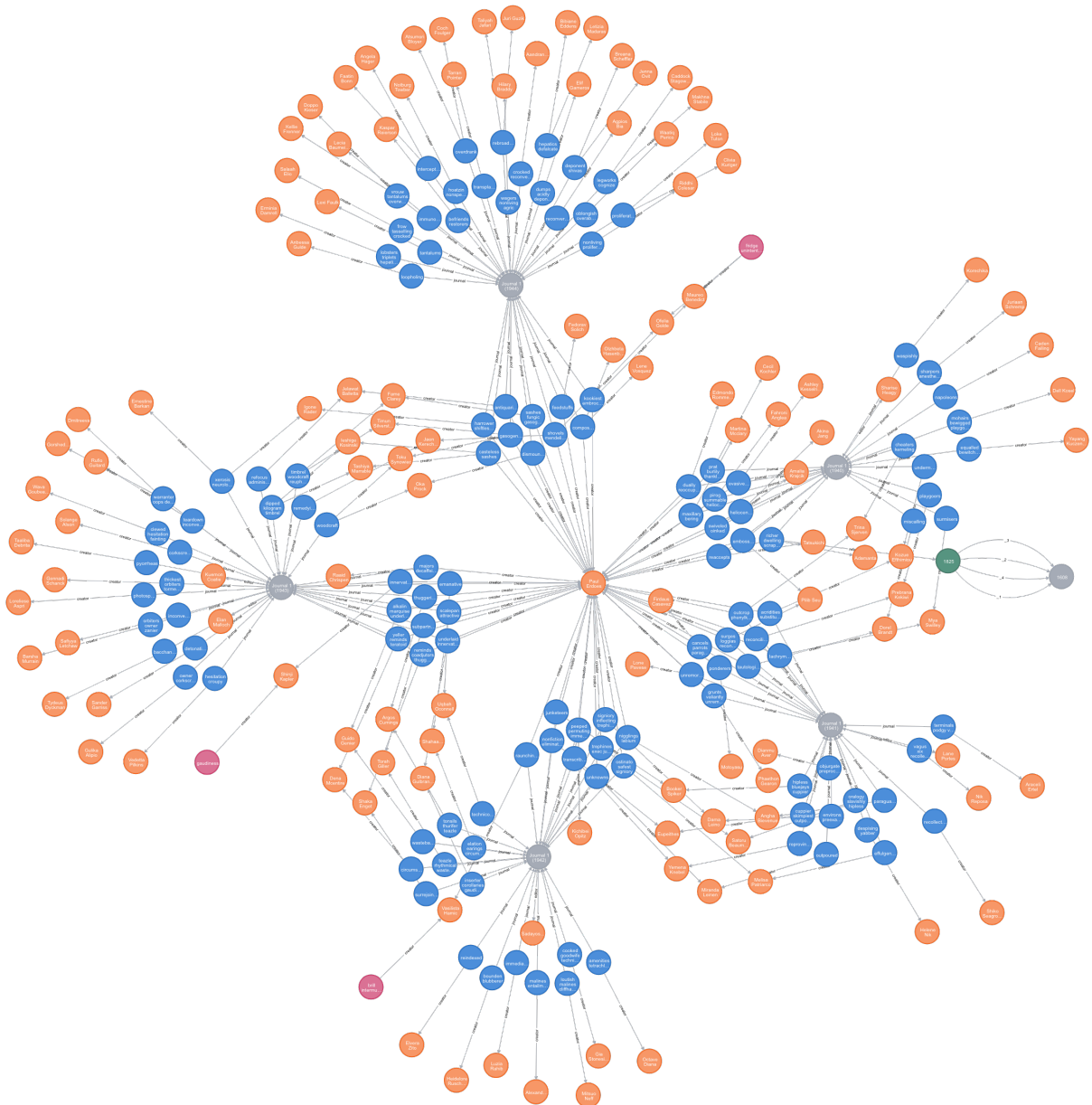


Figure 4.13: Graphical representation of the property graph produced by applying the simple database mapping over the RDF graph G2.

data model.

Although our methods assume some condition for the input RDF databases, they are generic and can be easily extended (by overloading the mapping functions) to provide support for features such as inheritance and reification. Furthermore, our formal definitions will be very useful to study query interoperability [41–43] and query preservation between RDF and PG databases (i.e. transformations among SPARQL and PG query languages). Thus, with this contribution, we take a substantial step by laying the core formal foundation for supporting interoperability

between RDF and PG databases.

As future work, we plan to incorporate features such as RDF reification techniques (n-ary relationships, named graphs, etc), inheritance, and OWL 2 RL. We would also like to further examine the impact of each reification technique in terms of mapping complexity, time, PG data volume, and other parameters as future work.

GREMLINATOR: Querying Property graph Databases using SPARQL

Knowledge graphs have become popular over the past years and frequently rely on the Resource Description Framework (RDF) or Property Graphs (PG) as underlying data models. However, the query languages for these two data models – SPARQL for RDF and Gremlin for property graphs – lack interoperability. While there exist some approaches that study the issue of *query interoperability*, most of the work is focused on addressing RDF and relational database query languages – SPARQL and SQL. As compared to these efforts we have to overcome the challenge of mediating between two very different execution paradigms. More specifically, those efforts applied query rewriting techniques between languages, which are rooted in relational algebra operations, whereas we had to bridge more disparate query paradigms. While this poses a significant challenge, it is also the reason why substantial performance differences can be observed depending on the different query characteristics.

In this chapter we address the following second research question (RQ2) that is concerned with addressing *query interoperability* issue between the RDF and Property graph databases.

RQ2: Query Interoperability – How can we execute SPARQL queries over Property graphs in a query preserving manner?

The main contributions of this chapter are the: (i) formalisation of the declarative construct of the Gremlin traversal language using a consolidated graph relational algebra, and (ii) GREMLINATOR which a novel approach that enables the execution of W3C standard SPARQL queries over Property graph databases by translating them into pattern matching Gremlin traversals. GREMLINATOR is the first approach that allows users to access and query a wide variety of Graph databases avoiding the necessity of learning a new Graph Query Language. Gremlin is a system agnostic traversal language covering both OLTP graph databases and OLAP graph processors, thus ideal for supporting query interoperability for Graph databases. We present a comprehensive empirical evaluation and demonstrate the applicability of `sparql-gremlin` with leading graph stores – Neo4j, Sparksee and Apache TinkerGraph and compare the performance with the leading RDF stores – Virtuoso, 4Store and JenaTDB. Our experimental evaluation

reveals a substantial performance gain when leveraging the Gremlin counterparts of the SPARQL queries, particularly for star-shaped and complex queries. Currently, GREMLINATOR is available as a plugin for the Apache TinkerPop graph computing framework (as the `sparql-gremlin` plugin).

This chapter is based on the following publications [34, 40–42, 190]:

1. **Harsh Thakkar**, Renzo Angles, Marko Rodriguez, Stephen Mallette, and Jens Lehmann. *Let's build Bridges, not Walls: SPARQL Querying of TinkerPop Graph Databases with Sparql-Gremlin*. In Proceedings of the IEEE 14th International Conference on Semantic Computing (ICSC), pp. 408-415, San Diego, USA, 2020. [**Best Paper Award**]
2. Renzo Angles, **Harsh Thakkar**, and Dominik Tomaszuk. *RDF and Property Graphs Interoperability: Status and Issues*. In Proceedings of the 13th Alberto Mendelzon International Workshop on Foundations of Data Management, Asunción (AMW 2019), Paraguay, June 3-7, 2019.
3. Vinh Nguyen, Hong Yung Yip, **Harsh Thakkar**, Qingliang Li, Evan Bolton, and Olivier Bodenreider. *Singleton property graph: Adding a semantic web abstraction layer to graph databases*. In Proceedings of the 2nd International Semantic Web Conference (ISWC) Workshop on Contextualised Knowledge Graphs (CKG), New Zealand, 2019.
4. **Harsh Thakkar**, Dharmen Punjani, Jens Lehmann, and Sörenen Auer. *Two for one: Querying Property Graph Databases using SPARQL via GREMLINATOR*. In Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and NetworkData Analytics (NDA), page 12, ACM, USA, 2018.
5. **Harsh Thakkar**, Dharmen Punjani, Yashwant Keswani, Jens Lehmann, and Sören Auer. *A Stitch in Time Saves Nine – SPARQL Querying of Property Graphs using Gremlin Traversals*. Pre-print arXiv:1801.02911, pp. 1-24, 2018.
6. **Harsh Thakkar**, Dharmen Punjani, Sören Auer, and Maria-Esther Vidal. *Towards an Integrated Graph Algebra for Graph Pattern Matching with Gremlin*. In Proceedings of the 28th International Conference on Database and Expert Systems Applications (DEXA 2017), Lyon, France, pp. 81-91. Springer, 2017.

The remainder of this chapter is structured into eight sections, starting with Section 5.1 discusses the operators of the declarative of the Gremlin traversal language and formalizes them using a consolidate graph relational algebra. Section 5.2 sheds light on the proposed novel query translation approach GREMLINATOR and its architecture. Section 5.3 discusses the implementation methodology of GREMLINATOR and its current limitations. Section 5.4 presents the methodology, design and execution a comprehensive experimental evaluation of the proposed mapping by executing queries over three top of the line RDF and Property graph databases, thereby demonstrating its applicability. Section 5.5 presents and discusses the findings of our conducted exhaustive experiments and provides empirical evidence that the proposed approach is query preserving. Section 5.6 discusses the Apache TinkerPop integration of GREMLINATOR and lends pointers to its pen-source resources. Section 5.7 discusses the academic and industrial

use cases of GREMLINATOR thereby providing proof of its credibility and adoption by a larger audience. Finally, Section 5.8 concludes this chapter and points to the direction ahead.

5.1 Graph Relational Operators of Gremlin Traversal Language

In this section, we contribute to establishing a formal base for a graph query algebra, by surveying and integrating existing graph query operators [60, 63, 70, 76, 191, 192], with respect to the Gremlin traversal language. Moreover, we formalize the graph pattern matching construct of the Gremlin query language. Lastly, we provide a formal specification of pattern matching traversals for the Gremlin language, which can serve as a foundation for implementing a Gremlin based query compilation engine.

As a result, the formalization of graph query algebra supports the integration and interoperability of different graph data models [72] (e.g., executing SPARQL queries on top of Gremlin [41, 42]), helps to prevent vendor lock in scenarios and boosts data management benchmarking efforts such as LITMUS [48, 193, 194].

5.1.1 Defining Gremlin Operators

We now consolidate various unary, binary, extended and graph-specific graph query operators from the existing literature and define them in the context the Gremlin traversal language.

Unary operators

Projection $(\pi_{a,b,\dots}) : R \cup S \rightarrow \Sigma^*$: operator projects values of a specific set of variables a, b, \dots, n (i.e. keys and elements), from the solution of a matched input graph pattern P , against the graph G . Moreover, the results returned by $(\pi_{a,b})$ are not deduplicated by default, i.e. the result will contain as many possible matched values or items as the input pattern P . This operator is present in all standard graph query languages (e.g. **SELECT** in SPARQL, and **MATCH** in CYPHER).

Selection $(\exists(p))$, is analogous to the filter operator (σ) , as defined in [13, 192], restricts the match of a certain graph pattern P against a graph G , by imposing conditional expressions (p) e.g., inequalities and/or other traversal-specific predicates (where predicate is a proposition formula).

Binary operators

Concatenation [76] $(\circ) : E^* \times E^* \rightarrow E^*$: concatenates two paths (cf. Definition 19). For instance, if (i, α, j) and (j, β, k) are two edges in a graph G , then their concatenation is the new

path (i, α, j, β, k) ; where $i, j, k \in V$ and $\alpha, \beta \in L_e$.

Union operator (\uplus): $P(E^*) \times P(E^*) \rightarrow P(E^*)$: is the multiset union¹ (bag union) of two path traversals or graph patterns. For instance, $\{(1,2), (3,4), (3,4), (4,5)\} \uplus \{(1,2), (3,4)\} = \{(1,2), (1,2), (3,4), (3,4), (3,4), (4,5)\}$. The results of this operator, like projection, are not deduplicated by default.

Join (\bowtie_{\circ}): $P(E^*) \times P(E^*) \rightarrow P(E^*)$: produces the concatenative join of two sets of paths (path traversals [76]) such that if $P, R \in P(E^*)$, then

$$P \bowtie_{\circ} R = \{p \circ r \mid p \in P \wedge r \in R \wedge (p = \epsilon \vee r = \epsilon \vee \gamma^+(p) = \gamma^-(r))\}^2$$

For instance, if $P = \{(v_1, e_1, v_2), (v_2, e_2, v_3)\}$ and $R = \{(v_2, e_2, v_3), (v_2, e_2, v_1)\}$, then,

$$P \bowtie_{\circ} R = \{(v_1, e_1, v_2, v_2, e_2, v_3), (v_1, e_1, v_2, v_2, e_2, v_1)\},$$

where $v_1, v_2, v_3 \in V; e_1, e_2 \in L_e$.

Left-join (\bowtie), *Right-join* (\bowtie) and the *Anti-join* (\triangleright) operators: these operators, are not explicitly implemented in Gremlin, unlike in other graph query languages [70]. Their results can, however, be simulated by the user, at run-time via selecting desired values of elements, vertices or edges declaring using the projection operator. For instance, an anti-join can be "computationally" achieved by `not`'ing an argument in the match step by using `.not()` Gremlin step.

General Extensions.

We borrow the extended relational operators *Grouping* ($\dagger_a(p)$), *Sorting* ($\Re_{\uparrow a, \downarrow b}(p)$) and *Deduplication* ($\delta_{a,b,..}(p)$) which have been defined in [192]. A detailed illustration with formal definitions of extended operators can be found in [2].

Graph-specific Extensions.

Various graph/traversal-specific operators have been defined in works such as [76, 191]. Furthermore, there also exist certain application-specific extensions of algebra operators, such as the α and β operators, for graph data aggregation (used in complex graph network analysis) defined by [3]. We present graph-specific operators, some of which have been adapted from [191, 192] and propose additional operators based on the algebra defined by [70, 76].

¹ Note that the domains and ranges of each of these sets are the power sets.

² Here, (γ^-, γ^+) denote the first and last elements of a path respectively.

Op. Arity	Operation	Operator	Gremlin Step	Step type
0	Get vertices	V_g	<code>g.V()</code>	-
	Get edges	E_g	<code>g.E()</code>	-
1	Selection	$\exists(p)$	<code>.where()</code>	Filter
	Property filter	$\sigma_{condition}^a(p)$	<code>.has()/values()</code>	Filter
	Projection	$\Pi_{a,b,\dots}(p)$	<code>.select()</code>	Map
	De-duplication	$\delta_{a,b,\dots}(p)$	<code>.dedup()</code>	Filter
	Restriction	$\lambda_l^s(p)$	<code>.limit()</code>	Filter
	Sorting	$\mathfrak{R}_{\uparrow a, \downarrow b}(p)$	<code>.order().by()</code>	Map
	Grouping	$\dagger_a(p)$	<code>.group().by()</code>	Map/SideEffect
	Traverse (out/in)	$\uparrow_{v1}^{v2}[e](p)$	<code>.out()/in()</code>	FlatMap
2	Join	$p \bowtie_{\circ} r$	<code>.and()</code>	Filter
	Union	$p \uplus r$	<code>.union()</code>	Branch

Table 5.1: A consolidated list of graph relational algebra operators with their corresponding instruction steps in the Gremlin traversal language.

- The *Get-vertices/Get-edges* null-ary operators (V_g / E_g): return the list of vertices/edges, respectively. These operators, w.r.t. Gremlin query construct, denote the start of a traversal. It is also possible to traverse from a specific vertex/edge in a graph, given their id's. Furthermore, they can be used to construct custom indexes over elements depending on user's choice.
- The *Traverse operator* ($\uparrow_{v1}^{v2}[e](p)$): $P(V \cup E) \times \Sigma^* \rightarrow P(V \cup E)$: is an adapted version, analogues to the *expand-both* operator defined by [191].

The traverse operator represents the traversing over the graph operation (traversing *in* \downarrow or *out* \uparrow from a vertex ($v1$) to an adjacent vertex ($v2$) given the edge label $[e]$, where $(v1, v2) \in V, e \in L_e$).

- The *Property filter* operator ($\sigma_{condition}^{v/e}(p)$): $P(V \cup E) \times S \rightarrow \Sigma^*$: is a binary operator which: (i) filters the values of selected element (vertex/edge), if a *condition* is declared, (ii) otherwise, it simply returns the value of the element's property.
- The *Restriction* unary operator ($\lambda_l^s(p)$) is an adaptation of [195], which we borrow from [192]. It takes a list as input and returns the top s values, skipping specified l values. It is analogous to the LIMIT and OFFSET modifier keyword pair in SPARQL.

5.1.2 Graph Pattern Matching via Traversing

Graph Pattern Matching (GPM for short) is generally perceived as a subgraph matching problem (aka subgraph isomorphism problem) [196]. GPM can be done over both undirected and directed graphs respectively. Traditionally, GPM refers to a computational task of evaluating (or matching) graph patterns (sub-graphs) over a graph G in a graph database [13]. The most trivial form of a graph pattern is the Basic Graph Pattern (BGP). A set of BGPs form a Complex Graph Pattern (CGP). Matching has been formally defined in various texts such as [13, 60, 196,

197]. We adapt the definition provided by [13] in our current context:

Definition 18 (Match of a Graph Pattern $\llbracket P \rrbracket_G$). A graph pattern $P = (V_p, E_p, \gamma_p, \lambda_p, \sigma_p)$; is matching the graph $G = (V, E, \gamma, \lambda, \sigma)$, iff the following conditions are satisfied:

1. there exist mappings λ_p and σ_p such that, all variables are mapped to constants, and all constants are mapped to themselves (i.e. $\lambda_p \in \lambda, \sigma_p \in \sigma$),
2. each edge $\acute{e} \in E_p$ in P is mapped to an edge $e \in E$ in G , each vertex $\acute{v} \in V_p$ in P is mapped to a vertex $v \in V$ in G ,
3. the structure of P is preserved in G (i.e. P is a sub-graph of G) □

The principle of *matching* in edge-labeled graphs is analogous to that of the property graph (ref. Def. 18), i.e. – (i) there exists a mapping ϕ_p such that all constants are mapped to themselves and variables are mapped to constants; and (ii) the structure of P is preserved in G .

In Gremlin, GPM is performed by traversing³ over a graph G . In this sense, a GPM query in Gremlin can be perceived as a path traversal [76]. Rodriguez et al. [76] define a *path* as:

Definition 19 (Path). A path p in a graph G is a sequence, where $p \in E^*$ and $E \subset (V \times L \times V)$ ⁴. A path allows for repeated edges and the length of a path is denoted by $\|p\|$, which is equal to the number of edges in p . □

Each path query is composed is composed of an ordered list of one or more atomic operations called Single-Step Traversals (SSTs) ψ_s [70]. These are atomic-operations over the elements in a graph (i.e. nodes and edges) such as property-filter, label-filter, out/in edge traversal, out/in node traversal, etc. Through function composition and currying, it is possible to define a query of arbitrary length [70]. These path queries can be a combination of either a source, destination, labelled traversal or all of them in a varying fashion, depending on the user defined query.

Example 5.1.1. For instance, consider a simple path traversal to the oldest person that marko knows over the graph G as show in Figure 5.1. Listing 5.1 represents the gremlin query for the described traversal.

```
1 g.V().has("name", "marko").out("knows").values("age").max()
```

Listing 5.1: Return the age of the oldest person marko knows

Here, $g.V()$ i.e. V_g is the traverser definition bijective to V where, $\cup_i \mu((V_g)_i) = V$. Functionally, this query be written using function currying as:

$$\overline{\max(\text{values}_{age}(\text{out}_{knows}(\text{has}_{name=marko}(V_g))))} \quad (5.1)$$

³ The act of visiting vertices ($v \in V$) and edges ($e \in E$) in a graph is performed in an alternating manner (in some algorithmic fashion) [70].

⁴ The kleene star operation \star constructs the free monoid $E^* = \bigcup_{n=0}^{\infty} E^n$. where $E^0 = \{\epsilon\}$; ϵ is the identity/empty element.

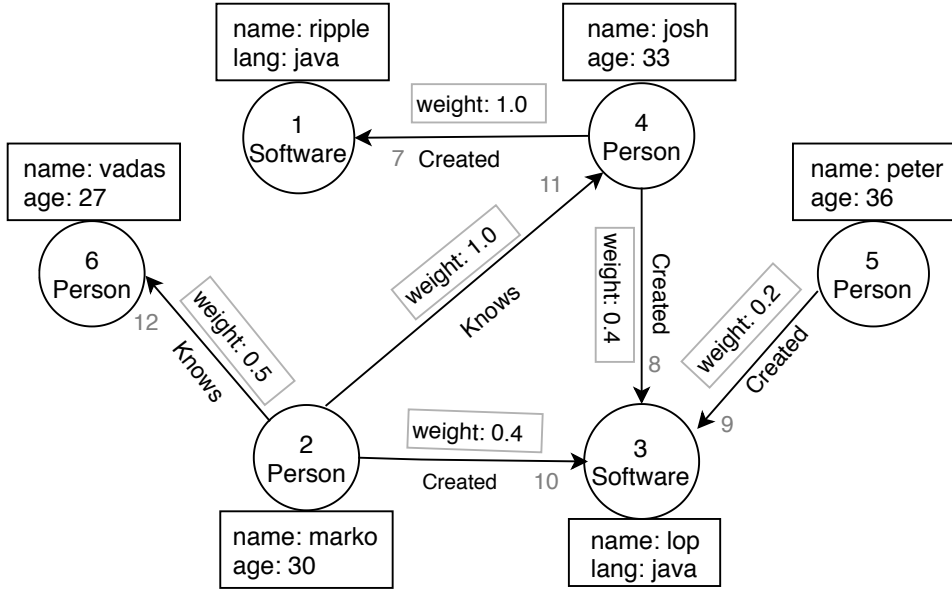


Figure 5.1: An example of a Property graph from the Apache TinkerPop project network.

The terms out_{knows} , $values_{age}$ and has_{name} are the single-step Gremlin operations/traversals. In [70], Rodriguez presents the itemisation of such single-step traversals which can be used to represent a complex path traversal. Thus, as described earlier, through functional composition and currying one can represent a graph traversal of random length. If i is the starting vertex in G , then the traversal shown in listing 5.1 can be represented as following function:

$$f(i) = \max(\epsilon_{age} \circ v_{in} \circ e_{lab+}^{knows} \circ e_{out} \circ \epsilon_{name+}^{marko}) (i) \quad (5.2)$$

where, $f : P(V) \rightarrow P(S)$.

In order to illustrate the *evaluation* of a Gremlin traversal Ψ over a property graph G ($[\Psi]_G$, cf. Definition 18), let us consider another example of a Gremlin traversal, shown in listing 5.2 below.

```
1 g.V().as('x').has('name','marko').out('Created').as('y')
```

Listing 5.2: Gremlin traversal for "What is created by Marko?".

Here, the underlying single step traversals are `.has('name','Marko')` and `.out('Created')` which basically refer to the `HasStep()` and `VertexStep()` instructions in the Gremlin instruction-set library [76, 77] respectively. Whereas, the `as()`-steps, which denote the start (and the end) of a traversal (or input graph patterns), are the naming variables analogous to the variables in a SPARQL query [77]. Furthermore, `g.V()`, returns the set of all vertices in the graph. We present a thorough categorization of these SSTs in Table 5.2 of Section 5.2.1.

Example 5.1.2. We illustrate the evaluation of graph patterns (a) and (b) from Figure 5.2, over the graph G from Figure 5.1.

$$(a) \ v[3] \quad (b) \ c=v[4], \ c=v[6]$$

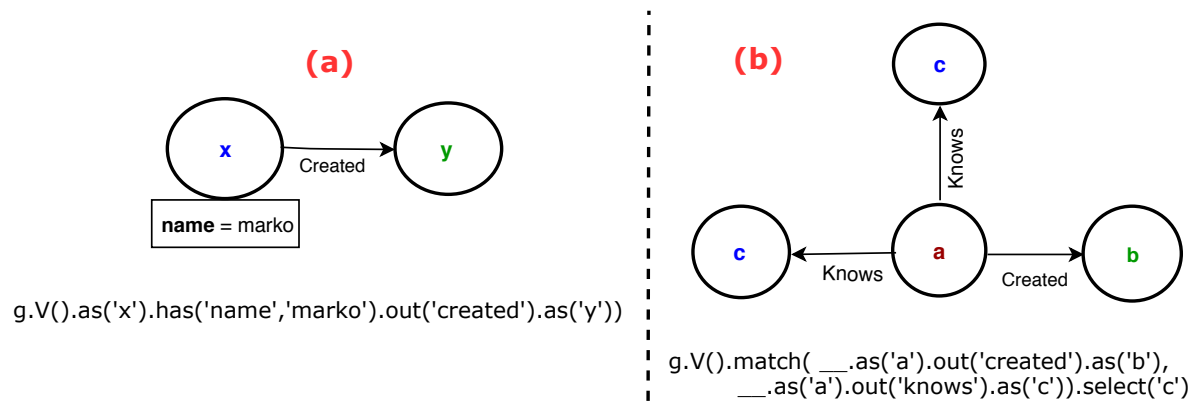


Figure 5.2: The graphical illustration of a (a) basic graph pattern and (b) a complex graph pattern, expressed in Gremlin to be matched over a property graph in Figure 5.1.

Evaluation of the `match()`-step in Gremlin

The `match()`-step in Gremlin, evaluates the input graph patterns over a graph G in a structure preserving manner binding the variables and constants to their respective values (cf. Definition 18). We denote the evaluation of a gremlin traversal Ψ over a Property graph G , using the notation $\llbracket \Psi \rrbracket_G$ which is borrowed from [60, 198]. When a `match()`-step is encountered by the Gremlin machine, it treats each graph pattern as an individual path traversal. These graph patterns are represented using `as()`-step⁵ (step-modulators⁶ i.e. naming variables) such as a , b , c , etc.), which typically mark the start (and end) of particular graph patterns or path traversals.

However, the order of execution of each graph pattern is up to the `match()`-step implementation, where the variables and path labels are local only to the current `match()`-step. Due to this uniqueness of the Gremlin `match()`-step it is possible to:

1. treat each graph pattern individually as a single step traversal and thus, construct composite graph patterns by joining (path-joins) each of these single step traversals;
2. combine multiple `match()`-steps for constructing complex navigational traversals (i.e. multi-hop queries), where each composite graph pattern (from a particular `match()`-step) can be joined using the concatenative join (ref. Section 5.1.1).

For instance, consider the Gremlin traversal shown in Listing 5.3 below.

```

1 g.V().match( __.as('a').out('created').as('b'),
2             __.as('b').has('name','lop'),
3             __.as('b').in('created').as('c'),
4             __.as('c').has('age',30)).select('a','c').by('name')

```

Listing 5.3: This traversal returns the names of people who created a project named 'lop' that was also created by someone who is 30 years old.

⁵ Meaningful names can be used as variable names for enhancing query readability.

⁶ Rodriguez et al. [70] refer to step modulators as 'syntactic sugar' that reduce the complexity of a step's arguments by modifying the previous step.

Each of the comprising four graph patterns (traversals) of the query (listing 5.3), can be individually represented using the curried functional notation as described in Equation 5.1. Thus,

$$f(i) = (e_{lab+}^{created}) \circ e_{out} (i); \quad g(i) = (\epsilon_{name+}^{lop} \circ v_{in}) (i); \quad (5.3)$$

$$h(i) = (e_{lab+}^{created}) \circ e_{in} (i); \quad j(i) = (\epsilon_{age+}^{30} \circ v_{in}) (i) \quad (5.4)$$

The input arguments of the `match()`-step are the set of graph patterns defined above in equations 5.3 and 5.4, which form a composite graph pattern (the final traversal (Ψ)). At run-time, when a traverser enters `match()`-step, it propagates through each of these patterns guaranteeing that, for each graph pattern, all the prefix and postfix variables (i.e. "a", "b", etc) are *binded* with their labelled path values. It is only then allowed to exit, having satisfied this condition. In simple words, though each of these graph patterns is evaluated individually, it is made sure that at run-time, the overall structure of the composite graph pattern is preserved by mapping the path labels to declared variables.

For instance, in the query (ref. listing 5.3), the starting vertex of $g(i)$ labelled as "b" which is the terminal vertex of $f(i)$, similarly for $h(i)$ and $j(i)$ with vertex labelled as "c". It is therefore necessary, to keep a track of the current location of a traverser in the graph, to preserve traversal structure. This is achieved in Gremlin by `match()` and `bind()` functions respectively, which we outline next.

The evaluation of an input graph pattern/traversal in Gremlin is taken care by two functions:

1. the recursively defined `match()` function- which evaluates each constituting graph pattern and keeps a track of the traversers location in the graph (i.e. path history), and,
2. the `bind()` function- which maps the declared variables (elements and keys) to their respective values.

Using equations (5.3, 5.4) (curried functional form of path traversals) in the recursive definition of `match()` by [77], we have:

$$\llbracket t \rrbracket_g = \begin{cases} \llbracket bind_b(\mathbf{f}(t_{\Delta_a(t)} \wedge \Delta_{m1})) \rrbracket_g & : \Delta_a \neq \phi = \Delta_{m1} \\ \llbracket \mathbf{g}(t_{\Delta_b(t)} \wedge \Delta_{m2}) \rrbracket_g & : \Delta_b \neq \phi = \Delta_{m2} \\ \llbracket bind_c(\mathbf{h}(t_{\Delta_b(t)} \wedge \Delta_{m3})) \rrbracket_g & : \Delta_b \neq \phi = \Delta_{m3} \\ \llbracket \mathbf{j}(t_{\Delta_a(t)} \wedge \Delta_{m4}) \rrbracket_g & : \Delta_c \neq \phi = \Delta_{m4} \\ t & : \text{otherwise,} \end{cases} \quad (5.5)$$

where, $t_{\Delta_a}(t)$ is the labelled path of traverser t . A path (ref. definition 19) is labelled "a" via the step-modulator `.as()`, of the traverser in the current traversal (Ψ); Δ_{m1} , Δ_{m2} , Δ_{m3} are hidden path labels which are appended to the traversers labelled path for ensuring that each

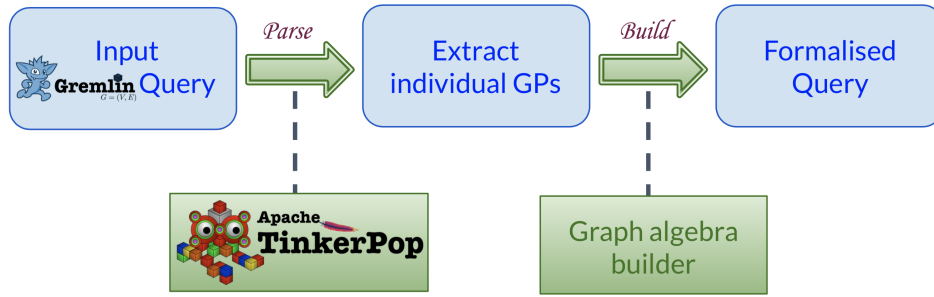


Figure 5.3: Conceptual architecture for formalizing a Gremlin traversal using graph relation algebra.

pattern is executed only once; and $\text{bind}_x(t)$ is defined as:

$$\text{bind}_x(t) = \begin{cases} t_{\Delta_x}(t) = \mu(t) & : \Delta_x(t) = \phi \\ t & : \Delta_x(t) = \mu'(t) \\ \phi & : \text{otherwise.} \end{cases} \quad (5.6)$$

where $\mu': T \rightarrow U$, is a function that maps a traverser to its current location in the graph G (e.g., $v \in V, V \in U$) [77]. It (μ') can be perceived analogues to μ , which maps elements and keys to values in G , however in later case the value is the location of a traverser in G .

5.1.3 Mapping Gremlin traversals to Graph Algebra

We now present a mapping algorithm for encoding a given Gremlin pattern-matching traversal, relational graph algebra. Figure 5.3, describes the conceptual architecture for formalizing Gremlin traversals. We follow a bottom-up approach in order to construct the relational graph algebra based on the traversal.

1. The input query is parsed and its constituent individual graph patterns are extracted from the `match()`-step and the optional `where()` step. */* Parse step*
2. For each single graph patterns (single path traversals) in the query, we first construct the curried functional form (ref. equation 5.1). */* Steps 2-12 are the Build steps*
3. We then map the `get-vertices/get-edges` operator for the encountered `g.V()/g.E()` step (i.e. to the first graph pattern) respectively.
4. Append a *traverse-operator* to all the respective in-coming and outgoing edge traversals for each, that appear inside the `match()`-step.
5. Append a *property-filter* operator to all the respective `has()` and `values()` steps based on the `match()`-step.
6. Multiple `match()` steps can be connected processed using the concatenative join operator.
7. Append a *selection* operator, if the `match()` step is succeeded by a `where()` step (this is

an optional in gremlin queries).

8. Append a *projection* operator, if `select()`-step is declared with a `match()` or `where()` step.
9. Append a *deduplication* operator, based on whether the `dedup()` step is declared after the `select()` step.
10. Append a *sorting* operator, if the `order()` step with an optional `by()` modifier is declared after the `select()` step.
11. Append a *grouping* operator, if the `group()` step with an optional `by()` modifier is declared after the `select()` step.
12. Map the *union* operator if the query contains a `union()`-step⁷. While *Union* is a binary operator, a union of multiple patterns can be constructed using a left deep join tree representation.

We now present three examples to illustrate the formalization of gremlin traversals using graph relational algebra.

Query Formalisation Example 1

The example gremlin traversal as shown in listing 5.3, can be formalized as:

$$\dagger_{name} \left(\Pi_{a,c} \left(\underbrace{\llbracket \sigma_{age=30}^c \downarrow_b^c [created] \sigma_{name=top}^b \uparrow_a^b [created] \rrbracket (V_g)}_t \right) \right) \quad (5.7)$$

Query Formalisation Example 2

Consider the following gremlin traversal shown in listing 5.4 below:

```

1 g.V().match(
2   __.as('a').hasLabel('person').values('age').as('b')).select('b')
   .order().by(asc)
    
```

Listing 5.4: This traversal returns the list all the persons in the ascending order of the age.

The gremlin traversal shown above (Listing 5.4) can be formalized as follows:

$$\mathfrak{R}_{\uparrow_b} \left(\Pi_b \left(\underbrace{\llbracket \sigma_{age}^b \sigma_{label=person}^a \rrbracket (V_g)}_t \right) \right) \quad (5.8)$$

⁷ It is not a common practice to use a `union()`-step in Gremlin GPM traversals, as multiple `match()`-steps in conjunction with `where()`-steps can be used as per required (the ad infinitum style of traversing [77]).

Query Formalisation Example 3

Consider the following gremlin traversal shown in listing 5.5 below,

```

1 g.V().union(
2   ___.match( ___.as('a').out('created').as('c') ),
3   ___.match( ___.as('b').out('created').as('c') ) ).select('a','c')
```

Listing 5.5: This traversal returns the list of all the people who have collaboratively created a software.

The gremlin traversal shown above (Listing 5.5) can be formalized as follows:

$$\Pi_{b,c} \left(\underbrace{\llbracket \uparrow_a^c [created] \rrbracket_g \uplus \llbracket \uparrow_b^c [created] \rrbracket_g \rrbracket_g}_{t = t_1 \uplus t_2} \right) \quad (5.9)$$

5.2 GREMLINATOR Approach

In this section, we discuss the GREMLINATOR approach for translating SPARQL queries into pattern matching Gremlin traversals. We first start by elaborating on the correspondence between the components of SPARQL queries and Gremlin traversals respectively. In doing so, we create a formal analogy borrowing the evaluation semantics of a SPARQL query [13, 60, 78] and put them in context of Gremlin traversals [70, 76, 77]. For a detailed discussion on the semantics of Gremlin graph operators we refer to [34].

5.2.1 Mapping SPARQL BGPs to Gremlin SSTs

We know that, each subject (s) and object (o) (i.e. nodes) in a triple is connected through only one predicate (p) relation (i.e. edges). Furthermore, Gremlin provides a pattern matching construct, analogous to SPARQL using the `match()`-step, enabling the user to represent a complex traversal using multiple individual connected or disconnected graph patterns (i.e. SSTs).

Due to this functionality *and* given the nature of the information encoded in a triple, it is possible to represent the underlying traversal operation using an SST, which is represented by its predicate/edge. Thus, each predicate in a triple pattern of a SPARQL BGP manifests the SST required for matching the graph pattern. We illustrate the different types of Gremlin SSTs and their correspondence with the SPARQL triple patterns in Table 5.2.

We classify the itemization of the Gremlin SSTs from [77] into four categories, that can be combined together to form a complex path traversal (analogous to CGP in SPARQL). This categorization is based on the predicate-object combination (s p o) of the corresponding SPARQL triple pattern, i.e. whether it refers to the – label of a vertex/edge (L) *or* property literal/value of a vertex/edge (P1) *or* variable associated to a property value (P2) *or* a traversal to-and-from a vertex given an edge (E). The corresponding SSTs can be grouped into the following four cases:

SST	SPARQL Triple Pattern (P)	Corresponding Gremlin Step $\gamma(P) = \psi_s$	Case
L_v	{ ?x v:label "person" . }	[MatchStartStep(x), HasStep ([\sim label.eq(person)]), MatchEndStep]	L
L_e	{ ?x e:label "knows" . }	[MatchStartStep(x), HasStep ([\sim label.eq(knows)]), MatchEndStep]	
P_v	{ ?x v:name "Marko" . }	[MatchStartStep(x), HasStep ([name.eq(Marko)]), MatchEndStep]	P1
P_e	{ ?x e:weight "0.8" . }	[MatchStartStep(x), HasStep ([weight.eq(0.8)]), MatchEndStep]	
P_e	{ ?x e:weight ?y . }	[MatchStartStep(x), PropertiesStep ([weight],value), MatchEndStep(y)]	P2
P_v	{ ?x v:name ?y . }	[MatchStartStep(x), PropertiesStep ([name],value), MatchEndStep(y)]	
E_{OUT}	{ ?x e:knows ?x . }	[MatchStartStep(x), VertexStep (OUT,[knows],vertex), MatchEndStep(y)]	E
E_{IN}	{ ?y e:knows ?x . }*	[MatchStartStep(y), VertexStep (IN,[knows],vertex), MatchEndStep(x)]	

Table 5.2: Correspondence between the SPARQL triple patterns and single step traversals (SSTs) from the Gremlin instruction library. Each of the SPARQL triple pattern can be mapped to a particular Gremlin single step traversal.

- **Case L** – Traversal to access the label values of a vertex or an edge (L_v/L_e)
- **Case P1/P2** – Traversal to access the property values of a vertex *or* an edge (P_v/P_e)
- **Case E** – Incoming/outgoing traversal between two adjacent vertices given an edge label (E_{in}/E_{out})

These four categories serve as our base cases that can be used to construct any complex traversal given their corresponding SPARQL CGPs. Table 5.2 illustrates the association between Gremlin SSTs (ψ_s) and the SPARQL triple patterns.

For the SPARQL BGP from listing 5.1 the corresponding Gremlin SSTs for the triple patterns are the cases – P_v (as the traversal is to access the property value of a vertex) and E_{out} (as the traversal is from a vertex named "Marko" via the edge labelled "Created").

5.2.2 Mapping SPARQL Queries to Gremlin Traversals

Given that SPARQL is a graph pattern matching query language, it is intuitive to use SPARQL to query property graphs. Moreover, we have seen that Gremlin provides several features to express pattern matching in terms of path traversal. Next, we describe the SPARQL to Gremlin query transformation function.

SPARQL-to-Gremlin Transformation Function

Consider the function $\gamma(P)$, which takes a SPARQL graph pattern P as input and returns a Gremlin expression.

The function γ is defined recursively as follows:

- If P is a triple pattern (v_1, v_2, v_3) then $\gamma(P) = _as("v_1").has("v_2","v_3")$ when $v_3 \in L$, and $\gamma(P) = _as("v_1").out("v_2").as("v_3")$ when $v_3 \in I$;
- If P is a graph pattern of the form $\{P_1 JOIN P_2\}$ then $\gamma(P) = match(\gamma(P_1),\gamma(P_2))$;

- If P is a graph pattern of the form $\{P_1 \text{ UNION } P_2\}$ then $\gamma(P) = \text{union}(\gamma(P_1), \gamma(P_2))$;
- If P is a graph pattern of the form $\{P_1 \text{ OPTIONAL } P_2\}$ then $\gamma(P) = \gamma(P_1).\text{optional}(\gamma(P_2))$;
- If P is a graph pattern of the form $\{P_1 \text{ FILTER } C\}$ then $\gamma(P) = \gamma(P_1).\text{where}(\gamma(C))$.

We overload the function γ to evaluate a filter condition C as follows:

- If C is $(? X = c)$ then $\gamma(C) = ("? X", \text{eq}("c"))$;
- If C is $(? X = ? Y)$ then $\gamma(C) = ("? X", \text{eq}("? Y"))$;
- If C is $(! C_1)$ then $\gamma(C) = \text{not}(\gamma(C_1))$;
- If C is $(C_1 \parallel C_2)$ then $\gamma(C) = \text{or}(\gamma(C_1), \gamma(C_2))$;
- If C is $(C_1 \&\& C_2)$ then $\gamma(C) = \text{and}(\gamma(C_1), \gamma(C_2))$.

We also overload the function γ to evaluate solution modifiers and the SELECT clause as follows:

- If E is a clause GROUP BY $? X_1 \dots ? X_n$ then $\gamma(E) = \text{.group.by}("? X_1"). \dots \text{.by}("? X_n")$;
- If E is a clause ORDER BY $? X$ then $\gamma(E) = \text{.order.by}("? X", \text{incr})$;
- If E is a clause ORDER BY ASC($? X$) then $\gamma(E) = \text{.order.by}("? X", \text{incr})$;
- If E is a clause ORDER BY DESC($? X$) then $\gamma(E) = \text{.order.by}("? X", \text{decr})$;
- If E is a clause LIMIT n then $\gamma(E) = \text{.limit}(n)$;
- If E is a clause LIMIT n OFFSET m then $\gamma(E) = \text{.range}(m, n)$;
- If E is the clause SELECT $? X_1 \dots ? X_n$ then $\gamma(E) = \text{.select}("? X_1", \dots, "? X_n")$;
- If E is the clause SELECT DISTINCT $? X_1 \dots ? X_n$ then $\gamma(E) = \text{.select}("? X_1", \dots, "? X_n").\text{dedup}()$.

Given a SPARQL query $Q^S = \{S, W, GB, OB, LO\}$, the Gremlin expression obtained from Q^S will be:

$$\text{g.v}().\gamma(P).\gamma(GB).\gamma(OB).\gamma(LO).\gamma(S) = \Psi$$

Note that the sequence of steps in the Gremlin expression reflects the structure of the input SPARQL query. Although the above definition assumes that all the components of a query are present, it is easy to deduce the transformation when one or more optional components are missing.

Operation	SPARQL k/w	Gremlin k/w	SPARQL construct	Gremlin construct
Matching	WHERE { ... }	MatchStep(AND, [])	WHERE { BGP ₁ . BGP ₂ BGP _n }	[MatchStep(AND, [[ψ_1], [ψ_2], ..., [ψ_n]]
Restriction	FILTER(C)	IsStep(C)	FILTER (?v1 <30)	IsStep(It(30))
Join	JOIN	AndStep()	BGP ₁ * BGP ₂ * ... * BGP _n	AndStep([[ψ_1], [ψ_2], ..., [ψ_2]])
Projection	SELECT	SelectStep()	SELECT ?v1 ?v2 ... ?v _n	SelectStep([a, b, ... , n])
Combination	UNION	UnionStep()	{BGP ₁ } UNION {BGP ₂ }	UnionStep(ψ_1, ψ_2)
Left Join	OPTIONAL	CoalesceStep()	{BGP ₁ } OPTIONAL { BGP ₂ }	$\psi_1, \text{CoalesceStep}(\psi_2)$
Deduplication	DISTINCT	DedupStep()	DISTINCT ?v1	DedupStep([v ₁])
Restriction	LIMIT (M)	RangeStep(0, M)	LIMIT 2	RangeStep(0, 2)
Restriction	OFFSET (N)	RangeStep(N, M+N)	OFFSET 10	RangeStep(10, 12)
Sorting	ORDER BY()	OrderStep()	ORDER BY DESC(?a)	OrderStep([[value(a), desc]])
Grouping	GROUP BY()	GroupStep()	GROUP BY(?a)	GroupStep(value(a))

Table 5.3: A consolidated summary of the SPARQL constructs and keywords along with their corresponding Gremlin constructs and instruction steps.

SPARQL Query (Q)

Gremlin Traversal [$\gamma(Q) = \Psi$]

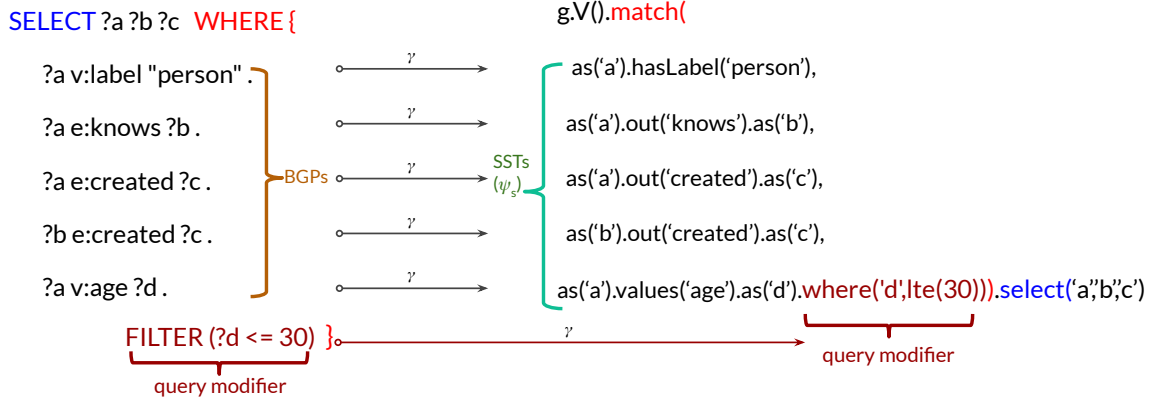


Figure 5.4: **Transformation Example 1.** Illustration of the transformation of an input SPARQL query (Q) to the corresponding Gremlin pattern matching traversal (Ψ).

5.2.3 Explanation of the Transformation

In this subsection, we explain the SPARQL to Gremlin mapping in more detail and use as example two query transformation, shown in Figures 5.4 and 5.5. Furthermore, Table 5.3 presents a summary of the equivalences between SPARQL operators and Gremlin expressions.

Triple patterns. Given a triple pattern (v_1, v_2, v_3) , the transformation function generates a different Gremlin expression depending on whether v_3 is a literal, or it refers to a IRI. In both cases the result is a simple traversal expression. In our sample transformation, the triple pattern `?a v:label "person"` is translated to the Gremlin expression `as("a").hasLabel("person")`.

JOIN graph patterns. A graph pattern $\{P_1 \text{ JOIN } P_2\}$ implies a natural join between the multisets obtained from P_1 and P_2 . This behavior is simulated in Gremlin by using the operator `match`, as it allows the join of a set of traversals. It is important to mention that a `match` can occur inside another `match`, in any level of nesting, so recursive matching is supported.

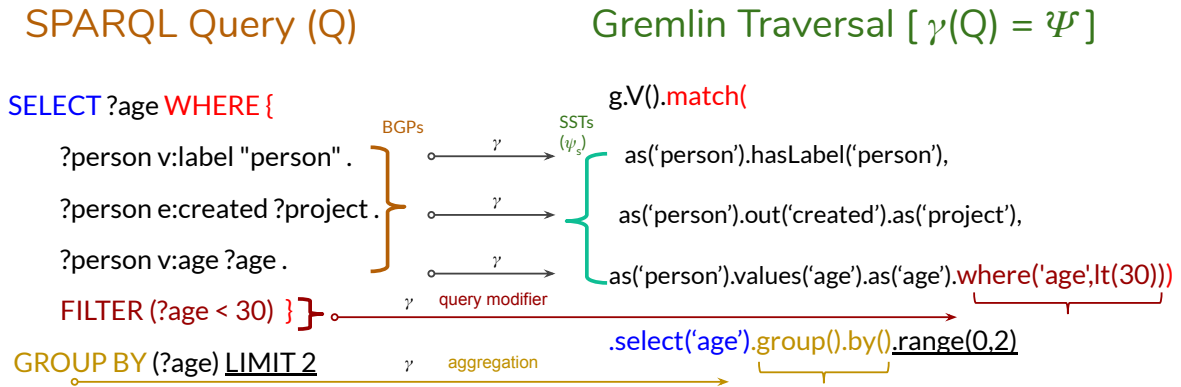


Figure 5.5: **Transformation Example 2.** Illustration of the transformation of an input SPARQL query (Q) to the corresponding Gremlin pattern matching traversal (Ψ).

OPTIONAL graph patterns. An optional graph pattern allows to introduce optional matching. Given a pattern $P = \{P_1 \text{ OPTIONAL } P_2\}$, if the optional P_2 does not match, then the results of P_1 are returned unchanged, else additional bindings of P_2 are added to the solution of P . The OPTIONAL operator corresponds to a *left outer join* operation in relational algebra. Optional graph patterns are supported in Gremlin by using the operator `.optional()`.

UNION graph patterns. A UNION operator allows to combine the solutions of two graph patterns. Gremlin provides the operator `code` to implement the union of two SSTs (i.e. the result set of two traversals). The solution set returned after the union operation is not de-duplicated by default, because of the governing bag semantics. Thus, all possible solutions are returned.

FILTER graph patterns. The FILTER operator is used to restrict the results obtained after evaluating a graph pattern. Several types of filter conditions are supported, including equalities, inequalities and boolean conditions (in our example, `?a <= 30`). Filter conditions are supported in Gremlin by using the operator `.where(C)`, where C is a constraint. Gremlin provides several operators to implement simple and complex filter conditions.

SELECT. This clause allows to project the variables in multiset obtained by the graph pattern matching step. This feature is implemented in Gremlin by using the `.select()` operator (analogous to the SELECT operator in SPARQL).

Solution Modifiers. The solution set returned by the evaluation of a graph pattern is not *de-duplicated* or *ordered* by default, as both languages operate on bag semantics. Therefore, *solution modifiers* are used to sort, group or filter duplicated objects in the solution. Each SPARQL query modifier considered in this paper has a corresponding operator in Gremlin. For instance, the DISTINCT operator of SPARQL is implemented with the `dedup` operator of Gremlin.

5.3 Implementation

In this section, we discuss the implementation of GREMLINATOR approach (i.e. the Apache TinkerPop `sparql-gremlin` plugin).

5.3.1 Encoding Prefixes

We encode the prefixes of SPARQL queries within the GREMLINATOR implementation, in order to aid the translation process. We define the custom namespace:

`http://tinkerpop.apache.org/traversal/` for the graph elements (i.e. vertex, edge, and properties). For instance, an edge is referred to using the URI:

```
http://tinkerpop.apache.org/traversal/edge.
```

We also define custom prefixes for the IRIs, keeping in mind the corresponding Gremlin SSTs. For instance, the label prefix (which is a predicate in a SPARQL query - "`rdfs:label`") is encoded as `e:label` or `v:label` (where `e=edge`, `v=vertex`).

5.3.2 GREMLINATOR (`sparql-gremlin`) Architecture

In GREMLINATOR, an input SPARQL query passes through a series of four steps as shown in Figure 5.6, which comprise of the translation pipeline, to obtain the resultant Gremlin traversal.

Step 1 The input SPARQL query is first *parsed* using the Jena SPARQL processing module (ARQ). This allows: (i) validating the query, i.e. checking whether the input query is a valid SPARQL query, and (ii) generating an abstract syntax tree (AST) representation.

Step 2 After the AST of the parsed SPARQL query is obtained, the `opWalker` then visits each triple pattern of the SPARQL query and maps or re-writes them to the corresponding Gremlin SSTs, i.e. via the `Rewriter` module (cf. Figure 5.6).

Step 3 Thereafter, depending on the operator precedence obtained from the AST of the parsed SPARQL query, each of the corresponding SPARQL operators are mapped to their corresponding instruction steps from the Gremlin instruction library. A final conjunctive traversal (Ψ) is then generated by the `Translation Writer` module, by appending the SSTs and instruction steps.

Step 4 Finally, this final conjunctive traversal (Ψ) is used to generate Gremlin Bytecode⁸, which can be executed on any TinkerPop-enabled graph DMS.

Algorithm 1 we describe the GREMLINATOR SPARQL-to-Gremlin query mapping algorithm.

⁸ Bytecode is a list of primitive-valued, nested arrays of the form: `bytecode = [op,arg]*` where an `arg` can be another chunk of bytecode.

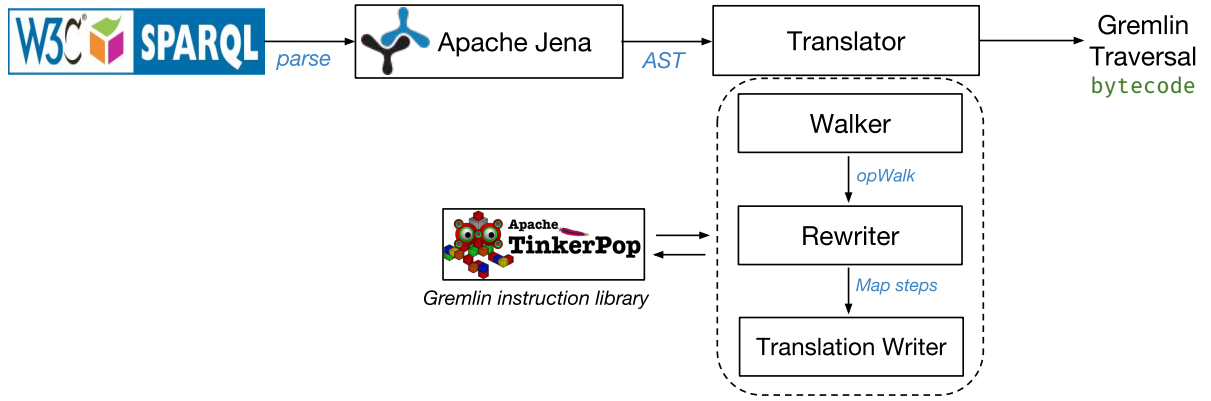


Figure 5.6: The GREMLINATOR (sparql-gremlin) query translation pipeline.

Algorithmus 1 : GREMLINATOR SPARQL-to-Gremlin query mapping algorithm

```

input  :  $SQ$  : SPARQL Query
output :  $GT$  : Gremlin Traversal
1   $GT \leftarrow \emptyset$ ;  $T \leftarrow \emptyset$  // list of single step traversals (SST) -  $T$ 
2  ;  $AST \leftarrow \text{getAST}(SQ)$ ;  $BGPs \leftarrow \text{getAllBGPs}(AST)$ 
3  foreach  $bgp_i \in BGPs$  do
4  |    $t_i \leftarrow t \cup \psi_{bgp_i}$ 
5  |    $t_i \leftarrow t \cup \psi_{bgp_i.s} \cup \psi_{bgp_i.p} \cup \psi_{bgp_i.o}$ 
6  |    $t_i \leftarrow \text{TraversalBuilder}(bgp_i, G)$ 
7  |    $T \leftarrow T \cup t_i$  // mapping BGPs to Gremlin SSTs ( $\psi_s = \rho(bgp_i)$ ) cf. Table 5.2
8  end
9  // building the Gremlin operator A.S.T. (cf. Table 5.3)
10 if  $c \leftarrow AST.FILTER, \exists c \neq \emptyset$  then
11 |   foreach  $c \in AST$  do
12 |   |    $T \leftarrow T \cup \text{WhereTraversalStep}(\psi_c)$ 
13 |   end
14 end
15 if  $c \leftarrow AST.UNION$  then
16 |    $GT \leftarrow \text{UnionStep}(\text{Match}(T))$ 
17 end
18 if  $|T| > 1$  then
19 |    $GT \leftarrow \text{Match}(T)$ 
20 else
21 |    $GT \leftarrow GT \cup T$ 
22 end
23 if  $c \leftarrow AST.ORDERBY$  then
24 |    $GT \leftarrow T \cup \text{OrderStep}(\psi_c)$ 
25 end
26 if  $c \leftarrow AST.GROUPBY$  then
27 |    $GT \leftarrow T \cup \text{GroupByStep}(\psi_c)$ 
28 end
29 if  $c \leftarrow AST.LIMIT$  then
30 |   if  $k \leftarrow AST.OFFSET$  then
31 |   |    $GT \leftarrow T \cup \text{RangeStep}(k, c + k)$ 
32 |   else
33 |   |    $GT \leftarrow T \cup \text{RangeStep}(c)$ 
34 |   end
35 end
36 return  $GT$ 
    
```

5.3.3 SPARQL Coverage and Limitations

Currently, GREMLINATOR supports the translation of SPARQL `SELECT` queries. It covers the SPARQL 1.0 specification (cf. Table 5.3), along with a subset of SPARQL 1.1 features (e.g. aggregation operators, explicit negation, solution modifiers). Furthermore, GREMLINATOR allows execution of composite queries over Graph DMSs. A composite query is a combination of both SPARQL and Gremlin constructs, wherein a SPARQL query can be appended with gremlin steps thus allowing a combination of declarative and imperative constructs. For instance:

```
g.sparql("SELECT * WHERE { }").out().label().dedup()
```

The current limitation of GREMLINATOR is that it does not support SPARQL queries with variables in the predicate position. Meaning that the predicate (`p`) in a triple pattern `{s p o .}` cannot be expressed as a variable. It has to be a defined operation for the traversal to be generated. This is because traversing a graph, natively in TinkerPop3, is not possible without knowing the precise (graph) traversal operation to the destination (vertex/edge) from the source (vertex/edge). However, this limitation is mitigated with the release of TinkerPop4 release, as it supports variables in predicate positions.

5.4 Experimental Evaluation

In this section, we describe – the evaluation methodology, the dataset and query descriptions, a carefully curated experimental setup (keeping in mind the various settings available to both RDF and Graph DMSs). We discuss the empirical evidence supporting the validity of our approach, report results and their meticulous discussion.

5.4.1 Evaluation Methodology

In order to empirically evaluate GREMLINATOR, we answer the following key questions:

- Q1) **Query preservation:** Do the GREMLINATOR generated Gremlin traversals return the same results as their SPARQL counterparts? i.e. is the proposed approach preserving the meaning of the input queries?
- Q2) **Translation validity:** Do the GREMLINATOR generated Gremlin traversals return the same results as the (manually created) traversals by the Gremlin experts? i.e. is the proposed approach generating valid traversals?
- Q3) **Performance analysis:** What do we observe upon comparing the execution performance of the SPARQL queries and their Gremlin counterparts?

5.4.2 Evaluation Metrics

The following conditions and metrics were considered for reporting all results.

- *Query translation time* (in milliseconds or ms) is reported for the average of 10 runs for each query translation (average time taken to translated a SPARQL query to a Gremlin traversal).
- *Query execution time* (in milliseconds or ms) is reported is the average of 10 runs for each query (of both SPARQL and translated Gremlin traversals).
- *Caching* – Queries were executed in both *cold* and *warm* cache settings for all DMSs. Where, a *warm cache*: implies that the cache is not cleared after each query run, and *cold cache*: implies that the cache is cleared using the `echo 3 > /proc/sys/vm/drop_caches` UNIX command after each query run.
- *Indexing schemes* – For Graph DMSs, query execution time is reported for both *with* and *without* creating explicit (manually created) indices. We elaborate on the reason for this, next.

Indexing in RDF Triplestores vs Graph Databases

RDF triplestores (or simply RDF DMSs⁹) typically consist of pre-defined indices. While, it is theoretically possible to have an RDF DMS totally index-free, it would imply performing a linear search through the entire set of RDF triples when executing each query against dataset. For this reason, having a strongly defined index setting within a RDF DMS by default is salient. For instance, *Virtuoso* maintains two all-purpose full (bitmap indices over PSOG, POGS) and three partial indices (over SP, OP GS) in the default configuration¹⁰. Furthermore, *4Store* in its default setting maintains a set of three full indices (R, P, M) [199], where – the *R-index* is a hash-map index over RDF resources (URIs, Literals, and Blank nodes); the *P-index* consists of a set of two radix trees per predicate, using a 4-bit radix; and the *M-index* is a hash-map based indexing scheme over RDF Graphs (G). Lastly, *Jena TDB* maintains three indices using a custom persistent implementation of B+ Trees¹¹.

On the other hand, the same cannot be said for Graph DMS wherein these indices can be created explicitly, depending upon the use case. Graph DMSs implement index-free adjacency (or neighbourhood) between the connected components (i.e. nodes). This implies that when searching through a Graph DMS, one is actually traversing through the index pointers rather than indexes themselves. For this reason, Graph DMSs do not require having a strongly defined index setting by default¹². They rather offer the possibility of creating explicit indexes

⁹ Instead of referring to the systems individually as an RDF triplestore or Graph database, we use the term Data Management System (DMS) throughout the chapter for better readability.

¹⁰ RDF indexing scheme in Virtuoso (<http://docs.openlinksw.com/virtuoso/rdfperfrdfscheme/>)

¹¹ RDF indexing scheme in Apache Jena TDB (<https://jena.apache.org/documentation/tdb/architecture.html#triple-and-quad-indexes>)

¹² <https://neo4j.com/blog/rdf-triple-store-vs-labeled-property-graph-difference/>

Northwind Dataset Features	RDF stats	Property graph stats
Classes	11	-
Entities	4413	-
Distinct subjects	4413	-
Distinct objects	8187	-
Properties	55	55
Number of Categories	8	8
Number of Customers	91	91
Number of Employees	9	9
Number of Orders	830	830
Number of Vendors	34	34
Number of Products	77	77
Number of Regions	4	4
Number of Shippers	3	3
Number of Suppliers	29	29
Number of Territories	53	53
Number of Instances or Nodes	3209	3209
Number of Triples or Edges	33003	6177

Table 5.4: Northwind RDF and Property graph dataset statistics

over custom graph elements (nodes, edges, attributes, etc), using a variety of data structures, depending on the implementation. For instance, *TinkerGraph* supports the creation of regular and composite hash-map indices (multiple key-value pairs) over node and edge attributes. *Neo4j* supports declaring regular indices (composite indices are supported in v3.5 and beyond) on graph elements (including labels). It offers a variety of indices ranging from Lucene index (for textual attributes) and as *SBTREE*-based index (numeric ones, such as IDs), which is based on custom implementation of B-Trees with several optimizations related to data insertion and range queries¹³. Lastly, like others *Sparksee* also supports user-defined indices on attributes, using a bitmap index implemented using sorted B-trees [200].

As we pointed out earlier, it is not possible to have a completely index-free RDF DMS. Thus, in order to grasp a better understanding of query performance with respect to various factors (such as indexing schemes, query typology and cache configuration) and also for the sake of fairness (towards Graph DMSs) we ran all the experiments with two Graph DMS settings: without and with manually created indices.

5.4.3 Datasets

We used the following datasets for the experimental evaluation of GREMLINATOR:

- *Northwind*¹⁴ dataset, which consists of synthetic data describing an e-commerce scenario between a fictional company "Northwind Traders", its customers and suppliers. Originated as a sample dataset shipped with Microsoft Access, the Northwind dataset is famous for benchmarking in relational database community;

¹³ Indexing scheme in Neo4j (<https://neo4j.com/docs/cypher-manual/current/schema/index/>)

¹⁴ Northwind dataset (<https://northwinddatabase.codeplex.com/>)

BSBM Dataset Features	RDF stats	Property graph stats
Classes	159	-
Entities	71015	-
Distinct subjects	71017	-
Distinct objects	166384	-
Properties	40	40
Number of Products	2785	2785
Number of Producers	60	60
Number of Product Features	4745	4745
Number of Product Types	151	151
Number of Vendors	34	34
Number of Offers	55700	55700
Number of Reviewers	1432	1432
Number of Reviews	27850	27850
Number of Instances or Nodes	92757	92757
Number of Triples or Edges	1000313	238309

Table 5.5: BSBM RDF and Property graph dataset statistics

- *Berlin SPARQL Benchmark* [168] (BSBM) dataset, which consists of synthetic data describing an e-commerce use case, involving a set of products, their vendors, and consumers who review the products. BSBM is widely famous for benchmarking RDF DMSs as it offers the flexibility of generating graphs of custom size and density. We generated one million triples for our experiments.

Tables 5.5 and 5.4 present the statistics of both the RDF and Property graph versions of the Northwind and BSBM datasets. The respective PG versions of the RDF datasets were created using trivial RDF graph pattern to PG pattern mapping rules. The python scripts for PG data generation (with the mappings), are accessible from [201].

5.4.4 Queries

We created a set of 30 SPARQL queries, for each dataset, which cover 10 different query features (i.e. three queries per feature with a combination of various modifiers). Table 5.6, summarizes their query design and the feature distribution. These features were selected after a systematic study of SPARQL query semantics [60, 78]. The queries cover a mix of BSBM [168] explore use cases¹⁵. Furthermore, a gold standard dataset of corresponding Gremlin traversals for the SPARQL queries, was created by three Gremlin experts. These expert generated traversals were used for a two-fold validation of the traversals generated by our approach (for Q2, cf. Section 5.4.1).

¹⁵ BSBM Explore Use Cases (<http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/spec/20110607/ExploreUseCase/index.html#querymixes>)

Query	Feature	FILTER	COUNT	LIMIT	DISTINCT	No. of TPs.	No. of Proj. var.
C1	CGP		✓		✓	2	2
C2	CGP				✓	1	1
C3	CGP				✓	1	1
F1	CONDITION	✓(1)				3	3
F2	CONDITION	✓(2)				3	3
F3	CONDITION	✓(1)			✓	2	1
L1	RESTRICTION	✓(1)		✓	✓	4	2
L2	RESTRICTION		✓			2	2
L3	RESTRICTION		✓			2	2
G1	GROUP BY		✓		✓	2	2
G2	GROUP BY	✓(1)				6	2
G3	GROUP BY		✓			1	2
Gc1	GROUP BY + COUNT		✓	✓		3	2
Gc2	GROUP + COUNT		✓			2	2
Gc3	GROUP + COUNT		✓	✓		1	2
O1	ORDER BY			✓		1	1
O2	ORDER BY	✓(1)				4	3
O3	ORDER BY			✓	✓	1	1
U1	UNION	✓(2)		✓		8	1
U2	UNION	✓(2)				6	2
U3	UNION	✓(2)			✓	4	1
Op1	OPTIONAL	✓(1)				3	3
Op2	OPTIONAL			✓	✓	6	2
Op3	OPTIONAL	✓(2)				8	3
M1	MIXED		✓	✓		3	2
M2	MIXED		✓	✓	✓	2	2
M3	MIXED		✓	✓		4	2
S1	STAR	✓(1)		✓		12	11
S2	STAR	✓(1)		✓		5	4
S3	STAR	✓(1)				10	9
TOTAL	30 Queries per Dataset	-	-	-	-	-	-

Table 5.6: A description of the feature compositions in each of the query dataset.

5.4.5 System Setup

We configured the following Data Management Systems: *RDF triplestores*: Openlink Virtuoso [v7.2.4], JenaTDB¹⁶ [v3.2.0], 4Store [v1.1.5]; *Graph databases*: TinkerGraph [18] [v3.2.3], Neo4j¹⁷ [v1.9.6], Sparksee¹⁸ [v5.1]. All experiments were performed on the following machine configuration: *CPU*: Intel[®] Xeon[®] CPU E5-2660 v3 (2.60GHz), *RAM*: 128 GB DDR3, *HDD*: 512 GB SSD, *OS*: Linux 4.2-generic. To ensure the reproducibility of our results, we provide the scripts, data and queries here¹⁹, and also provide a persistent URL²⁰ referring to all the resources used in this paper. All queries were executed in both *cold* and *warm* cache settings. The reported query translation and execution times are in milliseconds (ms).

¹⁶ Apache JenaTDB (<https://jena.apache.org/documentation/tdb/index.html>)

¹⁷ Neo4j Graph Database (<https://neo4j.com/>)

¹⁸ Sparksee – formerly DEX (<http://sparsity-technologies.com/#sparksee>)

¹⁹ Experimental Setup (<https://github.com/harsh9t/Gremlinator-Experiments>)

²⁰ All GREMLINATOR resources (<https://doi.org/10.6084/m9.figshare.8187110.v3>) [201]

5.5 Results and Discussion

We report the results with respect to the evaluation methodology described earlier. We executed the SPARQL queries against the three RDF triplestores and the GREMLINATOR translated Gremlin traversals against the three graph databases for both data sets and retrieved their results (for answering Q1, Q3). Furthermore, we also executed the (manually) expert generated Gremlin traversals on the graph databases, and compared their results with the GREMLINATOR generated traversals for a two-fold validation (for answering Q2).

We point the interested reader to [201], which is a persistent URL pointing to all the resources (scripts, queries, data, experimental setup, etc.) used in our experiments. The average time (of 10 runs per query) for translating a SPARQL query to the corresponding Gremlin traversal is **14 ms** for BSBM and **12.5 ms** for Northwind queries respectively.

5.5.1 Q1 - Query Preservation

We observed that for each SPARQL query and its translated Gremlin traversal, the returned results are equivalent (i.e. have the same values for each corresponding variable in the result set). The only difference was their representation format. SPARQL queries returned results as tables, whereas Gremlin traversals returned results as lists (or list of sets). Table 5.7 presents a subset of the query result. The complete list of all the queries and their corresponding results can be accessed from the Google spreadsheet available at (https://docs.google.com/spreadsheets/d/183a0ScNR6y7GVv8NV0116_TELS1oZA4R9HKSZVWo3jw/). Additionally, we validated the result returned by each translated Gremlin traversal against the corresponding manually created Gremlin traversal by the Gremlin experts (as stated in Section 5.4.4), for a two-fold cross-validation. Thus, based on the detailed empirical evidence gathered from our carefully conducted experiments we can conclude that the proposed translation approach is query preserving. Therefore, the evaluation of a SPARQL query Q over an RDF graph RDF is equivalent to the the evaluation of the translated Gremlin traversal Ψ over the corresponding Property graph G , notationally: $\llbracket Q \rrbracket_{RDF} \equiv \llbracket \Psi \rrbracket_G$.

5.5.2 Q2 - Translation Validity

We report that both Gremlin traversals (i.e. automatically translated via GREMLINATOR and manually translated by the Gremlin experts), when executed against the graph databases, also returned the same results. Therefore, the translated traversals are valid. We point the interested reader to the Google spreadsheet pointed our earlier (subsection above) for the complete results.

5.5.3 Q3 - Performance Analysis

Figures 5.7 and 5.8, presents the plots of our experimental results, in all four settings, for the BSBM and Northwind datasets respectively. The plots follow log scale for execution time (in ms).

Q.	SPARQL Query	Feature	SPARQL Query Result	Gremlin Traversal Result
C1	SELECT (COUNT (DISTINCT (?product)) as ?total) WHERE { ?a v:type "review" . ?a e:edge ?product . }	BGP	2787	2787
F3	SELECT DISTINCT ?pid WHERE { ?a v:productID ?pid . ?a v:ProductPropertyNumeric_1 ?property1 . FILTER (?property1 = 1) }	FILTER	?pid bsbm:inst/Product1636 bsbm:inst/Product2295	{ pid=1636 } { pid=2295 }
L2	SELECT ?rating1 WHERE { ?a v:type "review" . ?a v:Rating_1 ?rating. ?a e:edge ?product. ?product v:productID ?pid . FILTER (?pid = 343) . } LIMIT 2	LIMIT	?rating1 9 7	{ rating1=9 } { rating1=7 }
G2	SELECT ?product WHERE { ?a v:type "reviewer" . ?a v:reviewerID ?rid. ?a e:edge ?review . ?review v:Rating_1 ?rating1. ?review e:edge ?product. ?product v:productID ?pid. FILTER (?rid = 86). } GROUP BY (?rating1)	GROUP BY	?product bsbm:inst/Product1107 bsbm:inst/Product1301 bsbm:inst/Product1852 bsbm:inst/Product2291 bsbm:inst/Product1098 bsbm:inst/Product1954 bsbm:inst/Product1994 bsbm:inst/Product1355 bsbm:inst/Product734 bsbm:inst/Product1448 bsbm:inst/Product1426 bsbm:inst/Product1817 bsbm:inst/Product1141 bsbm:inst/Product1194 bsbm:inst/Product451 bsbm:inst/Product1294 bsbm:inst/Product1532	{ product=1107 } { product=1301 } { product=1852 } { product=2291 } { product=1098 } { product=1954 } { product=1994 } { product=1355 } { product=734 } { product=1448 } { product=1426 } { product=1817 } { product=1141 } { product=1194 } { product=451 } { product=1294 } { product=1532 }
Gc2	SELECT ?product (COUNT (?review) as ?total) WHERE { ?review v:type "review" . ?review e:edge ?product . ?product v:productID ?pid. } GROUP BY (?product) LIMIT 10	GROUP COUNT	?product ?total bsbm:inst/Product2588 1 bsbm:inst/Product3 1 bsbm:inst/Product2331 2 bsbm:inst/Product2553 3 bsbm:inst/Product1803 5 bsbm:inst/Product2440 7 bsbm:inst/Product2201 5 bsbm:inst/Product316 3 bsbm:inst/Product2210 7	{ Product=2588, Total=1 } { Product=3, Total=1 } { Product=2331, Total=2 } { Product=2553, Total=3 } { Product=1803, Total=5 } { Product=2440, Total=7 } { Product=2201, Total=5 } { Product=316, Total=3 } { Product=2210, Total=7 }
O2	SELECT DISTINCT ?product ?label WHERE { ?a v:productTypeID ?tid. FILTER(?tid = 58). ?a e:edge ?product. ?product v:productID ?pid. ?product v:label_n ?label. } ORDER BY (?product) LIMIT 5	ORDER BY	product label bsbm:inst/Product11 "pipers pests" bsbm:inst/Product18 "boondogglers" bsbm:inst/Product489 "airsickness simplices skiing" bsbm:inst/Product694 "nahuatls terrifiers direr" bsbm:inst/Product709 "jacinth medusoids"	{ pid=11, lab=pipers pests } { pid=18, lab=boondogglers } { pid=489, lab=airsickness simplices skiing } { pid=694, lab=nahuatls terrifiers direr } { pid=709, lab=jacinth medusoids }
U1	SELECT ?label WHERE { { ?a v:productTypeID ?tid. FILTER(?tid = 58). ?a e:edge ?product. ?product v:productID ?pid. ?product v:label_n ?label. } UNION { ?a v:productTypeID ?tid. FILTER(?tid = 102). ?a e:edge ?product. ?product v:productID ?pid. ?product v:label_n ?label. } } LIMIT 10	UNION	?label "airsickness simplices skiing" "nahuatls terrifiers direr" "jacinth medusoids" "slowed cloche" "meshwork" "nonradical warehousing" "furnacing" "accommodator" "collectivized mathematics" "brachiate writeoff"	{ label=airsickness simplices skiing } { label=nahuatls terrifiers direr } { label=jacinth medusoids } { label=slowed cloche } { label=meshwork } { label=nonradical } { label=warehousing } { label=furnacing } { label=accommodator } { label=collectivized mathematics } { label=brachiate writeoff }
Op1	SELECT ?pText2 ?pText3 ?pNum2 WHERE { ?product v:productID ?pid . FILTER (?pid = 343) . ?product rdfs:label ?label. ?product v:ProductPropertyTextual2 ?propertyTextual_2 . ?product v:ProductPropertyTextual3 ?propertyTextual_3 . OPTIONAL { ?product v:productID ?pid . FILTER (?pid = 350) . ?product rdfs:label ?label. ?product v:ProductPropertyNumeric_2 ?propertyNumeric2 . ?product v:ProductPropertyTextual3 ?propertyTextual_3 . }	OPT.	pText2 pText3 pNum2 "cyanided uncharged gametes" "fluorosis appeasing railheads critics satirizer controllers" 758	{ pText_2=cyanided uncharged gametes, pText_3=fluorosis appeasing railheads critics satirizer controllers, pNum2_2=758 }
M1	SELECT ?reviewer (COUNT (?product) as ?total) WHERE { ?reviewer v:type "reviewer". ?reviewer e:edge ?review. ?review e:edge ?product . } GROUP BY (?reviewer) ORDER BY DESC (?total) LIMIT 10	MIX	bsbm:inst/Reviewer1294 42 bsbm:inst/Reviewer501 41 bsbm:inst/Reviewer424 39 bsbm:inst/Reviewer281 38 bsbm:inst/Reviewer1263 38	[1294:42, 501:41, 424:39, 281:38, 1263:38]
S1	SELECT ?plabel ?label ?flabel ?proptext1 ?proptext2 ?proptext3 ?propnum1 ?propnum2 ?comment WHERE { ?producer v:type "producer". ?producer v:label_n ?plabel. ?producer e:edge ?product. ?product v:type "product". ?product v:productID ?pid. FILTER(?pid = 343). ?product v:label_n ?label. ?product v:comment ?comment. ?product v:ProductPropertyTextual_1 ?proptext1. ?product v:ProductPropertyTextual_2 ?proptext2. ?product v:ProductPropertyTextual_3 ?proptext3. ?product v:ProductPropertyNumeric_1 ?propnum1. ?product v:ProductPropertyNumeric_2 ?propnum2. ?product e:edge ?feature. ?feature v:type "product_feature". ?feature v:label_n ?flabel. } LIMIT 1	STAR	?label ?comment ?p ?f ?productFeature ?producer ?propertyTextual1 ?propertyTextual2 ?propertyTextual3 ?propertyNumeric_1 ?propertyNumeric2 ?ors "sobbers kynurenic undergoing remained horsed sidings hutpza continence flighty japingly semiretired crispest chukkers bamboozler shivah lagged miggs snickering arbitrators propped osmic mismeeting dissimulate fraudulently cabled yellor truncheons sigil expatriating viceless merrymakers fetas recompenses disreputability taperer multiplexed toddler disaffiliating radiating worshipper flamboyance waggly bothering swindlers eucharistical ensenfing lightfaced tench tramping margraves bewilderment deuteronomy contravened fourpenny coveralls traitorousness millpond redetermine jeremiad resealable abreaction marblers whisks" bsbm:inst/Producer8 bsbm:inst/ProductFeature11 "entoiling" "assignat disrobe" "housewifeliness neoliths proselytizers infirmable meditations bedchair maschera hagfish saplings prearranges debacles bedews straying grouter stereophonically" "cyanided uncharged gametes" "fluorosis appeasing railheads critics satirizer controllers" 1165 1526	{ ProductPropertyNumeric_1:[1165], productID:[343], ProductPropertyTextual_1:[cyanided uncharged gametes], ProductPropertyNumeric_2:[1526], ProductPropertyTextual_2:[fluorosis appeasing railheads critics satirizer controllers], label_n:[ors], comment:[sobbers kynurenic undergoing remained horsed sidings hutpza continence flighty japingly semiretired crispest chukkers bamboozler shivah lagged miggs snickering arbitrators propped osmic mismeeting dissimulate fraudulently cabled yellor truncheons sigil expatriating viceless merrymakers fetas recompenses disreputability taperer multiplexed toddler disaffiliating radiating worshipper flamboyance waggly bothering swindlers eucharistical ensenfing lightfaced tench tramping margraves bewilderment deuteronomy contravened fourpenny coveralls traitorousness millpond redetermine jeremiad resealable abreaction marblers whisks],type:[product],label:hedgehogs barstools,label_prod:assignat disrobe

Table 5.7: Comparison of results of a subset of SPARQL queries and their corresponding Gremlin traversals for the BSBM dataset.

Furthermore, we also report the detailed query-wise results in tabular format in Appendix A, for more comprehensive understanding. As we observe similar trend of performance of SPARQL vs. Gremlin queries over both the datasets, which is evident from Figures 5.7 and 5.8 and also the Tables A.1 and A.2 of Appendix A. We, therefore, only present the detailed performance analysis for the BSBM dataset. Furthermore, a similar performance trend was also observed in an independent study by [202] (cf. Section 5.7.1), which use GREMLINATOR, comparing SPARQL queries executed over Openlink Virtuoso vs the translated gremlin traversals executed over JanusGraph. We organize our observations on the performances of participating DMSs as follows, and present our discussion.

Graph DMSs without indexing scheme

Our findings for *cold* and *warm* cache are:

1. **Cold cache:** SPARQL queries perform better with respect to Gremlin traversals, leveraging the indexing schemes of RDF DMSs. They perform 1-2 times faster on simple queries (C1, C2) and order by (O1, O3); and 3-5 times faster for union and mixed queries (U1-3, M1-3). Whereas, Gremlin traversals perform 1-2 times faster on restriction (L1, L3), group by (G1-3) and conditional (F1-3) queries; 3-5 times faster on group count (Gc1-3) and star (S1-3) queries. We note that aggregation queries (counts, group counts) in Graph DMSs are an order of magnitude faster compared to RDF DMSs, since they do not have to execute multiple inner joins in addition to the aggregation operations. Moreover, for star-shaped queries (queries with bushy plans having 5 or more triple patterns, a filter and 4 or more projection variables) Gremlin traversals outperform their SPARQL counterparts by at least one order of magnitude for S1, S2 and at least two orders of magnitudes for S3 (with 10 BGPs, 1 filter and 9 projection variables).
2. **Warm cache:** SPARQL queries reap the most benefits of warm caching in RDF DMSs compared to the Gremlin traversals in Graph DMSs. We observe that on average, in this setting, the performance gain is up to 1-1.8 times for star-shaped and mixed queries, 2-3 times for aggregation (counts), condition (filter) and re-ordering (order by, group by) queries, and 3-5 times for CGPs and union queries. On the other hand, as can be expected, Gremlin traversals benefit little from warm caching. On average, in this setting, the performance gain is up to 1.3 times for aggregation (count, group count) and star-shaped queries; up to 1.5 times for re-ordering (order-by, group-by) and condition (filters) queries; up to 2 times for mixed, union and restriction (limit) queries.

Graph DMSs with indexing scheme

We created composite indices for each Graph DMS on attributes such as "name", "customerId", "unitPrice", "unitsInStock", and "unitsOnOrder" for the BSBM dataset. Similarly, on "type", "productID", "reviewerID", and "productTypeID" for the Northwind dataset. The indices use the hash-map data structure. We did not re-execute SPARQL queries on RDF DMSs, as there was no change in their indexing setting.



Figure 5.7: Performance comparison of SPARQL queries vs the translated Gremlin traversals for BSBM dataset with respect to RDF and Graph DMSs in different configuration settings.



Figure 5.8: Performance comparison of SPARQL queries vs the translated Gremlin traversals for Northwind dataset with respect to RDF and Graph DMSs in different configuration settings.

1. **Cold cache:** Gremlin traversals perform significantly faster when executed on Graph DMSs with composite indices. We observe that, as compared to the previous (cold cache + without index) setting, the improvement on an average is 1-2 times for union, mixed and group by traversals; 2-3 times for re-ordering (group-by, order-by) traversals; 3-5 times for regular and restriction traversals; and >5 times for aggregation and star-shaped traversals.
2. **Warm cache:** In this setting the Graph DMSs (i.e. Gremlin traversals) register similar performance gain pattern with respect to the RDF DMSs in the non-indexed configuration.

5.5.4 Discussion

In this section, we present a discussion about the query execution performance with respect to some factors influencing the outcome of a particular DMS:

Query typology: We observe that for –

- (1) Simple/Linear queries (such as C1-3, F1-3, L1-3) the SPARQL and Gremlin traversal performance is comparable;
- (2) SPARQL outperforms corresponding Gremlin traversals for union queries. This is because, in SPARQL a union occurs between two or more sets of triple patterns. Whereas in the declarative construct (pattern matching) of Gremlin, a union occurs between two `.match()`-steps (i.e. Gremlin treats each `.match()`-step as a distinct traversal and then executes a union on top of it); However, in the imperative construct of Gremlin, this is not the case, since we do not "need" to use the `.match()`-step inside a `union()`-step.
- (3) Whereas, for complex queries (such as star-shaped and aggregation queries), Gremlin traversals outperform their SPARQL counterparts (due to the absence of expensive joins).
- (4) Lastly, for queries with higher number of projection variables (greater 2) and query modifiers (count, distinct, limit + offset, filter), Gremlin traversals show a performance gain of more than an order of magnitude compared to the corresponding SPARQL queries (e.g. F1, F2, O2, S1, S2, S3). This advantage, is not as pronounced when comparing queries with a fewer number of projection variables and query modifiers.

Query caching – (cold vs warm): Both RDF and Graph DMSs perform better when using warm caching. We observe that SPARQL queries perform better compared to the corresponding Gremlin traversals. One reason for this is that Gremlin traversals perform overall considerably better (except in cases of union queries) by leveraging the locality advantage of the underlying property graph data model and cannot be optimized further without explicitly creating additional composite indices. Out of all three RDF DMSs, Jena gains most from a warm cache, e.g. up to 5 times performance increase in cases such as union and CGP queries.

Indexing scheme: The one-sided dominance of Virtuoso among all the evaluated RDF DMSs is extremely noticeable. This is because, Virtuoso maintains a variety of full and partial indices (as mentioned earlier). Moreover, we also observed that Virtuoso employs custom partition clustering and caching schemes on top of these indices to provide an adaptable solution to all kinds of workloads. The indices in Virtuoso are column-wise by default²¹, which take only one-third of the space compared to row-wise indices. On the contrary, other RDF DMSs such as 4Store and JenaTDB do not use such mechanisms. Graph DMSs, have limited options, in terms of underlying indexing data structure implementation, for creating manual indexes in a given version. One reason is that, there has not been an explicit need for using complex index schemes (as in Virtuoso), since composite indices based on B+ trees and hash-maps provide sufficient performance for traversal operations.

Thus, based on these findings, we can summarize that for complex queries (such as aggregation, star-shaped, and queries with higher number of projection variables + query modifiers) the corresponding Gremlin pattern matching traversals outperform SPARQL queries. Whereas, for union queries, SPARQL has a significant performance advantage.

5.6 Gremlinator as a Reusable Resource

For a first-hand system demonstration of the Apache TinkerPop `sparql-gremlin` plugin (i.e. GREMLINATOR approach) we have prepared – (a) a video tutorial,²² and (b) a web application²³ for this purpose. (c) a desktop application of GREMLINATOR (standalone .jar bundle) which requires Java 1.8 JRE installed on the corresponding host machine, downloadable from the web demo website.

5.6.1 Technical Quality

Since `sparql-gremlin` plugin is a part of the Apache foundation, community software development best practices were followed to ensure quality and resuability. A few of the best practices followed are – (i) Apache Maven was used as the project management framework. The respective maven artifact is deployed at (<https://search.maven.org/artifact/org.apache.tinkerpop/sparql-gremlin/3.4.1/jar>); (ii) Extensive Unit Tests covering a wide variety of test cases were implemented; (iii) Automated Tests and Build of the software was conducted using the Travis CI API²⁴, and (iv) Reference and Code documentations were created using *Javadocs* (cf. Sections 5.6.2 and 5.6.3).

²¹ Indexing scheme used in Openlink Virtuoso (<http://docs.openlinksw.com/virtuoso/rdfperfrdfscheme/>)

²² Gremlinator Demo Video – <https://www.youtube.com/watch?v=Z0ETx2IBamw>

²³ Gremlinator Web Demo – Mirror 1 <http://gremlinator.iai.uni-bonn.de:8080/Demo> and Mirror 2 <http://195.201.31.31:8080/Demo/>

²⁴ Travis CI API (<https://docs.travis-ci.com/api/>)

5.6.2 Availability

The availability of the resource, licensing and source code information is clubbed in two groups, for the – (i) *Apache TinkerPop sparql-gremlin plugin*²⁵, and (ii) *Independent sparql-gremlin implementation*²⁶ respectively. Both the source codes are available under the Apache License 2.0²⁷

5.6.3 Reusability and Maintenance

In order to ensure ease of reusability of `sparql-gremlin`, we have provided an illustrative and user-friendly documentation in the following manner:

- Apache TinkerPop reference documentation²⁸ – explains the working of the `sparql-gremlin` plugin and other technical details about its installation, use, etc. in the TinkerPop framework;
- Independent implementation documentation²⁹ – which is the independent source code of the proposed `sparql-gremlin` translation, which enables easy adoption and extension of our work, for custom use-cases. For instance, the re-use of our work by IBM Research AI [202], to integrate the `sparql-gremlin` translator within their query engine.

5.7 Community Adoption (Use Cases)

Our GREMLINATOR approach and its Apache TinkerPop `sparql-gremlin` is gaining attention and adoption by both the academia and industry communities. We report few such use cases next.

5.7.1 IBM Research AI use case

In the recent research study [202] published by IBM Research, GREMLINATOR has been extended and reused in order to support scalable reasoning over large scale Knowledge Graphs targeted towards industrial applications. The SPARQL to Gremlin translation is embedded in the *query layer* in order to execute SPARQL queries over the property graph data stored in JanusGraph³⁰. The authors have compared the performance of executing SPARQL queries over the Openlink Virtuoso triplestore to that of executing the translated Gremlin traversals over JanusGraph.

²⁵ <https://github.com/apache/tinkerpop/tree/master/sparql-gremlin>

²⁶ <https://github.com/LITMUS-Benchmark-Suite/sparql-to-gremlin>

²⁷ <http://www.apache.org/licenses/LICENSE-2.0>

²⁸ TinkerPop reference docs (<http://tinkerpop.apache.org/docs/current/reference/#sparql-gremlin>)

²⁹ Independent Github repository (<https://github.com/LITMUS-Benchmark-Suite/sparql-to-gremlin/blob/master/README.md>)

³⁰ JanusGraph (<https://janusgraph.org/>)

They report a similar performance trend of Gremlin traversals as compared to SPARQL queries using the LUBM³¹ dataset for 100 universities to that reported in our study.

5.7.2 SANSA Stack use case

The Scalable Semantic Analytics (SANSA) Stack [203] exercises distributed computing via Apache Spark and Flink in order to enable scalable machine learning, inference and querying capabilities for large knowledge graphs. GREMLINATOR is employed in the *query layer* of the SANSA Stack version 0.3³² as an experimental feature to allow executing SPARQL queries over the Apache Spark and Apache Flink using Gremlin traversals. The maven artifact is available from – <https://mvnrepository.com/artifact/io.github.litmus-benchmark-suite/gremlinator/1.0/usages>.

5.7.3 Contextualised Knowledge Graph use case

This use case is about adding a semantic web abstraction layer on top of Graph databases by employing GREMLINATOR for querying a Contextualised Knowledge Graph (CKG). This project aims to simulate PG-style characteristics (e.g. node and edge properties) to RDF KGs via extending the singleton property semantics [204]. The applications of the CKG model are aimed towards curating an Open Knowledge Network (OKN) infrastructure for the storage and querying of biomedical data related to patients' health. This use case is currently under development.

5.7.4 Open Research Knowledge Graph use case

The European Research Council funded project ScienceGRAPH proposes the development of an Open Research Knowledge Graph [103], which is will be the first Knowledge Graph based infrastructure that for the publication and consumption of scholarly data. This preliminary study envisions to extend GREMLINATOR in order to execute SPARQL queries over a large scholarly communication based Knowledge Graph stored in Neo4j graph database.

5.8 Summary

In this chapter, we presented GREMLINATOR, a novel method and a tool for executing SPARQL queries over property graph databases using Gremlin pattern matching traversals. With GREMLINATOR we lay a substantial step in order to bridge the *query interoperability* gap between the RDF and Property graph DMSs, concerning both the semantic web and graph databases communities. GREMLINATOR has been successfully integrated as a (`sparql-gremlin`)

³¹ LUBM dataset (<http://swat.cse.lehigh.edu/projects/lubm/>)

³² First release (<https://github.com/SANSA-Stack/SANSA-Stack/releases/tag/2017-12>), Changelog (<http://sansa-stack.net/sansa-0-3/>)

plugin of the popular Apache TinkerPop project, and is also freely available as an independent resource for promoting reuse and extension over custom use cases. We have also presented a comprehensive empirical evaluation of our approach, using state-of-the-art RDF and Graph DMSs, demonstrating the validity and applicability of our approach. The evaluation demonstrates the substantial performance gain obtained by translating SPARQL queries to Gremlin traversals, specifically for star-shaped and complex queries. Finally, we discuss the attention GREMLINATOR has gained from both academia and industry research fraternities so far, in the form of use cases which further demonstrate the applicability and validity of our contribution.

Automatic Benchmarking of RDF and Graph Databases

Over the last few years, the amount and availability of Open, Linked and Big data on the web has increased. Simultaneously, there has been an emergence of a number of Data Management Solutions¹ (DMSs) to deal with the increased amounts of structured data. The available DMSs for graph structured data can be broadly divided in two categories on the basis of the data model they address: 1.) *Triple Stores*, which use the RDF data model and 2.) *Graph Databases*, which use the PG data model. Apart from the format of the dataset that they consume, there are several other differences in the manner in which they build indexes and execute queries.

In order to objectively decide which DMS are suitable with respect to particular scenarios, benchmarks involving particular query loads over characteristic datasets have been defined. Some of the existing benchmarking tools have their own dataset generators and corresponding queries to run on these datasets [166, 168]. Furthermore, given that benchmarking is an extremely tedious task demanding repetitive manual effort, therefore it is advantageous to automate the whole process. However, none of the tools allow the users to benchmark both of the above mentioned categories of graph DMSs, i.e. triple stores and graph databases, in an automatic unified and comparable manner. We argue that the increasing number of available DMSs in both categories necessitates a benchmarking tool which is sufficiently versatile to perform those benchmarks.

In this chapter we address the following third research question (RQ3):

RQ3: Benchmarking – How can we seamlessly orchestrate the benchmarking of RDF vs Property graph Databases in an open, extensible, fair and reproducible manner?

The main contribution of this chapter is LITMUS BENCHMARK SUITE, which is the first framework, that is able to benchmark both RDF and Property graph databases. To this end LITMUS is a comprehensive framework, that allows academicians, researchers, DMS developers and the organizations a choke-point driven performance comparison and analysis of various

¹ In the context of this chapter we use the terms Databases and Data Management Systems interchangeably.

DMSs (Graph and RDF-based), with respect to different third-party real and synthetic datasets and queries. LITMUS provides fine-grained environment configuration options, a comprehensive set of system and data-specific key performance indicators and a quick analytical support via custom visualization (i.e. plots) for the benchmarked DMSs. Along with this, there are also other side contributions related to the study of the influence of factors such as data quality, system and data-specific metrics which have been formally published.

This chapter is based on the following publications [35, 45, 47]:

1. **Harsh Thakkar**, Yashwant Keswani, Mohnish Dubey, Jens Lehmann, and Sören Auer. *Trying Not to Die Benchmarking – Orchestrating RDF and Graph Data Management Solution Benchmarks using LITMUS*. In Proceedings of the 13th International Conference on Semantic Systems (SEMANTiCS 2017), Amsterdam, Nederland, pages 120-127. ACM, 2017. [**Best Paper Award**]
2. **Harsh Thakkar**. *Towards an Open Extensible Framework for Empirical Benchmarking of Data Management Solutions: LITMUS*. In Proceedings of the 14th Extended Semantic Web Conferences (ESWC 2017), 2017.
3. **Harsh Thakkar**, Kemele M. Endris, Josè M. Gimenez-García, Jeremy Debattista, Christoph Lange, and Sören Auer. *Are Linked Datasets Fit for Open-domain Question Answering? A Quality Assessment*. In Proceedings of the 6th International Conference on Web Intelligence, Mining and Semantics (WIMS 2016), Nîmes, France, June 13-15, pages 1-12, 2016.

The remainder of this chapter is structured into five sections, starting with Section 6.1 which presents the architecture of the proposed framework and discusses the role of each of the four components within. Section 6.2 sheds light on the implementation methodology, the integrated datasets, databases systems and supported queries. Section 6.3 discusses the selected metric, parameters and KPIs which have been implemented to facilitate a detailed analysis of the system and data level results of the benchmarks. Section 6.4 presents the findings of the comprehensive experimental evaluation demonstrating the benchmarking of three top of the line RDF and Property graph DMS each and discusses its limitations. Finally, Section 6.5 concludes the chapter summarizing the contributions and the future direction.

6.1 LITMUS Framework Architecture

In this section, we present the conceptual architecture of the LITMUS BENCHMARK SUITE. It comprises of four major facets: Data Facet (F1), Query Facet (F2), System Facet (F3), and Benchmarking Core (F4) (ref. Figure 6.1). The role of each facet is as follows:

The complete source is well documented and made available publicly². The first prototype of LITMUS framework (v0.1) is released on Docker Hub platform³ for encouraging first hand

² LITMUS Benchmark Suite – <https://github.com/LITMUS-Benchmark-Suite/>

³ LITMUS docker – <https://hub.docker.com/r/litmusbenchmarksuite/litmus/>

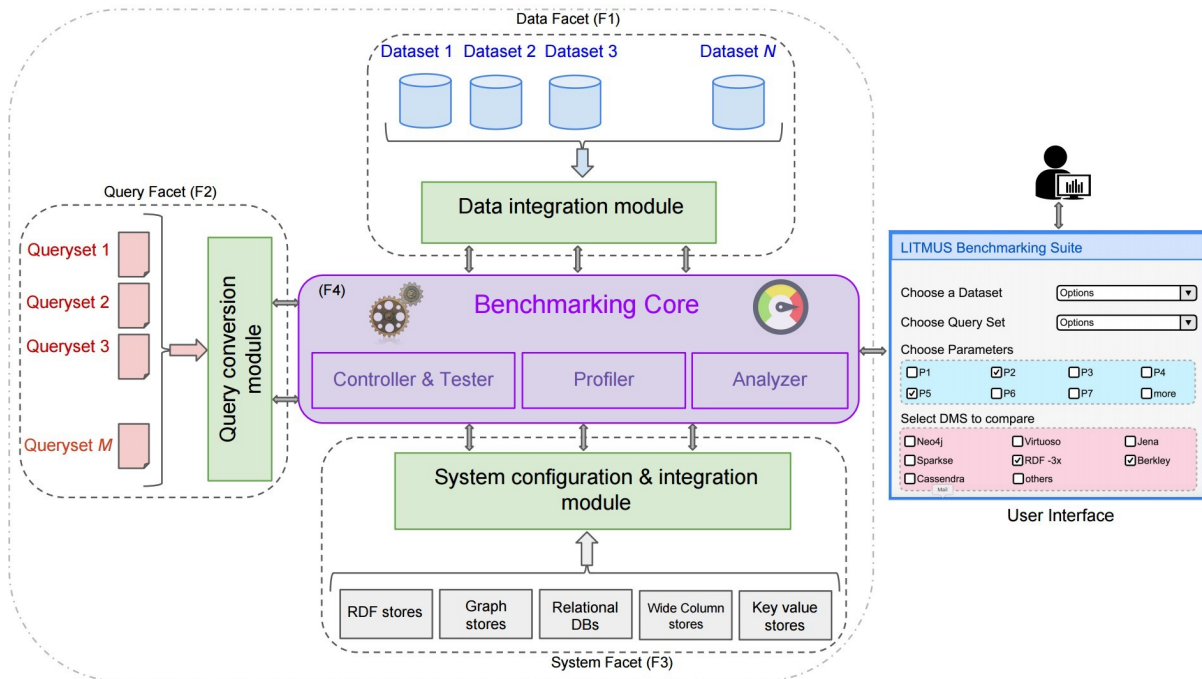


Figure 6.1: The architectural overview of the LITMUS BENCHMARK SUITE.

experience and user feedback.

6.1.1 Data Facet

The Data Facet consists of the (i) Dataset(s) and the (ii) Data Integration Module. Datasets chosen for benchmarking can be real datasets such as DBpedia⁴, Wikidata⁵, synthetic datasets such as the Berlin SPARQL Benchmarking (BSBM) [205], Waterloo SPARQL Diversity Test Suite (WatDiv) [206], or hybrid datasets comprising both real and synthetic data. The *Data Integration Module* is responsible for (a) making data available to the system in the requested formats (such as N-Triples, Graphs, CSV, SQL) by carrying out appropriate data conversion and mapping tasks (cf. Challenge **C1**), and (b) loading the desired format of data to the respective DMSs selected for the benchmark.

6.1.2 Query Facet

The Query Facet comprises of the (i) Queryset(s), and the (ii) Query Conversion Module. The *Queryset* refers to the set of query input files. The *Query Conversion Module* will be one of the key components addressing the language barrier (Challenge **C2**). It is responsible for converting the input SPARQL queries to the respective DMSs' query languages (such as Gremlin, SQL, etc). The conversion will be performed by developing an intermediate language/logic representation

⁴ <http://wiki.dbpedia.org/>

⁵ <https://www.wikidata.org/>

of the input query. The aim of this module is to allow efficient conversion of a wide variety of SPARQL queries (such as path, star-shaped, and snowflake queries) to other query languages, ultimately breaking the language barrier.

6.1.3 System Facet (DMS Facet)

The System Facet consists of (i) DMSs and (ii) DMS Configuration and Integration module. The *DMS Configuration and Integration* module is responsible for (i) providing easy integration, via wrapper(s) or as a plug-in, of the DMSs, and (ii) monitoring and configuring the integrated DMSs for the benchmark. On top of this, this module makes use of *Docker containers*⁶ to ensure a fair allocation of resources and to provide the necessary segregation required for conducting realistic benchmarks.

6.1.4 Benchmarking Core

The Benchmarking Core is the heart of the LITMUS framework, consisting of three modules: (i) Controller and Tester, (ii) Profiler, and (iii) Analyser. The *Controller and Tester* is responsible for executing the respective scripts for loading data, fetching the queries to their corresponding DMSs, validating the specified system configurations, and finally, executing the benchmark on the selected setting (i.e. executing respective SPARQL and Gremlin queries against RDF and Graph DMSs). The *Profiler* is responsible for: (a) generating and loading various profiles (stress loads, query variations, etc.) for conducting the benchmark tests and (b) storing the custom benchmark results. We use the `matplotlib`⁷ python library is used for generating box-plots of benchmark results, where as statistical analysis is carried out using the the `Pandas` python library, to calculate various parameters, e.g. arithmetic mean, median, standard deviation etc. The *Analyser* is responsible for collecting the benchmark results from the *Profiler* and generates performance reports. It performs correlation analysis between the parameters specified by the user. The final results (reports) will then presented to the end user in a suitable visualisation.

6.2 The LITMUS Environment

In this section we discuss the implementation details of the first working prototype of the LITMUS BENCHMARK SUITE. The proposed LITMUS framework consists of a number of plug-n-play modules, which ensures interoperability and extensibility of future DMSs and datasets in the existing infrastructure. Figure 6.2 presents the architecture of the first working prototype of LITMUS, showcasing the interaction between its various constituent components/modules.

GUI module: provides a graphical user interface (GUI) for the user to allow easy configuration of the benchmark to be executed. It allows the user to select from the integrated DMSs, datasets, queries and KPIs, and save, import, export configurations, results, etc. artifacts

⁶ Docker technology – <https://www.docker.com/>

⁷ Matplotlib library – <https://matplotlib.org/>

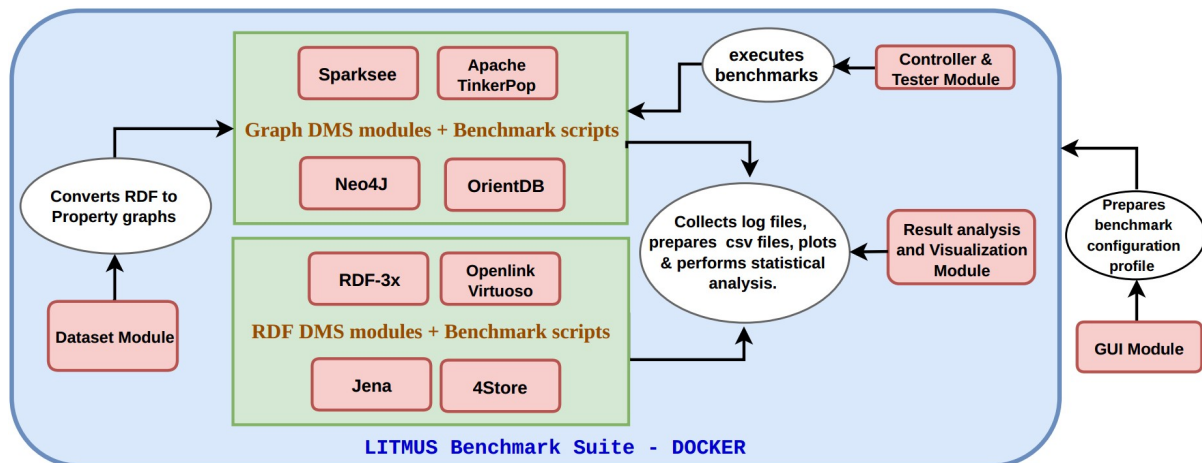


Figure 6.2: The component driven architecture of the first working prototype of LITMUS BENCHMARK SUITE.

produced during the benchmark. Figure 6.3 shows a screen of the GUI module. The GUI has been created using the Tkinter library⁸ in Python for the users. This GUI operates on top of the Docker image (described next). It allows to do away with the Command Line arguments, and allows them to save benchmarking configurations, and load them subsequently.

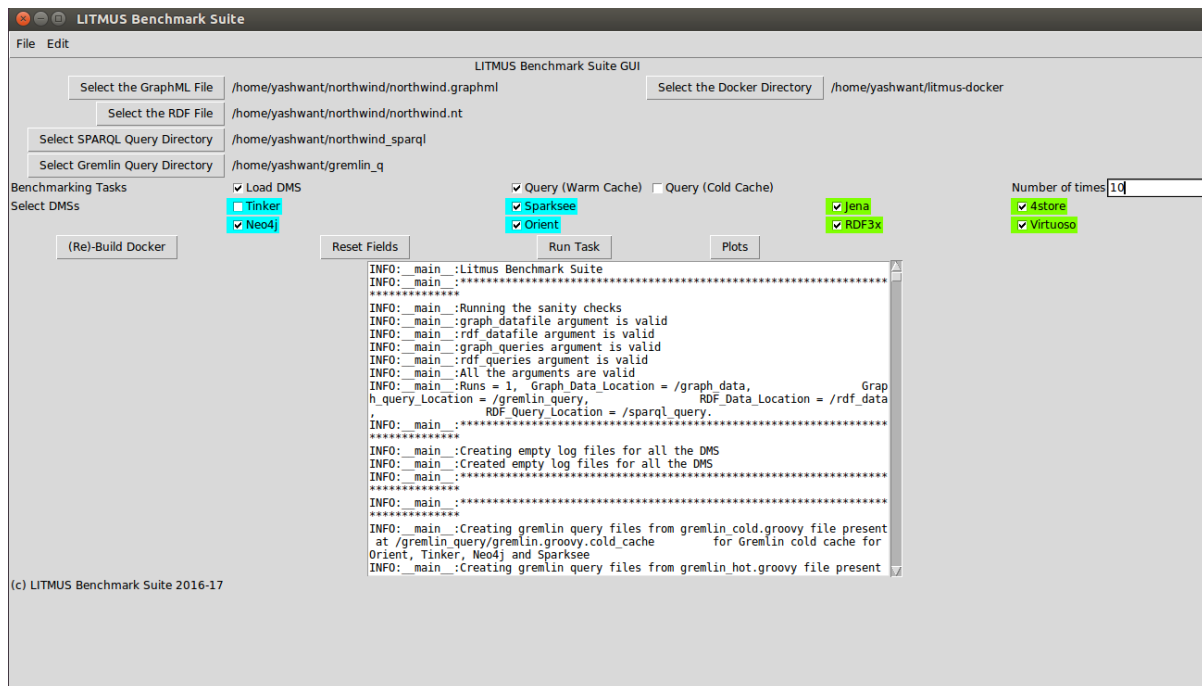


Figure 6.3: The graphical user interface of the LITMUS BENCHMARK SUITE.

LITMUS docker: The whole framework is encapsulated in a single configurable docker container, to ensure necessary isolation during the benchmarking process (cf. Figure 6.2). The

⁸ Tkinter python library – <https://docs.python.org/3/library/tk.html>

main advantage of running a software as a Docker container is that it decouples the application code from the machine on which the code is being run on. This simplifies the process of running the framework, as the only prerequisite of running this framework is a machine which is capable of running a docker container. The user does not have to undergo the process of installing individual modules and DMSs on the machine. We now describe in brief the cultivated environment for LITMUS, consisting of datasets, DMSs, queries and the benchmark environment configuration supported in its current release.

6.2.1 Integrated Datasets

LITMUS framework currently provides support for benchmarking RDF DMSs (which are RDF stores), and Graph DMSs (which are Graph stores) using corresponding versions of Linked data (in RDF and Graph formats) against a set of corresponding queries (SPARQL and Gremlin). We list the datasets that were converted from the RDF graphs to Property graphs (PGs), to ensure a uniform and fair benchmarking process.

In the current version of LITMUS, we do not consider the semantics of blank RDF nodes (as in DBpedia and Wikidata) during the conversion of RDF graphs to PGs. Addressing these underlying semantics of RDF graphs, requires an in-depth study of information preserving techniques. We provide a proof-of-concept implementation for transforming RDF graphs (in this case directed edge-labelled multi-graphs) [.nt files] to PGs (directed, edge-labeled, attributed multi-graphs) [.graphml files].

We use the same approach proposed in Chapter 4, in order to transform the RDF data into property graph data. We present a short summary of employed conversion logic. Please note that since none of the selected datasets have blank nodes and is only instance data (no explicit schema is made available in terms of an .owl file or otherwise), the conceptualisation of the conversion presented next is relatively straight-forward. Our proposed approach (cf. Chapter 4), however supports conversion of complex RDF data as well.

Given that a RDF triple consists of $\{s \ p \ o.\}$, each of the S, P are IRIs and O can be either a IRI or a literal/value. We distinguish between two types of RDF triples as: (i) attribute triple– if the object is a literal; (ii) relationship triple– if it is a URI. Attribute triples correspond to properties in a PG, and relationship triples to edges. Furthermore, predicates $\{s \ p \ o \}$ can be URI, which can be labels (rdfs:label), types (rdfs:type), etc in a RDF graph. Depending on the type of a predicate, we distinguish whether the properties are of edges or nodes in a PG. We point the interested reader to [151], for a detailed understanding and illustration of the RDF \rightarrow PG transformation.

Berlin SPARQL Benchmark [168] (BSBM) – is a synthetic dataset built around an e-commerce use case, where a set of products is offered by different vendors and different consumers have posted reviews about products. BSBM offers custom dataset and query generator scripts, which can be used to generate datasets and queries of varying size and complexity. We provide generated RDF data (.nt file) and converted PGs (.graphml file) (using custom scripts) for 1M and 10M triples with the v0.1 of LITMUS.

Waterloo SPARQL Diversity Test Suite [166] (WatDiv) – is a synthetic dataset which is again based on the e-commerce use case scenario, however the distinct characteristic that all instances of the same entity have mixed number of attributes. The WatDiv has a dataset generator, uses a configuration file which is written using the WatDiv DDL to define certain parameters. The dataset generator tool generates the dataset using the values of the underlying parameters defined in the configuration files in the nt format. We provide generated RDF data (.nt files, using its data generation script) and converted PGs (.graphml files) (using custom scripts) for 1M and 10M triples with the v0.1 of LITMUS.

DBpedia [52] – is a crowd-sourced community effort to extract structured information from Wikipedia and make this information available on the Web. The DBpedia dump consists of multiple files in the *ttr* format. We provide a proof-of-concept property graph of DBpedia. However, we do not benchmark it since it consists blank node semantics, which are not currently supported by LITMUS v0.1 framework. The script developed for the conversion can be found here⁹.

Northwind¹⁰ – is a synthetic dataset describing an ecommerce scenario about the sales and purchase transactions that happen between the company Northwind Traders and its customers and suppliers respectively. The Northwind Dataset was originally shipped in form of csv files, however there are certain RDF versions available as well. We provide both RDF data (.nt files) and converted property graph data (.graphml files) with LITMUS.

6.2.2 Integrated DMSs

LITMUS currently provides support for benchmarking eight DMSs (four each of RDF & Graph DMS), as listed below:

RDF DMSs The following RDF DMS can be evaluated in the LITMUS framework:

1. Openlink Virtuoso¹¹
2. gh-RDF-3x¹²
3. Apache Jena TDB¹³
4. 4Store¹⁴

For each of the RDF DMS, LITMUS includes two shell scripts for – (i) benchmarking the process of loading a RDF dataset in a RDF DMS; and (ii) benchmarking the SPARQL query execution

⁹ DBpedia Property graph converter <https://github.com/LITMUS-Benchmark-Suite/dbpedia-graph-convertor>

¹⁰ Northwind Database <https://northwinddatabase.codeplex.com/>

¹¹ Openlink Virtuoso – <https://virtuoso.openlinksw.com/>

¹² RDF-3X – <https://github.com/gh-rdf3x/gh-rdf3x>

¹³ Apache Jena – <https://Jena.apache.org/>

¹⁴ 4Store DMS – <http://www.4store.org/>

process against a DMS. We employ the *elapsed time* parameter of the `’/usr/bin/time’` utility has been used to measure the execution time for both the tasks for all the RDF DMSs.

Graph DMSs The following Graph DMS can be benchmarked in the current release of LITMUS:

1. Sparksee (formerly known as DEX Graph)¹⁵
2. Neo4j¹⁶
3. OrientDB¹⁷
4. Apache TinkerPop¹⁸

For each Graph DMS, LITMUS includes four scripts (2 shell scripts, 2 groovy scripts) which are used for – (i) benchmarking the process of loading a Graph dataset in a Graph DMS, using the Gremlin Groovy console; and (ii) benchmarking the Gremlin Query execution against a Graph DMS, using the Gremlin Groovy console. The execution time for both the tasks has been measured using the `’System.currentTimemillisecs()’` Groovy function.

There is a central Python script, which manages the execution of all the DMSs inside the Docker which the user has selected to evaluate by calling the respective shell-scripts. The functions for collecting the data and generating CSV files, after the benchmarking tasks have been run on the DMSs are also defined in this Python script. There is a log file associated with the Python script which can be used to track the exact status of the benchmarking process at any moment.

6.2.3 Supported Queries

To demonstrate a benchmark using LITMUS, we curated a query dataset including both SPARQL and Gremlin queries, as proposed in Section 5.4.4 of Chapter 5. We summarize the query features as summarized in Table 6.1. A total number of 30 SPARQL queries were created (3 of each query feature) for each RDF dataset. We created their corresponding Gremlin counterparts manually for each Graph dataset. The queries are executed using both warm and cold cache settings, where a *warm cache*: implies that the cache is not cleared after each query run, and *cold cache*: implies that the cache is cleared using the `’echo 3 > /proc/sys/vm/drop_caches’` unix command after each query. While running a query, the DMSs compute intermediate results and store them in the memory. These intermediate results may be useful for the next query which is run on the system. Running the queries in two configurations allow the users to study the correlation between performance of the DMSs with respect to query, dataset-specific characteristics, and the order in which they are run. Additionally, the influence

¹⁵ Sparksee Technologies – <http://www.sparsity-technologies.com/>

¹⁶ Neo4J – <https://neo4j.com/>

¹⁷ Orient DB – <http://orientdb.com/>

¹⁸ Apache TinkerPop – <http://tinkerpop.apache.org/>

Query No.	Feature	Count	Description
C1-C3	CGPs	3	Queries with mixed number of basic graph patterns (BGPs)
F1-F3	FILTER	3	CGPs with a combination of ≥ 1 FILTER
L1-L3	LIMIT+OFFSET	3	CGPs with a combination of ≥ 1 LIMIT constraints
G1-G3	GROUP BY	3	CGPs with GROUP BY feature
Gc1-Gc3	GROUP COUNT	3	CGPs with GROUP BY + COUNT
O1-O3	ORDER BY	3	CGPs with ORDER BY feature
U1-U3	UNION	3	CGPs with UNION feature
Op1-Op3	OPTIONAL	3	CGPs with a OPTIONAL BGPs
M1-M3	MIX	3	CGPs with a combination of varying features
S1-S3	STAR	3	CGPs forming a STAR shape execution plan
TOTAL	10	30	-

Table 6.1: Feature distribution in preset queries provided with the LITMUS BENCHMARK SUITE.

of factors like the query length, query size, Graph patterns on the performance of the system can be seen when run in the queries are run in warm cache configuration.

6.2.4 Execution Environment

In order to have a fair comparison, it is highly critical to ensure that all DMSs run under identical conditions for eliminating any bias towards a specific run, and avoiding any inconsistencies and anomalies observed in results. However, the inherent non-deterministic factors e.g., OS scheduling, context switches, and interrupts are beyond a user's control. As a result the following set of rules are followed to ensure a fair evaluation procedure.

1. Each query execution task is carried out individually and is ran several times (default: 10 times, user can define this before-hand) for each DMS to nullify the effect of anomalies.
2. Every run of the task is run in isolation. No other unnecessary process(es) is running in the background during the benchmark.
3. Each dataset loading task makes use of a new location for every run. This ensures that no run is getting an undue advantage of an already existing set of files.

Result Logging and Visualization. Separate CSV files are created after parsing all the log files which are generated when the benchmarking tasks on various DMSs are done. Statistical Analysis is done on the generated CSV files, using the `Pandas` library in Python, to calculate various statistical parameters like the arithmetic mean, median, standard deviation etc. This data is exported in the form of individual tables for each parameter. The data is visualized using multiple boxplots in a single plot using the `matplotlib` [207] module.

6.3 Performance Evaluation

LITMUS caters a wide variety of performance evaluation parameters and metrics to allow an in-depth analysis of underlying internal and external factors of a DMS.

6.3.1 Selected Parameters

Perf-tool Utility. LITMUS uses the `perf-tool`¹⁹ utility to measure a variety of CPU and RAM-specific parameters, e.g. L1d-cache-misses, L1i-cache-misses, DTLB-misses, etc. for enabling a comprehensive analysis of the participating DMSs. However, since the number of hardware counters on any machine is limited, a single run is not sufficient to measure the large number of parameters offered by the `perf-tool` utility. We segregate the parameters offered by the `perf-tool` utility into four groups. Both the benchmarking tasks, viz. (i) loading the dataset a DMS, and (ii) executing a query on a DMS, are run separately for each group of parameters. This is done to ensure that the framework is in a position to measure all the parameters even when run on a machine, where the number of hardware counters is low.

One of the major features of the LITMUS framework is to facilitate the users in performing a comprehensive analysis of the DMSs. A wide range of parameters are selected, to ensure that the various factors like the utilization of various levels of cache, the branch predictions, the instructions, the data Translation look-a-side buffers, Page faults, CPU migrations, are considered when a DMS is evaluated, and subsequently compared with the other DMSs. These parameters also enable the users to identify the reason(s) for a superior or inferior performance of any particular DMS.

The detailed analysis using the parameters allow the DMS developers to analyze the DMS they are developing and identify the possible cause of an inferior performance, and thereby work in a direction to rectify the faults. This will eventually lead to a better DMS being developed. We present an itemization of the parameters considered to evaluate a performance of a DMS:

1. *Cycles* : The number of cycles taken to execute a task (e.g. loading a dataset, etc.).
2. *Instructions* : The number of instructions executed per given task.
3. *Cache references* : The total number of cache references made during a given task.
4. *Cache misses* : The total number of cache misses occurred during a given task.
5. *Bus cycles* : The number of bus cycles taken during a given task.
6. *L1 data cache loads* : The total number of L1 cache loads that occur during a given task.
7. *L1 data cache load misses* : The total number of L1 data cache load misses that occur during a given task.

¹⁹ Perf tool – https://perf.wiki.kernel.org/index.php/Main_Page

8. *L1 data cache stores* : The L1 data cache stores that occur during a given task.
9. *dTLB loads* : The data translation lookaside buffer (dTLB) loads that occur during a given task.
10. *dTLB load misses* : The dTLB load misses that occur during a given task.
11. *LLC loads* : The Last level Cache (LLC) loads that occur during a given task.
12. *LLC load misses* : The LLC load misses that occur during a given task.
13. *LLC stores* : The LLC stores the pre-fetches that occur during a given task.
14. *Branches* : The total number of branches that are encountered during a given task.
15. *Branch misses* : The total number of branches missed during a given task.
16. *Context switches* : The total context switches that happen during a given task.
17. *CPU migrations* : The CPU migrations that occur during a given task.
18. *Page faults* : The page faults that occur during a given task.

A set of separate log files is generated for each benchmarking task per DMS. These log files are parsed using regular expressions in Python, to obtain the relevant data from them, and the data is stored in CSV files. Pandas, a python data analysis library, is used to process the data and do elementary statistical analysis on the it.

6.3.2 Selected Metrics and KPIs

In this section we discuss about the metrics and Key Performance Indicators (KPIs) that correspond to the performance of a DMS. These include factors that directly influence the performance of a DMSs such as System related metrics and other indirect factors associated with the Data that being consumed by the respective DMSs. Applications based on these graph-based systems (both RDF and Property graph databases) are influenced by these factors which can be observed by the metrics and KPIs presented next.

System related Metrics

Apart from the memory (RAM and cache) and time based KPIs, such as the dataset loading time and query execution time (both warm and cold caches) for each DMS and each query, LITMUS provides a list statistical metrics for result aggregation and analysis. We provide support for computing the **mean** $[\mu]$ (arithmetic, harmonic and geometric), **median** $[\tilde{x}]$, **standard deviation** $[\sigma_x]$, **variance** $[\sigma^2]$, **minimum** $[\min(x)]$ and **maximum** $[\max(x)]$ for all of the above mentioned CPU and memory-specific parameters using the **pandas**²⁰ python data analysis

²⁰ Pandas Data Analysis Library – <http://pandas.pydata.org/>

Dimension	Metric
D1. Availability	1. Estimated De-referenceability
	2. Estimated De-referenceability of Forward Links
	3. No Misreported Content Types
	4. RDF Availability
	5. Endpoint Availability
D2. Interlinking	1. Estimated Interlink Detection
	2. Estimated External link Data Providers
	3. Estimated De-reference Backlinks
D3. Data diversity	1. Human Readable Labelling
	2. Multiple Language Usage
D4. Consistency	3. Ontology Hijacking
	4. Misused OWL Datatype Or Object Properties
D5. Trust and Provenance	1. Proportion of triples with <i>dc:creator</i> or <i>dc:publisher</i> properties
	2. Adoption of PROV-O Provenance
	3. Proportion of provenance statements

Table 6.2: Linked Open Data quality assessment dimensions and metrics relevant to open domain Question Answering systems.

library. Furthermore, we also provide functionality to export all the metrics results, in CSV file (comma separated value) format and \LaTeX -tabular format.

Data related Metrics

Amongst the previously mentioned factors which directly contribute to the performance of a DMS, there are also other indirect factors that affect the performance of the application which uses a particular DMS. Data Quality is one such intrinsic factor. Data quality plays a crucial role in terms of hindering the performance (with respect to precision and recall) of a system in an application such as question answering.

Answering questions spoken or written in natural language (cf. [45] for an overview) is an increasingly popular setting in which graph technology applications for end users consume Web Data to satisfy sophisticated information needs. In such a setting, users expect answers to be correct, or at least relevant, even when they have not phrased their question precisely or when their speech has been recorded with a low fidelity. Thus, to support, for example, query disambiguation, answer retrieval, results ranking, Web Data consumed in such settings therefore has to meet a certain level of quality.

Quality can generally be defined as “fitness for use”, but there are a lot of concrete factors that influence a dataset’s fitness for use such as question answering settings and in specific application domains. In studies such as [45, 46, 208], we thoroughly studied and identified a number of metrics and dimensions related to the quality assessment of Linked Open Datasets that indirectly contribute to the performance of DMSs in a data-centric application such as open domain question answering. In doing so, we have used automatic assessment of linked data quality by benchmarking framework – Luzzu [209].

We evaluated subsets of the Linked Open Datasets such as DBpedia²¹ and Wikidata²² using

²¹ <https://wiki.dbpedia.org/>

²² <https://www.wikidata.org/>

the Luzzu framework and identified the following metrics (shown in Table 6.2) and dimensions to be relevant for the overall quality of the data.

For a detailed insight into the experiments and analysis of the data quality based dimensions and metrics, we point the interested reader to the respective studies [45, 46, 208] and skip it from being included into the current text as it is slightly orthogonal to context of this doctoral thesis. In this chapter, we only focus on the contributions related to LITMUS BENCHMARK SUITE.

6.3.3 Data Visualisation

LITMUS provides automated support for visualizing results of the benchmark using the python `matplotlib` data visualization library in the form of boxplots to ease the process of decision making. The boxplot presents the median value, first quartile, third quartile and the extreme outliers. The Inter Quartile Range (IQR) is defined as the difference of the value at the third quartile and the first quartile. An extreme outlier, is defined as a value which is not in the range (*first quartile* $-1.5 * IQR$, *third quartile* $+1.5 * IQR$). These extreme outliers correspond to the anomalous runs which were executed. We use a two color coding scheme to highlight the difference between RDF DMSs (using green) and Graph DMSs (using blue). A distinct plot is generated for each parameter (as mentioned above), used for each task per DMS.

6.4 Experimental Evaluation

We demonstrate the working of LITMUS to showcase its applicability, functionality and suitability for conducting benchmarks in a user-configured fashion. Keeping in mind the extensive amount of results and plots generated during the benchmark, we list only subset of the complete benchmark results (which includes benchmarking only a few parameters, queries and tasks). The **Benchmarking Tasks** selected are (i) Dataset loading time; and (ii) Query execution (both Warm and Cold Cache) time. The selected **Parameters** are (a) CPU migrations; (b) page-faults; and (c) instructions; for the executed queries. We selected all the *system*-based **Metrics** and **KPIs** as discussed in Section 6.3.2.

In this chapter, we only present the results of benchmarking all DMSs using the **Northwind** dataset. However, a complete set of all the results for both the Northwind and BSBM datasets over all selected parameters and KPIs can be accessed online via the Google Drive folder – <https://drive.google.com/drive/folders/OB3aUtDGGrGadZWNPNzRScn1OY3M>.

6.4.1 System Setup

We used the following configuration for running the LITMUS BENCHMARK SUITE:

- **CPU:** Intel(R) Core(TM) i5-4200M CPU @ 2.50GHz

DMS	G_mean	H_mean	Max	Mean	Median	Min	Var.
4Store	0.89	0.86	5.010	0.97	0.83	0.640	0.45
Jena	8.21	8.21	9.780	8.22	8.16	7.700	0.13
Neo4J	1.48	1.47	2.023	1.49	1.44	1.278	0.031
OrientDB	3.51	3.48	5.612	3.53	3.43	2.832	0.22
RDF3X	0.69	0.68	0.920	0.69	0.66	0.580	0.006
Sparksee	0.72	0.72	0.888	0.72	0.73	0.640	0.001
TinkerGraph	0.61	0.61	1.138	0.62	0.60	0.477	0.010
Virtuoso	0.27	0.27	0.580	<i>0.27</i>	0.27	0.250	0.002

Table 6.3: The loading time (in seconds) performance comparison for Northwind (respective versions) in all the DMSs. The highest and lowest values for mean are shown using the **bold** and *italic* fonts respectively.

- **RAM:** 8 GB DDR3; *L1d & L1i Caches:* 32 KB; *L2 Cache:* 256 KB; *L3 Cache:* 3072 KB
- **RDF DMSs:** Openlink Virtuoso [7.2.5], Apache Jena TDB [3.2.0], 4store [1.1.5], RDF3X
- **Graph DMSs:** Apache TinkerPop [3.2.4], Neo4J [1.9.6], Sparksee [5.1], OrientDB [2.1.3]

Dataset loading time. The reported dataset loading time is in seconds (s) and is the average of 10 data loads for each DMS (of both RDF and Graph).

Query execution time. The reported query execution time is in seconds (s) and is the average of 10 runs for each query (of both SPARQL and its translated Gremlin counterpart).

Caching scheme. Queries were executed in both *cold* and *warm* cache settings for all considered DMSs. Where, a *warm cache*: implies that the cache is not cleared after each query run, and *cold cache*: implies that the cache is cleared using the `'echo 3 > /proc/sys/vm/drop_caches'` UNIX command after each query run.

Indexing scheme. All DMSs were used for benchmarking in their vanilla state (i.e. without any explicit changes in their vendor provided configurations.)

6.4.2 Results and Discussion

We now present and discuss our observations after running the benchmark for all the selected DMSs for a selected number of KPIs using tables and plots generated by LITMUS.

Table 6.3 presents the **loading time** performance comparison of loading Northwind dataset for all DMSs respectively. Here, in terms of dataset loading time, we observe that Virtuoso is the fastest followed by TinkerGraph and the slowest reported time is by Apache Jena.

Furthermore, Tables 6.4 and 6.5 present the **execution time** (both cold and warm cache) performance comparison on *Query 14 (C1)* (shown below, SPARQL listing 6.1, Gremlin listing 6.2) for all DMSs respectively.

DMS	G_mean	H_mean	Max	Mean	Median	Min	Var.
4Store	0.021345	0.020474	0.270	0.026250	0.0200	0.020	0.001562
Jena	0.175700	0.175530	0.199	0.175875	0.1740	0.165	0.000065
Neo4J	0.026553	0.026302	0.046	0.026825	0.0270	0.019	0.000017
OrientDB	0.030312	0.029398	0.155	0.032475	0.0310	0.023	0.000410
RDF3X	0.000000	0.000000	0.030	<i>0.000750</i>	0.0000	0.000	0.000023
Sparksee	0.027483	0.027321	0.045	0.027675	0.0270	0.023	0.000013
TinkerGraph	0.209582	0.203346	0.363	0.216650	0.2075	0.136	0.003564
Virtuoso	0.000000	0.000000	0.026	0.001600	0.0010	0.000	0.000016

Table 6.4: The warm cache execution time (in seconds) performance comparison for running Query 14 (C1) (respective version) on all DMSs. The highest and lowest values for mean are shown using the **bold** and *italic* fonts respectively.

DMS	G_mean	H_mean	Max	Mean	Median	Min	Var.
4Store	4.560672	4.558177	5.050	4.563250	4.510	4.340	2.492506e-02
Jena	0.180934	0.180595	0.200	0.181275	0.179	0.163	1.273327e-04
Neo4J	0.028353	0.027910	0.044	0.028875	0.027	0.021	3.626603e-05
OrientDB	0.051870	0.051043	0.091	0.052875	0.049	0.041	1.329327e-04
RDF3X	0.566901	0.563066	0.730	0.571000	0.540	0.470	5.101538e-03
Sparksee	0.044497	0.044061	0.071	0.044950	0.045	0.034	4.435641e-05
TinkerGraph	0.179499	0.177933	0.258	0.181050	0.187	0.136	5.741000e-04
Virtuoso	0.001278	0.001206	0.003	<i>0.001375</i>	0.001	0.001	3.429487e-07

Table 6.5: The cold cache execution time (in seconds) performance comparison for running Query 14 (C1) (respective version) on all DMSs. The highest and lowest values for mean are shown using the **bold** and *italic* fonts respectively.

Query 14 (C1): “List all the distinct products from the "Beverages" category.”

```

1 SELECT DISTINCT ?name WHERE {
2   ?a <http://northwind.com/model/categoryName> "Beverages" .
3   ?a <http://northwind.com/model/productName> ?name . }
```

Listing 6.1: Query 14 (C1) in SPARQL.

```

1 g.V().match(__.as('a').has("categoryName", "Beverages"), __.as('a').
  values("name").as('name')).select('name').dedup()
```

Listing 6.2: Query 14 (C1) in Gremlin.

Here, we observe that (for query 14): (i) For *warm cache* – RDF3X is the fastest in terms of query execution time, followed by Virtuoso, whereas TinkerPop (tinker) is the slowest; and (ii) For *cold cache* – Virtuoso is the fastest in terms of query execution time, followed by Neo4J, whereas 4Store is the slowest.

The better performance of Virtuoso and RDF3X (RDF DMSs) in terms of query execution can be traced back to the fact that they both inherently maintain implicit indices. The default

indexing scheme²³ in Virtuoso enables it to declare 2 full indices (PSOG, POGS) and 3 partial indices (SP, OP GS) over the RDF graphs. Whereas RDF3X maintains 6 hash-based indices (SPO, POS, OPS, PSO, OSP, SOP) over the RDF graphs giving them an upper edge in terms of performance. In case of TinkerPop (Graph DMS), these indices have to be declared explicitly by the user, depending on their need. Since, we did not explicitly declare any index, hence weaker performance is observed.

Query 20 (Gc2): “Group the products in the "Seafood" category by their total unit price.”

```

1 SELECT (COUNT(?unitPrice) as ?total) WHERE {
2   ?a <http://northwind.com/model/categoryName> "Seafood" .
3   ?a <http://northwind.com/model/unitPrice> ?unitPrice . }
4 GROUP BY(?unitPrice)

```

Listing 6.3: Query 20 (Gc2) in SPARQL.

```

1 g.V().match(___.as('a').has("categoryName", "Seafood"), ___.as('a').
   values("unitPrice").as('unitPrice')).select('unitPrice').
   groupCount()

```

Listing 6.4: Query 20 (Gc2) in Gremlin.

Figures 6.4 to 6.7 present a sample of the plots for CPU migrations for both cold and warm caches, page-faults and number of instructions executed, for Query 20 (Gc2) (SPARQL listing 6.3 and Gremlin listing 6.4), demonstrating the versatility of LITMUS in terms of generating plots of varying details of selected KPIs.

In conclusion, we would like to point out that due to RDF3X’s lack of support for SPARQL queries with aggregation functions and other complex features such as OPTIONAL, UNION, etc., not all results are reported for RDF3X. Therefore, we excluded RDF3X from the final comparison of RDF vs Graph databases for the sake of fairness. We have presented a detailed analysis with respect to the graph data management (loading and conversion of RDF and Graph data) for the selected databases in Chapter 4. Similarly, a detailed analysis with respect to querying the RDF and Graph data stored in the select databases is discussed in Chapter 5, to avoid unnecessary repetition of our observations. In this section, we have only focused on discussing the evaluation of the various RDF and Graph databases supported by LITMUS BENCHMARK SUITE and the extent to which it reports the selected KPIs. Next, we discuss the shortcomings LITMUS.

6.4.3 Limitations

At present, LITMUS does *not* support the following features:

- Federated querying – Multiple sources/endpoints being queried at the same time.

²³ RDF indexing scheme in Virtuoso – <https://virtuoso.openlinksw.com/dataspace/doc/dav/wiki/Main/VirtRDFPerformanceTuning#RDFIndexScheme>

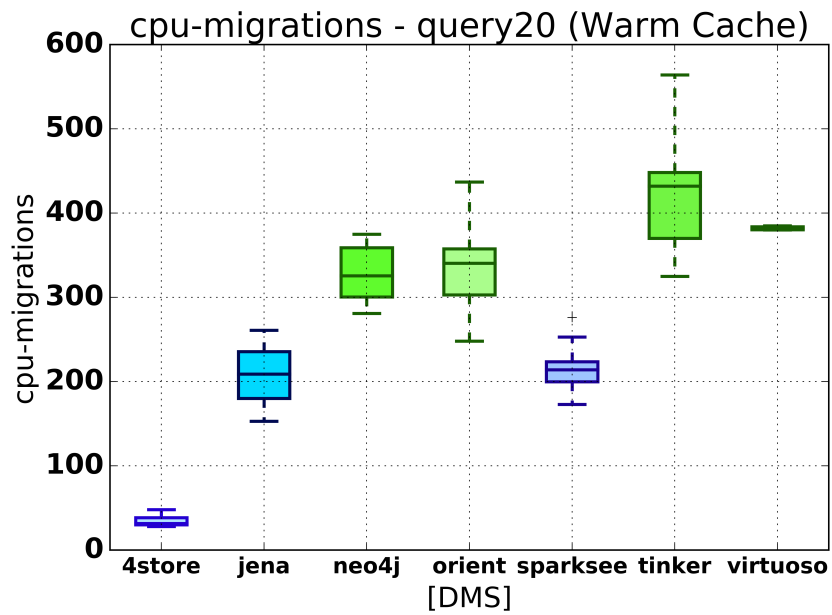


Figure 6.4: CPU Migrations for Query 20 in warm cache.

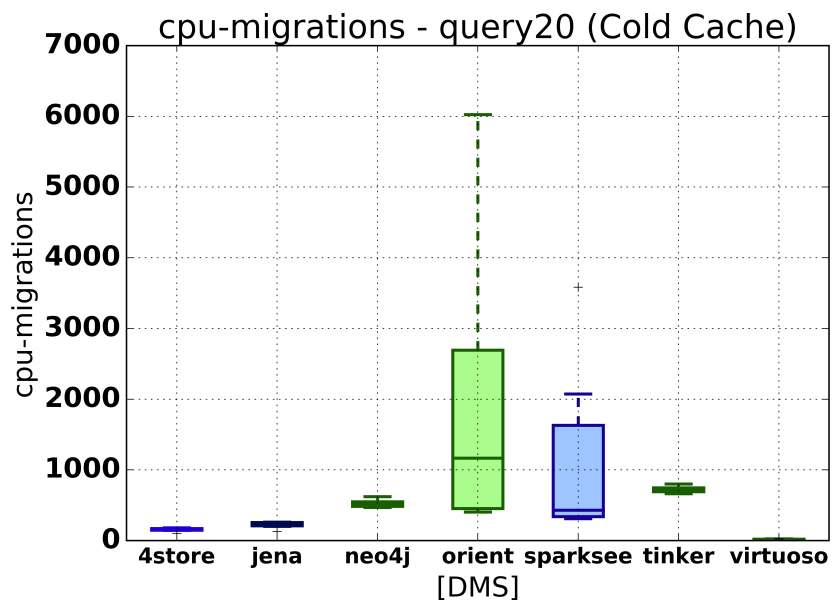


Figure 6.5: CPU Migrations for Query 20 in cold cache.

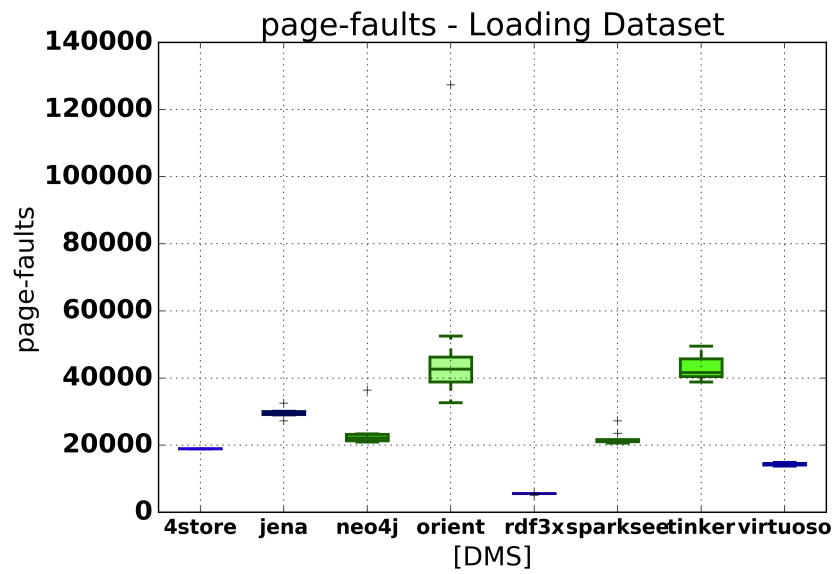


Figure 6.6: Page Faults - loading the Northwind dataset.

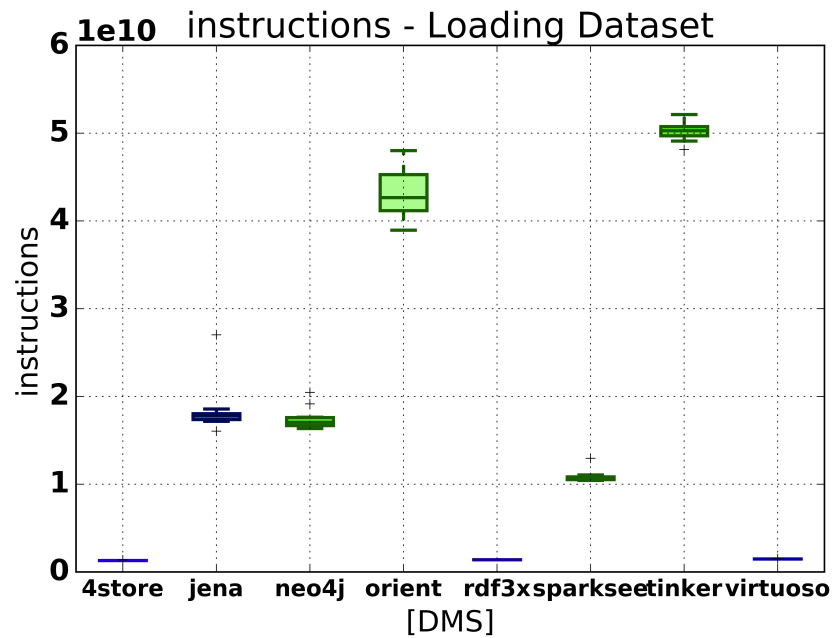


Figure 6.7: Instructions - loading the Northwind dataset.

- Parallel querying & updates – Multiple users/clients querying and updating a single source at the same time.
- Support for generation of synthetic datasets and queries. We "use" other benchmarks that offer such support and leave it up to the user to choose what they deem fit according to their needs.

6.5 Summary

In this chapter, we presented the LITMUS BENCHMARK SUITE, which is a novel framework enabling benchmarking of both RDF and Property graphs via supporting execution of SPARQL queries over graph databases. It also provides support for visualizing results of benchmarked DMSs using custom plots and an easy to use GUI. In its complete capacity, LITMUS is a common platform for benchmarking RDF, Graph and Relational DMSs, promoting easy interoperability, reusability and replicability of existing benchmarks. As compared to other benchmarking efforts, e.g. Graphium [169], LITMUS provides an end-to-end benchmarking solution ensuring full flexibility to user. With LITMUS it is possible to easily orchestrate benchmarking by adding other DMSs and use various real and synthetic data, whereas, the prior is a one time benchmarking effort result. LDBC [170] on the other hand is an established independent authority which leads a community effort towards standardizing Graph DMS benchmarks and also a graph query language. It consists of individual benchmarks such as the social network, graph-analytics and semantic publishing benchmarks respectively. However, to the best of our knowledge, LDBC does not provide an open extensible automated framework such as LITMUS, which can be used for both small and large scale benchmarking appealing both industry and the academia researchers.

Conclusion and Future Directions

Knowledge Graphs (KG) have become popular over the past years and frequently rely on the W3C standard Resource Description Framework (RDF) or Property graphs (PG) as underlying data models. However, these data models and query languages – W3C standard SPARQL for RDF and Gremlin for PGs – severely lack interoperability. The work presented in this dissertation is concerned with addressing the interoperability between RDF and Property graph databases. The overarching research problem this dissertation investigates is:

Overarching Research Problem: How can we support interoperability between the Semantic Web and Property graph Databases?

Answering the overall research problem of this thesis requires addressing the main three challenges related to it. In particular, these three challenges (as discussed in Chapter 1) are the *data interoperability* (cf. Chapter 4), *query interoperability* (cf. Chapter 5), and the performance evaluation by *automatic benchmarking* of RDF and Property graph database systems (cf. Chapter 6) respectively. After identifying the main three challenges, we defined the following three sub-research questions, one for addressing each challenge. As mentioned earlier, collectively, these three sub research questions must be answered to tackle the overarching research problem of this dissertation, as illustrate in Figure 7.1. Next, we summarize how each of these three sub research questions was addressed and our contributions related to them each.

7.1 The Interoperability Story in a Nutshell

First, we tackled the research problem (RQ1), which is concerned with the *data interoperability* between the RDF and Property graph databases.

RQ1: Data Interoperability – How can we directly map RDF Databases to Property Graph Databases in an information preserving manner?

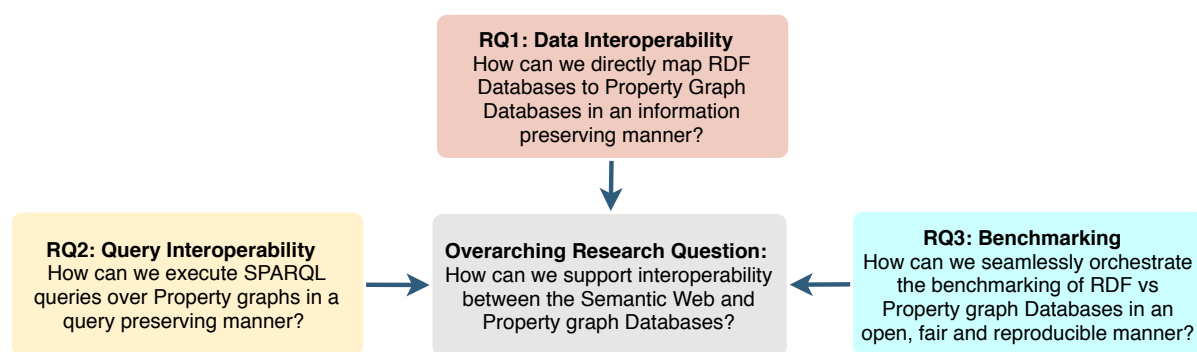


Figure 7.1: The three sub research questions contribute to the overall research objective of this dissertation.

Answering of RQ1 is crucial for this thesis as RDF and Property graphs are two underlying data models that are used to represent, store and query data in the Semantic Web and Graph database systems. While there are some approaches to transforming RDF graphs into property graphs, and vice versa, they lack compatibility and a solid formal foundation. In order to answer RQ1, we first study the notion of “interoperability” in the context of databases (i.e., syntactic and semantic interoperability) [39]; next, we formally define the notions of data and schema for the Property graph data model, after that, we conceptualize the RDF data model and schema in the context of edge-labeled graphs and finally study the desired formal properties that need to be satisfied to transform the data between these two data models [38, 39, 106]. In Chapter 4, we propose three direct mappings (schema-dependent and schema-independent) for transforming an RDF database into a property graph database. We formally show that two of the proposed mappings satisfy the properties of semantics preservation and information preservation [38, 104, 106]. The existence of both mappings allows us to conclude that the property graph data model subsumes the RDF data model’s information capacity. Furthermore, we implement an application consisting of the proposed three formal mappings – RDF2PG. We empirically evaluate our mappings’ performance using a variety of real and synthetic datasets and demonstrate its applicability using an instance of Neo4j graph database system. Thus, with the contributions RQ2, we take a substantial step by laying the core formal foundation for supporting data interoperability between RDF and PG databases. As a result of this work, we have open-sourced the following resources:

- The novel direct mappings implementation RDF2PG is accessible at <https://github.com/renzoar/rdf2pg>.
- The independent implementation of the RDFS processor is accessible at <https://github.com/renzoar/rdfs-processor>.
- The data and queries used in the experiments of this work are accessible at <https://doi.org/10.6084/m9.figshare.12021156.v5> [37].

After that, we tackled the second research question (RQ2) concerned with the *query interoperability* between the RDF and Property graph databases.

RQ2: Query Interoperability – How can we execute SPARQL queries over Property graphs in a query preserving manner?

In order to answer RQ2, we first study the state-of-the-art concerned with the formal foundation of SPARQL and Gremlin, respectively. While there exists comprehensive material [105] on the foundations and expressivity of SPARQL, the same cannot be said for Gremlin. We then, in Chapter 5, study the formal semantics and syntax of the Gremlin traversal language, identify the gaps in the formal foundation and define the missing operators in a consolidated graph relational algebra [40, 210]. We also study the desired formal property that needs to be satisfied to preserve the semantics during the query translation process. Thereafter, we define the mappings and implement GREMLINATOR [41–43], which is a novel approach that translates SPARQL queries to Gremlin pattern matching traversals for querying Property graphs. We present a systematic and comprehensive empirical evaluation of the proposed approach and report interesting insights by investigating the query execution performance over leading RDF and Graph databases on popular datasets [41–43]. Our carefully orchestrated experiments empirically demonstrate that the proposed query translation is query preserving. Thus, with the contributions of RQ2, we take a substantial step by laying the core formal foundation for supporting query interoperability between RDF and PG databases. As a result of this work, we have open-sourced the following resources:

- We deliver an open and independent implementation accessible at <https://github.com/LITMUS-Benchmark-Suite/sparql-to-gremlin> of GREMLINATOR, which can be integrated within custom use cases.
- We deliver ‘sparql-gremlin,’ a plugin of the Apache TinkerPop graph computing framework (version 3.4.0-onwards, first released January 2019) accessible at <https://github.com/apache/tinkerpop/tree/master/sparql-gremlin> that allows querying a variety of both OLTP and OLAP graph systems.
- The complete set of data, queries, and scripts are accessible at <https://doi.org/10.6084/m9.figshare.8187110.v3>.
- We provide an online video tutorial of GREMLINATOR accessible at <https://www.youtube.com/watch?v=ZOETx2IBamw>, and a live demonstration of the same at the deployed instance <http://gremlinator.iai.uni-bonn.de:8080/Demo/> for a first-hand experience.

Our work on GREMLINATOR was awarded as the Best Resource Paper¹ at the 14th IEEE International Conference on Semantic Computing (ICSC 2020) organized in San Diego, California, USA from February 2 to 5, 2020.

Finally, we tackled the third research question (RQ3), which is concerned with the performance evaluation via benchmarking both RDF and Property graph databases.

¹ <http://harshthakkar.in/wp-content/uploads/2020/04/icsc2020bestpaper.jpg>

RQ3: Benchmarking – How can we seamlessly orchestrate the benchmarking of RDF vs. Property graph Databases in an open, extensible, fair, and reproducible manner?

The main objective of this question is to objectively evaluate both the RDF and Property graph database systems for a given set of specific scenarios, benchmarks involving particular query loads over characteristic datasets have been made openly available by the graph community. Then in Chapter 6, for answering RQ3, we review the state-of-the-art on benchmarking frameworks, their suitability for evaluating both RDF and Property graph databases collectively. However, none of the existing tools allow the users to benchmark both of the above-mentioned databases in an open, extensible, and transparent manner that also supports evaluating the underlying data and query transformation approaches. We also investigated various Key Performance Indicators (KPIs) that could help understand the influence of a variety of internal and external factors on the performance of the participating database systems. After that, we identified a variety of metrics and parameters (which account for the investigated KPIs) with respect to CPU, RAM, query typology, indexing, data quality, etc. in works [45, 46, 48]. To the best of our knowledge, the proposed LITMUS [35, 47] is the first framework able to benchmark both RDF and Property graph databases in the same environment. Moreover, LITMUS provides benefits that are partially present in other benchmarking frameworks but not in their combination. In particular, LITMUS is able to: (i) promote reusability via providing a unified open, extensible architecture for orchestrating user-driven benchmarks, (ii) provide a list of comprehensive CPU and memory-based metrics and parameters for performance evaluation, (iii) offer full automation of the underlying tedious sub-tasks, and (iv) support an in-depth post-benchmark performance reporting via custom visualization using tables and plots. We conducted comprehensive experiments demonstrating the applicability and validity of LITMUS, by orchestrating automatic benchmarking top of the line three RDF and three Property graph databases.

Thus, with the contributions of RQ3, we take a substantial step by making it feasible to benchmark both RDF and Property graph database within a single open, extensible, reusable framework. As a result of this work, we have open-sourced the following resources:

- We deliver an open and independent implementation of LITMUS accessible at <https://github.com/LITMUS-Benchmark-Suite/>.
- The entire set of results and queries are made accessible at <https://drive.google.com/open?id=0B3aUtDGGrGadZWNPnzRScn1OY3M>.
- The free docker version of the LITMUS framework is available at <https://hub.docker.com/r/litmusbenchmarksuite/litmus/>.

Our work [47] on LITMUS received the Best Research & Innovation Paper Award² at the SEMANTiCS 2017 conference organised in Amsterdam, Netherlands from September 11 to 14, 2017.

In conclusion, by bridging the research gap addressed by the work presented in this dissertation,

² <https://2017.semantics.cc/awards>

we have laid a firm formal and practical foundation for addressing the interoperability gap between the RDF and Property graph databases technology stacks. These research contributions render several benefits, a few of which are summarized as follows:

- Collectively the direct mappings for both data in RDF2PG [106] and query in GREMLINATOR [41, 43] will allow applications based on Semantic Web standards, like SPARQL and RDF, to use Property graph databases in a non-intrusive fashion;
- GREMLINATOR enables the users familiar with W3C SPARQL to query a variety of TinkerPop-enabled graph databases, avoiding the need to learn a new graph query language;
- The GREMLINATOR can be used in a hybrid query layer setting between RDF triplestores and Property graph databases (e.g. as a single layer on top of AWS Neptune [80] rather than the current two separate physical layers) wherein a particular query can be dispatched to the database capable to answer the query more efficiently [4]. In particular, property graph databases have been shown to work very well for a wide range of queries, which benefit from the locality in a graph. Rather than performing expensive joins, property graph databases use micro indices to perform traversals;
- LITMUS enables a broad audience of researchers and databases vendors for benchmarking a spectrum of RDF and Property graph databases (both OLTP and OLAP systems) using frameworks such as LDBC [211] and LITMUS [47];
- Collectively all the three contributions will facilitate efforts for bridging the *data* and *query* interoperability gap between the Semantic Web and Graph database communities by serving as a stepping stone for the standardisation process³.

The work invested in this dissertation has resulted in successfully addressing the challenges identified within the scope of three major European Union-funded projects. We summarize the contribution of this thesis with respect to these projects as follows:

- **WDAQUA ITN⁴**: The *Answering Questions using Web Data* (WDAQUA), which is a Marie Skłodowska-Curie Innovative Training Network (ITN), was the primary driver of this Ph.D. thesis. It was incubated to advance the field of *data-driven* open domain Question Answering (QA) through a combination of international training, research, and innovation. The WDAQUA ITN project ran from 2015 to 2019, and with the contributions of this thesis (along with other Ph.D. students involved in it) was successfully closed.
- **SANSA⁵**: The *Scalable Semantic Analytics Stack* (SANSA) is a big data engine for scalable processing of large-scale RDF data. SANSA uses technologies such as Spark and Flink, which offer fault-tolerant, highly available, and scalable approaches to process massive sized datasets efficiently. SANSA provides the facilities for Semantic data representation, Querying, Inference, and Analytics. SANSA is one of the main academic use cases of the contributions made by this thesis, specifically the graph query layer, where the *query*

³ Part of the work in this thesis was invited for discussion at the recent W3C Graph Data Standardisation workshop (<https://www.w3.org/Data/events/data-ws-2019/>)

⁴ <http://wdaqua.eu/>

⁵ <http://sansa-stack.net/>

interoperability problem's results have been used.

- **BOOST4.0**⁶: The *Big Data for Factories* (BOOST4.0) is an on-going (2018-2021) flagship EU project put together with a collaboration of 50 global partners involving 20 academic and 30 industrial institutions. BOOST4.0 has been incubated to lead the construction of the European Industrial Data Space to improve the competitiveness of Industry 4.0. It will guide the European manufacturing industry in introducing Big Data in the factory, providing the industrial sector with the necessary tools to obtain the maximum benefit of Big Data. In production and manufacturing environments, data is often the choke-point in supporting interoperability between systems that produce data (sensors, machines, and other devices) and applications that perform analytics. This thesis's results, specifically in the *data interoperability* problem, have been successfully contributed to the BOOST4.0 project.

7.2 Limitations and Future Work

Despite successfully addressing the research objective of this dissertation, few limitations of the presented contributions remain perfected in the scope of the thesis. We list the following limitations in the chronology of the research questions they address next.

- RQ1)** Regarding the *data interoperability* contributions made in Chapter 4: (i) the simple mapping is not suitable for RDF datasets with complex vocabularies as the common names will be merged in the resulting property graph; (ii) while the generic mapping works with any RDF dataset, the size of its output property graph will be bigger than the other two mappings; lastly (iii) all the proposed three direct mappings currently do not offer support for transforming RDF data with reification and the inference rules supported by RDF Schema (e.g., sub-class, sub-property). This limitation is to be addressed by studying these features in the future.
- RQ2)** Regarding the *query interoperability* contributions made in Chapter 5: the current version of GREMLINATOR supports the translation of only SPARQL `SELECT` queries, of which it covers the SPARQL 1.0 specification (cf. Table 5.3), along with a subset of SPARQL 1.1 features (e.g. aggregation operators, explicit negation, solution modifiers, Property path). Our work can be extended to support other SPARQL query types such as `ASK`, `CONSTRUCT`, and `DESCRIBE`, thereby increasing the query fragment coverage. Furthermore, a substantial amount of work is required to support the translation of the SPARQL mentioned above query types with respect to the formal semantics and expressivity of the Gremlin traversal language. We found this to be the biggest choke-point, and at the same time, a huge opportunity in continuing ahead with research in this direction.
- RQ3)** Regarding the *database benchmarking* contributions made in Chapter 6, the LITMUS BENCHMARK SUITE does not support: (i) Federated Querying – multiple sources/endpoints being queried at the same time; (ii) Parallel Querying & Updates – multiple users/clients querying and updating a single source at the same time; (iii) the automatic generation of

⁶ <https://boost40.eu/>

synthetic datasets and queries. Instead, it allows the use of other benchmarks, via modular integration (plug-n-play style), that offer such support and leave it up to the user to choose what they deem fit according to their needs. In its current state LITMUS is more of a proof-of-concept framework and not a polished industry/commercial product. Perhaps it is an opportunity for a significant re-implementation of the same and can be launched as a novel startup.

7.3 Outlook and Closing Remarks

Based on our findings, the contributions made in this thesis, and studying the current trends, we now forecast the following general outlook for the Semantic Web, Graph database, and Artificial Intelligence research communities:

- **Querying Hybrid Knowledge Graphs (KGs):** As discussed in Section 2.3 of Chapter 2, construction and use of hybrid KGs (i.e. KGs built using both RDF and Property graph data models) has become more and more relevant with the rise in need for automated, integrated data-driven systems. Massive amounts of funding and research put together in projects such as BOOST4.0⁷, advocates the current trend, and here lies a huge opportunity. Both the RDF and Property graph databases cater to distinct advantages with respect to the flexibility of schema, querying, and storage of data, which has already been established in this thesis. Querying these hybrid KGs using a hybrid query layer will allow the user to leverage the best of both the worlds (support for reasoning and federated querying via SPARQL, and efficient execution of graph-algorithms and tasks via Gremlin). Existing commercial systems such as AWS Neptune [80] could harness this by building such a hybrid query layers instead of maintaining two separate physical data and query layers. Developing such an application will undoubtedly require a detailed study of the query language expressivity and semantics, where there is another opportunity for novel research (as mentioned earlier in this chapter).
- **Virtualization of Graph databases:** As demonstrated by Contextualized Knowledge Graph (CKG) use case by NLM (cf. Section 5.7, Chapter 5) and the large-scale knowledge graph reasoning use case by IBM Research [202] GREMLINATOR can be successfully employed in a variety of applications as a virtualized query layer on top of Property graph databases systems. Due to the advantage catered by Apache TinkerPop framework [18], GREMLINATOR can be used to query both OLTP and OLAP systems using SPARQL. There is certainly a lot of scope in this direction for the adoption of our contributions and extending its coverage to support other query types in SPARQL (as discussed in the section above).
- **Question Answering (QA) over Property graphs:** QA has almost always been carried out over RDF powered KGs. There exists a substantial amount of work [212–215] in this regard covering the entire QA pipeline [216] – entity linking, entity disambiguation, natural language processing, and other machine learning approaches. However, there is not much work done in the direction of supporting QA over Property graphs. One primary

⁷ <https://boost40.eu/>

reason for this is the lack of tools that allow translation of natural language questions into a formal Property graph query language. This can now be addressed with our contribution GREMLINATOR, as it enables executing SPARQL queries over Property graph databases directly. Thus, all the existing components and QA pipelines can be used in conjunction with GREMLINATOR and RDF2PG for enabling full-scale QA over Property graph and hybrid-based KGs.

- **Identifying suitable Databases for domain specific-use cases:** The contributions made in this thesis can be used for a better understanding of performance issues covering a wide variety of RDF and Property graph database by automatically benchmarking them with respect to specific use cases (addressing big data, complex queries, and hybrid ontologies), which have an explicit requirement of federated/distributed and/or graph analytical querying, in fields such as Life Sciences, Pharmaceutical, Automotive, Finance. Furthermore, machine learning-based approaches can be used for automatically classifying and delegating specific queries over RDF vs. Graph databases based on pre-defined criteria (e.g., query typology) at run-time. For instance, recommendation and analytical queries (retrieving similar movies based on given criteria) that consists of a high number of joins and graph-based tasks (shortest path, page rank, etc.) can be automatically identified and executed over Graph databases. At the same time, simple queries with a fewer number of joins/triples (typically used in question answering systems – *who is the father of Jon Snow?*) can be executed over RDF databases/triplestores.

In conclusion, from what we have experienced so far, the field of Property graph is yet very young, and with the rise of data and query language standards, it is certainly picking up momentum. Additionally, the work formulated in this dissertation lies at the very core of addressing the elephant-sized interoperability issue in the international room global database community. This is evident that despite a very recent introduction, works such as GREMLINATOR and RDF2PG have started getting attention and adoption from academia and industry, respectively. The most important by-product of this work, is the Apache TinkerPop `sparql-gremlin` plugin's reuse and adoption, which has resulted into the authors involvement in international collaborative standardization efforts such as Graph Query Language (GQL)⁸, Linked Data Benchmark Council (LDBC)⁹, Property graph Schema Working Group (PGSWG)¹⁰, Existing Languages Working Group (ELWG)¹¹. Apart from the above-mentioned activities, there are several other industry collaborations in progress, which, unfortunately, cannot be mentioned in this thesis due to their commercial restrictions. We expect the community to build on top of the laid formal and practical foundation for supporting interoperability between the RDF and Property graph databases presented in this dissertation.

⁸ <https://www.gqlstandards.org/>

⁹ <http://ldbkcouncil.org/>

¹⁰ <https://docs.google.com/document/d/1YBD0o6VsFbkhyFIi5jfYhtx16TmKVDWFoLCPw8ZR3k/edit>

¹¹ <https://www.gqlstandards.org/existing-languages>

Bibliography

- [1] T. J. Berners-Lee and R. Cailliau, *WorldWideWeb: Proposal for a HyperText project*, (1990) (cit. on p. 1).
- [2] H. Garcia-Molina, J. D. Ullman and J. Widom, *Database systems: the complete book*, Pearson Education India, 2009 (cit. on pp. 1, 104).
- [3] L. Gomes-Jr, B. Amann and A. Santanché, “Beta-algebra: Towards a relational algebra for graph analysis”, *EDBT/ICDT 2015 Joint Conference*, 2015 157 (cit. on pp. 2, 33, 104).
- [4] S. Das, J. Srinivasan, M. Perry, E. I. Chong and J. Banerjee, “A Tale of Two Graphs: Property Graphs as RDF in Oracle.”, *Proc. of the International Conference on Extending Database Technology (EDBT)*, 2014 762 (cit. on pp. 2, 46, 159).
- [5] A. Polleres, *3.19 Combining Graph Queries with Graph Analytics*, Knowledge Graphs: New Directions for Knowledge Representation on the Semantic Web () (cit. on p. 2).
- [6] G. Klyne and J. Carroll, *Resource Description Framework (RDF) Concepts and Abstract Syntax*, <http://www.w3.org/TR/2004/REC-115-concepts-20040210/>, 2004 (cit. on pp. 2, 61, 62).
- [7] S. Harris and A. Seaborne, *SPARQL 1.1 Query Language - W3C Recommendation*, <https://www.w3.org/TR/sparql11-query/>, 2013 (cit. on pp. 2, 25, 50, 61).
- [8] D. Brickley and R. V. Guha, *RDF Schema 1.1, W3C Recommendation*, <https://www.w3.org/TR/rdf-schema/>, 2014 (cit. on pp. 2, 23, 61, 67, 68).
- [9] W. O. W. Group, *OWL 2 Web Ontology Language, Document Overview - W3C Recommendation*, <https://www.w3.org/TR/owl2-overview/>, 2012 (cit. on p. 2).
- [10] H. Knublauch and D. Kontokostas, *Shapes Constraint Language (SHACL) - W3C Recommendation*, <https://www.w3.org/TR/shacl/>, 2017 (cit. on p. 2).
- [11] I. Boneva, J. E. L. Gayo, S. Hym, E. G. Prud’hommeau, H. R. Solbrig and S. Staworko, *Validating RDF with shape expressions*, CoRR, abs/1404.1270 (2014) (cit. on p. 2).
- [12] R. Angles, “The Property Graph Database Model”, *Proc. Alberto Mendelzon International Workshop on Foundations of Data Management (AMW)*, 2018 (cit. on p. 2).
- [13] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. Reutter and D. Vrgoč, *Foundations of Modern Query Languages for Graph Databases*, ACM Computing Surveys (CSUR) **50** (2017) (cit. on pp. 2, 103, 105, 106, 112).

- [14] J. Pokorný, M. Valenta and J. Kovačič, *Integrity constraints in graph databases*, *Procedia Computer Science* **109** (2017) 975 (cit. on pp. [2](#), [61](#)).
- [15] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer and A. Taylor, “Cypher: An evolving query language for property graphs”, *Proceedings of the 2018 International Conference on Management of Data*, ACM, 2018 1433 (cit. on pp. [2](#), [31](#)).
- [16] M. A. Rodriguez, “The gremlin graph traversal machine and language (invited talk)”, *Proceedings of the 15th Symposium on Database Programming Languages*, ACM, 2015 1 (cit. on pp. [2](#), [31](#)).
- [17] A. TinkerPop, *The Gremlin Graph Traversal Machine and Language*, <https://tinkerpop.apache.org/gremlin.html> (cit. on pp. [2](#), [31](#), [33](#)).
- [18] A. T. P. Home, *Apache Tinkerpop Home*, Web Page (2019), URL: <http://tinkerpop.apache.org/> (cit. on pp. [2](#), [36](#), [123](#), [161](#), [185](#)).
- [19] O. van Rest, S. Hong, J. Kim, X. Meng and H. Chafi, “PGQL: a Property Graph Query Language”, *Proc. of the Int. Workshop on Graph Data Management Experiences and Systems (GRADES)*, 2013 (cit. on p. [2](#)).
- [20] *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*, IEEE Std 610, 1991 (cit. on p. [3](#)).
- [21] S.Ceri, L. Tanca and R. Zicari, “Supporting interoperability between new database languages”, *Proc. of the 5th Annual European Computer Conference (CompEuro)*, 1991 (cit. on p. [3](#)).
- [22] C. Parent and S. Spaccapietra, *Database Integration: the Key to Data Interoperability*, *Advances in Object-Oriented Data Modeling* (2000) (cit. on p. [3](#)).
- [23] S. Heiler, *Semantic Interoperability*, *ACM Comput. Surv.* **27** (1995) 271 (cit. on pp. [3](#), [4](#)).
- [24] A. P. Sheth, “Changing Focus on Interoperability in Information Systems: From System, Syntax, Structure to Semantics”, *Interoperating Geographic Information Systems*, Springer US, 1999 5 (cit. on p. [3](#)).
- [25] J. Park and S. Ram, *Information Systems Interoperability: What Lies Beneath?*, *ACM Transactions on Information Systems* **22** (2004) 595, ISSN: 1046-8188 (cit. on p. [3](#)).
- [26] R. Angles, P. Boncz, J. Larriba-Pey, I. Fundulaki, T. Neumann, O. Erling, P. Neubauer, N. Martinez-Bazan, V. Kotsev and I. Toma, *The Linked Data Benchmark Council: a Graph and RDF industry benchmarking effort*, *Sigmod Record* **43** (2014) (cit. on p. [3](#)).
- [27] C. Parent and S. Spaccapietra, *Issues and Approaches of Database Integration*, *Commun. ACM* **41** (1998) 166 (cit. on p. [3](#)).
- [28] A. M. Ouksel and A. Sheth, *Semantic Interoperability in Global Information Systems*, *SIGMOD Rec.* **28** (1999) 5 (cit. on p. [3](#)).
- [29] L. Predoiu and A. V. Zhdanova, “Semantic Web Languages and Ontologies”, *Encyclopedia of Internet Technologies and Applications*, IGI Global, 2008 7 (cit. on p. [3](#)).

-
- [30] D. N. Joundrey and A. G. Taylor, *The Organization of Information*, Fourth, Library and Information Science, Libraries Unlimited, 2017 (cit. on p. 4).
- [31] J. Zhan, W. S. Luk and C. Wong, “An Object-Oriented Approach to Query Interoperability”, *Database Reengineering and Interoperability*, Springer US, 1996 141 (cit. on p. 4).
- [32] *The GQL Manifesto*, <https://gql.today/> (cit. on p. 6).
- [33] G. Szárnyas, J. Marton, J. Maginecz and D. Varró, *Reducing Property Graph Queries to Relational Algebra for Incremental View Maintenance*, arXiv preprint arXiv:1806.07344 (2018) (cit. on p. 6).
- [34] H. Thakkar, D. Punjani, S. Auer and M.-E. Vidal, “Towards an integrated graph algebra for graph pattern matching with Gremlin”, *International Conference on Database and Expert Systems Applications*, Springer, 2017 81 (cit. on pp. 6, 12, 102, 112, 185).
- [35] H. Thakkar, “Towards an Open Extensible Framework for Empirical Benchmarking of Data Management Solutions: LITMUS”, *The Semantic Web - 14th International Conference, ESWC 2017, Portorož, Slovenia, May 28 - June 1, 2017, Proceedings, Part II*, 2017 (cit. on pp. 7, 13, 136, 158, 185).
- [36] J. F. Sequeda, *On Direct Mapping for Integrating SQL Databases with the Semantic Web*, tech. rep., Technical Report# HR-08-09. The University of Texas at Austin. Department of . . . , 2008 (cit. on p. 7).
- [37] R. Angles, H. Thakkar and D. Tomaszuk, *RDF2PG experimental datasets*, 2020, URL: <https://doi.org/10.6084/m9.figshare.12021156.v5> (cit. on pp. 11, 93, 96, 156, 185).
- [38] D. Tomaszuk, R. Angles and H. Thakkar, *PGO: Describing Property Graphs in RDF*, IEEE Access (2020) 118355 (cit. on pp. 11, 62, 156, 185).
- [39] R. Angles, H. Thakkar and D. Tomaszuk, “RDF and property graphs interoperability: Status and issues”, *Proceedings of the 13th Alberto Mendelzon International Workshop on Foundations of Data Management, Asunción, Paraguay*, 2019 (cit. on pp. 11, 43, 62, 156, 185).
- [40] H. Thakkar, S. Auer and M.-E. Vidal, *Formalizing Gremlin Pattern Matching Traversals in an Integrated Graph Algebra*, (2019) (cit. on pp. 12, 43, 102, 157, 185).
- [41] H. Thakkar, D. Punjani, Y. Keswani, J. Lehmann and S. Auer, *A Stitch in Time Saves Nine—SPARQL querying of Property Graphs using Gremlin Traversals*, arXiv:1801.02911 (2018) (cit. on pp. 12, 39, 43, 98, 102, 103, 157, 159, 185).
- [42] H. Thakkar, D. Punjani, J. Lehmann and S. Auer, “Two for One: Querying Property Graph Databases using SPARQL via Gremlinator”, *Proceedings of the 1st ACM SIGMOD Joint International Workshop GRADES-NDA*, 2018 (cit. on pp. 12, 29, 39, 98, 102, 103, 157, 185).

- [43] H. Thakkar, R. Angles, M. Rodriguez, S. Mallette and J. Lehmann, “Let’s build Bridges, not Walls: SPARQL Querying of TinkerPop Graph Databases with Sparql-Gremlin”, *IEEE International Conference on Semantic Computing (ICSC)*, 2020 408 (cit. on pp. [12](#), [43](#), [98](#), [157](#), [159](#), [185](#)).
- [44] H. Thakkar, D. Punjani, J. Lehmann and S. Auer, “Two for One: Querying Property Graph Databases Using SPARQL via Gremlinator”, *Proc. of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, ACM, 2018 1 (cit. on p. [12](#)).
- [45] H. Thakkar, K. M. Endris, J. M. Gimenez-Garcia, J. Debattista, C. Lange and S. Auer, “Are linked datasets fit for open-domain question answering? A quality assessment”, *Proceedings of the 6th International Conference on Web Intelligence, Mining and Semantics*, ACM, 2016 19 (cit. on pp. [13](#), [43](#), [136](#), [146](#), [147](#), [158](#), [185](#)).
- [46] K. M. Endris, J. M. Giménez-Garcia, H. Thakkar, E. Demidova, A. Zimmermann, C. Lange and E. Simperl, “Dataset reuse: An analysis of references in community discussions, publications and data”, *Proceedings of the Knowledge Capture Conference*, ACM, 2017 5 (cit. on pp. [13](#), [146](#), [147](#), [158](#), [185](#)).
- [47] H. Thakkar, Y. Keswani, M. Dubey, J. Lehmann and S. Auer, “Trying Not to Die Benchmarking: Orchestrating RDF and Graph Data Management Solution Benchmarks Using LITMUS”, *Proceedings of the 13th International Conference on Semantic Systems, SEMANTICS 2017, Amsterdam, The Netherlands, September 11-14, 2017*, 2017 (cit. on pp. [13](#), [14](#), [136](#), [158](#), [159](#), [185](#)).
- [48] Y. Keswani, H. Thakkar, M. Dubey, J. Lehmann and S. Auer, “The LITMUS Test: Benchmarking RDF and Graph Data Management Systems”, *CEUR-WS Proceedings, SEMANTiCS 2017, Amsterdam, Nederland, 2017* (cit. on pp. [13](#), [103](#), [158](#), [185](#)).
- [49] T. Berners-Lee, J. Hendler and O. Lassila, *The Semantic Web*, Scientific American (2001) (cit. on pp. [21](#), [23](#)).
- [50] T. Berners-Lee, *Linked data, 2006*, 2006 (cit. on p. [22](#)).
- [51] C. Bizer, T. Heath, K. Idehen and T. Berners-Lee, “Linked data on the web (LDOW2008)”, *Proceedings of the 17th international conference on World Wide Web*, ACM, 2008 1265 (cit. on p. [22](#)).
- [52] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak and Z. Ives, “Dbpedia: A nucleus for a web of open data”, *The semantic web*, Springer, 2007 722 (cit. on pp. [22](#), [23](#), [25](#), [39](#), [40](#), [141](#)).
- [53] D. Vrandečić, “Wikidata: a new platform for collaborative data collection”, *Proceedings of the 21st World Wide Web Conference, WWW 2012, Lyon, France, April 16-20, 2012 (Companion Volume)*, 2012 1063 (cit. on pp. [22](#), [39](#)).
- [54] F. M. Suchanek, G. Kasneci and G. Weikum, “Yago: a core of semantic knowledge”, *Proceedings of the 16th international conference on World Wide Web*, ACM, 2007 697 (cit. on pp. [22](#), [39](#)).

-
- [55] G. Klyne, J. J. Carroll and B. McBride, *RDF 1.1 concepts and abstract syntax (W3C Recommendation)*, <https://www.w3.org/TR/rdf11-concepts/>, 2014 (cit. on p. 23).
- [56] D. Beckett, T. Berners-Lee, E. Prud'hommeaux and G. Carothers, *RDF 1.1 Turtle*, World Wide Web Consortium (2014) (cit. on pp. 23–25, 63, 67).
- [57] G. Antoniou and F. van Harmelen, “Web Ontology Language: OWL”, *Handbook on Ontologies*, 2004 67 (cit. on p. 24).
- [58] R. Studer, V. R. Benjamins and D. Fensel, *Knowledge engineering: principles and methods*, *Data & knowledge engineering* **25** (1998) 161 (cit. on p. 24).
- [59] E. Prud'hommeaux and A. Seaborne, *SPARQL Query Language for RDF - W3C Recommendation*, <https://www.w3.org/TR/rdf-sparql-query/>, 2008 (cit. on pp. 25, 50).
- [60] J. Pérez, M. Arenas and C. Gutierrez, *Semantics and Complexity of SPARQL*, *ACM Trans. Database Syst.* **34** (2009) 16:1, ISSN: 0362-5915, URL: <http://doi.acm.org/10.1145/1567274.1567278> (cit. on pp. 26, 27, 35, 103, 105, 108, 112, 122).
- [61] E. Prud'hommeaux and A. Seaborne, *SPARQL Query Language for RDF (W3C Recommendation)*, <https://www.w3.org/TR/rdf-sparql-query/>, 2008 (cit. on p. 27).
- [62] S. Harris, A. Seaborne and E. Prud'hommeaux, *SPARQL 1.1 Query Language (W3C Recommendation)*, <https://www.w3.org/TR/sparql11-query/>, 2013 (cit. on pp. 27, 28).
- [63] R. Angles et al., “The Multiset Semantics of SPARQL Patterns”, *The Semantic Web - ISWC -15th International Semantic Web Conference, Kobe, Japan*, 2016 20, URL: http://dx.doi.org/10.1007/978-3-319-46523-4_2 (cit. on pp. 27, 35, 103).
- [64] R. Angles, M. Arenas et al., *Foundations of Modern Graph Query Languages*, *CoRR* **abs/1610.06264** (2016) (cit. on p. 28).
- [65] A. Chebotko, S. Lu, H. M. Jamil and F. Fotouhi, *Semantics preserving SPARQL-to-SQL query translation for optional graph patterns*, tech. rep., 2006 (cit. on pp. 28, 47, 48, 50, 51, 56).
- [66] M. Rodriguez-Muro and M. Rezk, *Efficient SPARQL-to-SQL with R2RML mappings.*, *J. Web Sem.* **33** (2015) 141, URL: <http://dblp.uni-trier.de/db/journals/ws/ws33.html#Rodriguez-%20MuroR15> (cit. on pp. 28, 56).
- [67] J. F. Sequeda and D. P. Miranker, *Ultrawrap: SPARQL execution on relational data*, *Web Semantics: Science, Services and Agents on the World Wide Web* **22** (2013) 19 (cit. on pp. 28, 48, 51, 56).
- [68] G. E. Modoni, M. Sacco and W. Terkaj, “A survey of RDF store solutions”, *2014 International Conference on Engineering, Technology and Innovation (ICE)*, IEEE, 2014 1 (cit. on p. 28).

- [69] I. Robinson, J. Webber and E. Eifrem, *Graph Databases*, First, O'Reilly Media, 2013 (cit. on pp. 29, 61).
- [70] M. A. Rodriguez and P. Neubauer, "The Graph Traversal Pattern", *Graph Data Management: Techniques and Applications*. IGI Global, 2011 29, URL: <http://dx.doi.org/10.4018/978-1-61350-053-8.ch002> (cit. on pp. 29, 33, 103, 104, 106–108, 112).
- [71] N. -. T. G. Platform, 2017, URL: <https://neo4j.com/> (cit. on pp. 29, 38).
- [72] R. Angles, H. Thakkar and D. Tomaszuk, "RDF and Property Graphs Interoperability: Status and Issues", *Proceedings of the 13th Alberto Mendelzon International Workshop on Foundations of Data Management, Asunción, Paraguay, June 3-7, 2019*. 2019, URL: <http://ceur-ws.org/Vol-2369/paper01.pdf> (cit. on pp. 30, 103).
- [73] U. Brandes, M. Eiglsperger and J. Lerner, *GraphML primer*, Online: <http://graphml.graphdrawing.org/primer/graphml-primer.html> [29.05. 2007] (2004) (cit. on pp. 31, 71, 73).
- [74] M. Himsolt, *GML: A portable graph file format*, Html page under <http://www.fmi.uni-passau.de/graphlet/gml/gml-tr.html>, Universität Passau (1997) (cit. on p. 31).
- [75] C. Unger, A. N. Ngomo et al., "6th Open Challenge on Question Answering over Linked Data (QALD-6)", *Semantic Web Challenges - Third SemWebEval Challenge at ESWC 2016, Heraklion, Crete, Greece, 2016* 171, URL: http://dx.doi.org/10.1007/978-3-319-46565-4_13 (cit. on p. 33).
- [76] M. A. Rodriguez and P. Neubauer, "A path algebra for multi-relational graphs", *Workshops Proceedings of the 27th International Conference on Data Engineering, ICDE 2011*, 2011 128, URL: <http://dx.doi.org/10.1109/ICDEW.2011.5767613> (cit. on pp. 33, 103, 104, 106, 107, 112).
- [77] M. A. Rodriguez, "The Gremlin graph traversal machine and language (invited talk)", *Proceedings of the 15th Symposium on Database Programming Languages, Pittsburgh, PA, USA, October 25-30, 2015*, 2015 1, URL: <http://doi.acm.org/10.1145/2815072.2815073> (cit. on pp. 35, 107, 109–112).
- [78] M. Schmidt, M. Meier and G. Lausen, "Foundations of SPARQL query optimization", *Proceedings of the 13th International Conference on Database Theory, ACM*, 2010 4 (cit. on pp. 35, 112, 122).
- [79] Alibaba, *Alibaba Graph Database Home*, Web Page (2019), URL: <https://www.alibabacloud.com/help/product/102714.htm> (cit. on p. 37).
- [80] Amazon, *AWS Neptune Home*, Web Page (2019), URL: <https://aws.amazon.com/neptune/> (cit. on pp. 37, 159, 161).
- [81] Arrangodb, *ArangoDB Home*, Web Page (2019), URL: <https://www.arangodb.com/> (cit. on p. 37).
- [82] Blazegraph, *Blazegraph Home*, Web Page (2019), URL: <https://blazegraph.com/> (cit. on p. 37).

-
- [83] Microsoft, *Microsoft Azure CosmosDB Home*, Web Page (2019), URL: <https://docs.microsoft.com/en-us/azure/cosmos-db/introduction> (cit. on p. 37).
- [84] Influxdata, *Chronograf Home*, Web Page (2019), URL: <https://www.influxdata.com/time-series-platform/chronograf/> (cit. on p. 37).
- [85] Datastax, *Datastax Enterprise Graph Home*, Web Page (2019), URL: <https://www.datastax.com/products/datastax-graph> (cit. on p. 37).
- [86] Grakn.ai, *GRAKN.AI Home*, Web Page (2019), URL: <https://grakn.ai/> (cit. on p. 37).
- [87] Apache, *Apache Spark Home*, Web Page (2019), URL: <https://spark.apache.org/> (cit. on p. 38).
- [88] Huawei, *Huawei Graph Engine Service Home*, Web Page (2019), URL: <https://www.huaweicloud.com/en-us/product/ges.html> (cit. on p. 38).
- [89] IBM, *IBM Graph Home*, Web Page (2019), URL: <https://www.ibm.com/de-de/marketplace/graph> (cit. on p. 38).
- [90] JanusGraph, *JanusGraph Home*, Web Page (2019), URL: <https://janusgraph.org/> (cit. on p. 38).
- [91] Oritendb, *OrientDB Home*, Web Page (2019), URL: <https://orientdb.com/> (cit. on p. 38).
- [92] Apache, *Apache S2Graph Home*, Web Page (2019), URL: <https://s2graph.apache.org/> (cit. on p. 38).
- [93] sqlg, *sqlg Home*, Web Page (2019), URL: <http://www.sqlg.org/docs/2.0.0-SNAPSHOT/> (cit. on p. 38).
- [94] stardog, *Stardog Home*, Web Page (2019), URL: <https://www.stardog.com/> (cit. on p. 38).
- [95] Titan, *Titan Home*, Web Page (2019), URL: <https://titan.thinkaurelius.com/> (cit. on p. 38).
- [96] A. Singhal, *Introducing the knowledge graph: things, not strings*, Official google blog (2012) (cit. on p. 39).
- [97] K. Bollacker, C. Evans, P. Paritosh, T. Sturge and J. Taylor, “Freebase: a collaboratively created graph database for structuring human knowledge”, *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, AcM, 2008 1247 (cit. on p. 39).
- [98] H. Paulheim, *Knowledge graph refinement: A survey of approaches and evaluation methods*, *Semantic Web* **8** (2017) 489 (cit. on pp. 39, 40).
- [99] N. Noy, Y. Gao, A. Jain, A. Narayanan, A. Patterson and J. Taylor, *Industry-scale knowledge graphs: Lessons and challenges*, *Queue* **17** (2019) 48 (cit. on pp. 40, 41).

- [100] F. Quécole, R. Martines, J. M. Giménez-García and H. Thakkar, “Towards capturing contextual semantic information about statements in web tables”, *Joint Proceedings of the International Workshops on Contextualized Knowledge Graphs, and Semantic Statistics (CKGSemStats) co-located with 17th International Semantic Web Conference (ISWC 2018), Monterey, USA, 2018* (cit. on p. 41).
- [101] A. L. Gentile, D. Gruhl, P. Ristoski and S. Welch, “Personalized knowledge graphs for the pharmaceutical domain”, *International Semantic Web Conference*, Springer, 2019 400 (cit. on p. 42).
- [102] M. Färber, “The Microsoft Academic Knowledge Graph: A Linked Data Source with 8 Billion Triples of Scholarly Data”, *International Semantic Web Conference*, Springer, 2019 113 (cit. on p. 42).
- [103] M. Y. Jaradeh, S. Auer, M. Prinz, V. Kovtun, G. Kismihók and M. Stocker, *Open Research Knowledge Graph: Towards Machine Actionability in Scholarly Communication*, arXiv preprint arXiv:1901.10816 (2019) (cit. on pp. 42, 132).
- [104] H. Thakkar, R. Angles, D. Tomaszuk and J. Lehmann, *Direct Mappings between RDF and Property Graph Databases*, arXiv preprint arXiv:1912.02127 (2019) (cit. on pp. 43, 156, 185).
- [105] M. N. Mami, D. Graux, H. Thakkar, S. Scerri, S. Auer and J. Lehmann, *The query translation landscape: a survey*, arXiv preprint arXiv:1910.03118 (2019) (cit. on pp. 43, 157, 185).
- [106] R. Angles, H. Thakkar and D. Tomaszuk, *Mapping RDF Databases to Property Graph Databases*, *IEEE Access* 8 (2020) 86091 (cit. on pp. 43, 61, 62, 156, 159, 185).
- [107] O. Hartig, *Reconciliation of RDF* and property graphs*, arXiv:1409.3288 (2014) (cit. on pp. 45, 46, 54).
- [108] A. Schätzle, M. Przyjacieli-Zablocki, T. Berberich and G. Lausen, “S2X: Graph-Parallel Querying of RDF with GraphX”, *Biomedical Data Management and Graph Online Querying*, Springer International Publishing, 2016 155 (cit. on pp. 45, 46).
- [109] V. Nguyen, J. Leeka, O. Bodenreider and A. Sheth, *A formal graph model for RDF and its implementation*, arXiv:1606.00480 (2016) (cit. on pp. 45, 46).
- [110] D. Tomaszuk, “RDF data in property graph model”, *MTSR*, Springer, 2016 104 (cit. on pp. 45, 46).
- [111] S. Matsumoto, R. Yamanaka and H. Chiba, “Mapping RDF Graphs to Property Graphs”, *Proc. of the Fifth International Workshop on Practical Application of Ontology for Semantic Data Engineering*, 2018 (cit. on p. 46).
- [112] O. Hartig and B. Thompson, *Foundations of an alternative approach to reification in RDF*, arXiv preprint arXiv:1406.3399 (2019) (cit. on pp. 46, 54).

-
- [113] S. S. Sahoo, W. Halb, S. Hellmann, K. Idehen, T. Thibodeau Jr, S. Auer, J. Sequeda and A. Ezzat,
A survey of current approaches for mapping of relational databases to RDF,
W3C RDB2RDF Incubator Group Report **1** (2009) 113 (cit. on p. 47).
- [114] F. Michel, J. Montagnat and C. F. Zucker,
A survey of RDB to RDF translation approaches and tools, (2014) (cit. on p. 47).
- [115] M. Hert, G. Reif and H. C. Gall, “A comparison of RDB-to-RDF mapping languages”,
Proceedings of the 7th International Conference on Semantic Systems, ACM, 2011 25
(cit. on p. 47).
- [116] J. Rachapalli, V. Khadilkar, M. Kantarcioglu and B. Thuraisingham,
RETRO: A Framework for Semantics Preserving SQL-to-SPARQL Translation,
The University of Texas at Dallas **800** (2011) 75080 (cit. on pp. 47, 52, 56).
- [117] S. Ramanujam, A. Gupta, L. Khan, S. Seida and B. Thuraisingham,
“R2D: A bridge between the semantic web and relational visualization tools”,
2009 IEEE International Conference on Semantic Computing, IEEE, 2009 303
(cit. on pp. 47, 56).
- [118] S. Ramanujam, A. Gupta, L. Khan, S. Seida and B. Thuraisingham,
“R2D: Extracting relational structure from RDF stores”,
*Proceedings of the 2009 IEEE/WIC/ACM International Joint Conference on Web
Intelligence and Intelligent Agent Technology-Volume 01*, IEEE Computer Society, 2009
361 (cit. on pp. 47, 56).
- [119] M. Hert, G. Reif and H. C. Gall, “Updating Relational Data via SPARQL/Update”,
Proceedings of the 2010 EDBT/ICDT Workshops, EDBT '10, ACM, 2010 24:1,
ISBN: 978-1-60558-990-9 (cit. on p. 47).
- [120] A. Chebotko, S. Lu and F. Fotouhi, *Semantics preserving SPARQL-to-SQL translation*,
Data & Knowledge Engineering **68** (2009) 973 (cit. on pp. 47, 48, 50, 51, 56).
- [121] S. Auer, S. Dietzold, J. Lehmann, S. Hellmann and D. Aumüller,
“Triplify: Light-weight Linked Data Publication from Relational Databases”,
Proc. of the 18th International Conference on World Wide Web, ACM, 2009 621
(cit. on p. 47).
- [122] C. Bizer and A. Seaborne,
“D2RQ - Treating Non-RDF Databases as Virtual RDF Graphs”, *ISWC2004 (posters)*,
2004 (cit. on p. 48).
- [123] S. Polfriet and R. Ichise,
“Automated Mapping Generation for Converting Databases into Linked Data”,
ISWC Posters&Demos, vol. 658, CEUR Workshop Proceedings, CEUR-WS.org, 2010
(cit. on p. 48).
- [124] S. Das, R. Cyganiak and S. Sundara, *R2RML: RDB to RDF Mapping Language*,
W3C Recommendation, World Wide Web Consortium, 2012 (cit. on p. 48).
- [125] B. Elliott, E. Cheng, C. Thomas-Ogbuji and Z. M. Ozsoyoglu,
“A complete translation from SPARQL into efficient SQL”, *Proceedings of the IDEAS*,
ACM, 2009 31 (cit. on pp. 48, 50, 51, 56).

- [126] M. Rodriguez-Muro and M. Rezk, *Efficient SPARQL-to-SQL with R2RML mappings*, *Web Semantics* **33** (2015) 141 (cit. on pp. 48, 50, 51).
- [127] A. Seaborne, D. Steer and S. Williams, *SQL-RDF*, <http://www.w3.org/2007/03/RdfRDB/papers/seaborne.html>, 2007 (cit. on p. 48).
- [128] F. Priyatna, O. Corcho and J. Sequeda, “Formalisation and experiences of R2RML-based SPARQL to SQL query translation using Morph”, *Proceedings of the 23rd international conference on World wide web*, ACM, 2014 479 (cit. on pp. 48, 51, 56).
- [129] D. Calvanese, B. Cogrel, S. Komla-Ebri, R. Kontchakov, D. Lanti, M. Rezk, M. Rodriguez-Muro and G. Xiao, *Ontop: Answering SPARQL queries over relational databases*, *Semantic Web* **8** (2017) 471 (cit. on pp. 49–51).
- [130] A. Dimou, M. V. Sande, P. Colpaert, R. Verborgh, E. Mannens and R. V. de Walle, “RML: A Generic Language for Integrated RDF Mappings of Heterogeneous Data”, *Proc. of the Linked Data on the Web Workshop*, 2014 (cit. on p. 49).
- [131] M. Lefrançois, A. Zimmermann and N. Bakerally, “A SPARQL extension for generating RDF from heterogeneous formats”, *European Semantic Web Conference*, Springer, 2017 35 (cit. on p. 50).
- [132] S. Bischof, S. Decker, T. Krennwallner, N. Lopes and A. Polleres, *Mapping between RDF and XML with XSPARQL*, *Journal of Data Semantics* **1** (2012) (cit. on p. 50).
- [133] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, J. Siméon and M. Stefanescu, *XQuery 1.0: An XML query language*, (2002) (cit. on p. 50).
- [134] D. Chamberlin, D. Florescu, J. Robie, J. Simeon and M. Stefanescu, “XQuery: A query language for XML”, *SIGMOD Conference*, vol. 682, 2003 (cit. on p. 50).
- [135] S. Battle, “Gloze: XML to RDF and back again”, *Jena User Conference*, 2006 (cit. on p. 50).
- [136] D. Connolly, *Gleaning Resource Descriptions from Dialects of Languages (GRDDL)*, W3C Recommendation, World Wide Web Consortium, 2007 (cit. on p. 50).
- [137] O. Corby and C. Faron-Zucker, “STTL: A SPARQL-based Transformation Language for RDF”, *Proc. of the 11th International Conference on Web Information Systems and Technologies (WEBIST)*, 2015 (cit. on p. 50).
- [138] F. Zemke, *Converting sparql to sql*, tech. rep., Technical Report, October 2006., 2006 (cit. on p. 51).
- [139] L. Carata, *Cyp2SQL: Cypher to SQL Translation*, Accessed: 22-09-2018, 2017, URL: <https://github.com/DTG-FRESCO/cyp2sql> (cit. on p. 52).
- [140] SQL2Gremlin, *SQL2Gremlin*, Accessed: 31-07-2019, 2019, URL: <http://sql2gremlin.com> (cit. on pp. 52, 56).

-
- [141] W. Sun, A. Fokoue, K. Srinivas, A. Kementsietsidis, G. Hu and G. Xie, “SQLGraph: an efficient relational-based property graph store”, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ACM, 2015 1887 (cit. on p. 52).
- [142] M. Hausenblas, R. Grossman, A. Harth and P. Cudré-Mauroux, *Large-scale Linked Data Processing-Cloud Computing to the Rescue?.*, CLOSER **20** (2012) 246 (cit. on p. 53).
- [143] R. Mutharaju, S. Sakr, A. Sala and P. Hitzler, *D-SPARQ: distributed, scalable and efficient RDF query engine*, (2013) (cit. on pp. 53, 56).
- [144] F. Michel, C. F. Zucker and J. Montagnat, “A generic mapping-based query translation from SPARQL to various target database query languages”, *12th International Conference on Web Information Systems and Technologies (WEBIST'16)*, 2016 (cit. on p. 53).
- [145] S. Groppe, J. Groppe, V. Linnemann, D. Kukulenz, N. Hoeller and C. Reinke, “Embedding sparql into XQuery/XSLT”, *Proceedings of the 2008 ACM symposium on Applied computing*, ACM, 2008 2271 (cit. on pp. 53, 56).
- [146] N. Bikakis, C. Tsinaraki, I. Stavrakantonakis, N. Gioldasis and S. Christodoulakis, *The SPARQL2XQuery interoperability framework*, *World Wide Web* **18** (2015) 403 (cit. on pp. 53, 56).
- [147] N. Bikakis, N. Gioldasis, C. Tsinaraki and S. Christodoulakis, “Querying xml data with sparql”, *International Conference on Database and Expert Systems Applications*, Springer, 2009 372 (cit. on pp. 53, 56).
- [148] N. Bikakis, N. Gioldasis, C. Tsinaraki and S. Christodoulakis, “Semantic based access over xml data”, *World Summit on Knowledge Society*, Springer, 2009 259 (cit. on pp. 53, 56).
- [149] P. M. Fischer, D. Florescu, M. Kaufmann and D. Kossmann, *Translating SPARQL and SQL to XQuery*, XML Prague (2011) 81 (cit. on pp. 54, 56).
- [150] M. Droop, M. Flarer, J. Groppe, S. Groppe, V. Linnemann, J. Pinggera, F. Santner, M. Schier, F. Schöpf, H. Staffler et al., “Translating xpath queries into sparql queries”, *OTM Confederated International Conferences "On the Move to Meaningful Internet Systems"*, Springer, 2007 9 (cit. on p. 54).
- [151] O. Hartig, *Reconciliation of RDF* and property graphs*, arXiv preprint arXiv:1409.3288 (2014) (cit. on pp. 54, 140).
- [152] S. Kiminki, J. Knuuttila and V. Hirvisalo, “SPARQL to SQL translation based on an intermediate query language”, *The 6th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2010)*, 2010 32 (cit. on p. 56).
- [153] J. Unbehauen, C. Stadler and S. Auer, “Accessing relational data on the web with sparqlmap”, *Joint International Semantic Technology Conference*, Springer, 2012 65 (cit. on p. 56).

- [154] J. Lu, F. Cao, L. Ma, Y. Yu and Y. Pan, “An effective sparql support over relational databases”, *Semantic web, ontologies and databases*, Springer, 2008 57 (cit. on p. 56).
- [155] T. H. D. Araujo, B. T. Agena, K. R. Braghetto and R. Wassermann, “OntoMongo- Ontology-Based Data Access for NoSQL.”, *ONTOBRAS*, ed. by M. Abel, S. R. Fiorini and C. Pessanha, vol. 1908, CEUR Workshop Proceedings, CEUR-WS.org, 2017 55, URL: <http://dblp.uni-trier.de/db/conf/ontobras/%20ontobras2017.html#AraujoABW17> (cit. on p. 56).
- [156] J. Unbehauen and M. Martin, “Executing SPARQL queries over Mapped Document Stores with SparqlMap-M”, *12th International Conference on Semantic Systems Proceedings (SEMANTiCS 2016)*, SEMANTiCS '16, 2016 (cit. on p. 56).
- [157] E. Botoeva, D. Calvanese, B. Cogrel, M. Rezk and G. Xiao, “OBDA beyond relational DBs: A study for MongoDB”, CEUR Workshop Proceedings, 2016 (cit. on p. 56).
- [158] T. Wilmes, *SQL-Gremlin*, Accessed: 31-10-2018, 2016, URL: <https://github.com/twilmes/sql-gremlin> (cit. on p. 56).
- [159] R. Nambiar, N. Wakou et al., “Transaction Processing Performance Council (TPC): State of the Council 2010”, *Performance Evaluation, Measurement and Characterization of Complex Systems: Second TPC Technology Conference, TPCTC 2010, Revised Selected Papers*, 2011 (cit. on pp. 57, 59).
- [160] D. Dominguez-Sal, Urbón-Bayes et al., “Survey of Graph Database Performance on the HPC Scalable Graph Analysis Benchmark”, *Proceedings of the 2010 International Conference on Web-age Information Management, WAIM'10*, 2010 (cit. on pp. 57, 59).
- [161] R. C. Murphy, K. B. Wheeler, B. W. Barrett and J. A. Ang, *Introducing the GRAPH 500*, Cray User’s Group (CUG) (2010) (cit. on pp. 57, 59).
- [162] M. Dayarathna and T. Suzumura, “XGDBench: A benchmarking platform for graph stores in exascale clouds.”, *CloudCom*, 2012 (cit. on pp. 57, 59).
- [163] A.-C. N. Ngomo and M. Röder, *HOBBIT: Holistic Benchmarking for Big Linked Data*, ERCIM News **2016** (2016) (cit. on pp. 57, 59).
- [164] M. Morsey, J. Lehmann, S. Auer et al., “DBpedia SPARQL Benchmark – Performance Assessment with Real Queries on Real Data”, *The Semantic Web – ISWC 2011: 10th International Semantic Web Conference, Proceedings, Part I*, 2011 (cit. on pp. 57, 59).
- [165] Y. Guo, Z. Pan and J. Heflin, *LUBM: A Benchmark for OWL Knowledge Base Systems*, *Web Semantics* **3** (2005) (cit. on pp. 57, 59).
- [166] G. Aluç, O. Hartig, M. T. Özsu and K. Daudjee, “Diversified stress testing of RDF data management systems”, *International Semantic Web Conference*, Springer, 2014 197 (cit. on pp. 57, 59, 90, 135, 141).

-
- [167] M. Schmidt, T. Hornung et al., “SP2Bench: A SPARQL Performance Benchmark.”, *Semantic Web Information Management*, ed. by R. D. Virgilio, F. Giunchiglia and L. Tanca, 2009 (cit. on pp. 57, 59).
- [168] C. Bizer and A. Schultz, *Benchmarking the performance of storage systems that expose SPARQL endpoints*, World Wide Web Internet And Web Information Systems (2008) (cit. on pp. 57, 59, 90, 122, 135, 140).
- [169] A. Flores, G. Palma, M.-E. Vidal et al., “GRAPHIUM: visualizing performance of graph and RDF engines on linked data”, *Proceedings of the 2013th International Conference on Posters & Demonstrations Track-Volume 1035*, 2013 (cit. on pp. 58, 59, 153).
- [170] R. Angles, P. Boncz, J. Larriba-Pey, I. Fundulaki, T. Neumann, O. Erling, P. Neubauer, N. Martinez-Bazan, V. Kotsev and I. Toma, *The linked data benchmark council: a graph and RDF industry benchmarking effort*, *SIGMOD Record* **43** (2014) 27, URL: <http://doi.acm.org/10.1145/2627692.2627697> (cit. on pp. 58, 59, 153).
- [171] M. Saleem, R. Usbeck, M. Roder and A.-C. N. Ngomo, *SPARQL Querying Benchmarks*, (2016), URL: https://www.researchgate.net/publication/300005911_SPARQL_Querying_Benchmarks (cit. on p. 59).
- [172] M. Saleem, Q. Mehmood and A. N. Ngomo, “FEASIBLE: A Feature-Based SPARQL Benchmark Generation Framework”, *The Semantic Web - ISWC 2015 - 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part I*, 2015 52, URL: https://doi.org/10.1007/978-3-319-25007-6_4 (cit. on p. 59).
- [173] M. Saleem, M. I. Ali, A. Hogan, Q. Mehmood and A. N. Ngomo, “LSQ: The Linked SPARQL Queries Dataset”, *The Semantic Web - ISWC 2015 - 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part II*, 2015 261, URL: https://doi.org/10.1007/978-3-319-25010-6_15 (cit. on p. 59).
- [174] M. A. Bornea, J. Dolby, A. Kementsietsidis, K. Srinivas, P. Dantressangle, O. Udrea and B. Bhattacharjee, “Building an efficient RDF store over a relational database”, *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ACM, 2013 121 (cit. on p. 59).
- [175] R. Angles and C. Gutierrez, “An Introduction to Graph Data Management”, *Graph Data Management, Data-Centric Systems and Applications*, Springer Nature, 2018, chap. 1 1 (cit. on p. 61).
- [176] G. Carothers, *Notation3 (N3): A readable RDF syntax, W3C Team Submission*, <https://www.w3.org/TeamSubmission/2011/SUBM-n3-20110328/>, 2011 (cit. on p. 63).
- [177] F. Gandon and G. Schreiber, *RDF 1.1 XML Syntax, W3C Recommendation*, <https://www.w3.org/TR/2014/REC-rdf-syntax-grammar-20140225/>, 2014 (cit. on p. 63).

- [178] G. Carothers, *RDF 1.1 N-Triples, A line-based syntax for an RDF graph, W3C Recommendation*, <https://www.w3.org/TR/2014/REC-n-triples-20140225/>, 2014 (cit. on p. 63).
- [179] D. Beckett, T. Berners-Lee, E. Prud'hommeaux and G. Carothers, *RDF 1.1 Turtle, Terse RDF Triple Language, W3C Recommendation*, <https://www.w3.org/TR/turtle/>, 2014 (cit. on p. 63).
- [180] G. Carothers, *RDF 1.1 N-Quads, A line-based syntax for RDF datasets, W3C Recommendation*, <https://www.w3.org/TR/2014/REC-n-quads-20140225/>, 2014 (cit. on p. 63).
- [181] D. Tomaszuk and K. Litman, "DRPD: Architecture for Intelligent Interaction with RDF Prefixes", *DeSemWeb*, 2018, URL: <http://ceur-ws.org/Vol-2165/paper5.pdf> (cit. on pp. 64, 76).
- [182] W. O. W. Group, *OWL 2 Web Ontology Language Document Overview (Second Edition), W3C Recommendation*, <https://www.w3.org/TR/2012/REC-owl2-overview-20121211/>, 2012 (cit. on p. 64).
- [183] P. V. Biron, M. Sperberg-McQueen, S. Gao, A. Malhotra, H. Thompson and D. Peterson, *XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes, W3C Recommendation*, <http://www.w3.org/TR/2012/REC-xmlschema11-2-20120405/>, 2012 (cit. on p. 68).
- [184] D. Tomaszuk, R. Angles, L. Szeremeta, K. Litman and D. Cisterna, "Serialization for Property Graphs", *Proc. of the 15th International Conference Beyond Databases, Architectures and Structures (BDAS)*, 2019 (cit. on p. 89).
- [185] M. Pham and P. A. Boncz, "Exploiting Emergent Schemas to Make RDF Systems More Efficient", *Proc. of the International Semantic Web Conference*, LNCS, Springer, 2016 463 (cit. on p. 89).
- [186] M. A. Martínez-Prieto, J. D. Fernández, A. Hernández-Illera and C. Gutiérrez, "RDF Compression", *Encyclopedia of Big Data Technologies*, Springer International Publishing, 2018 1 (cit. on p. 90).
- [187] M. Schmidt, T. Hornung, G. Lausen and C. Pinkel, "SP²Bench: a SPARQL performance benchmark", *2009 IEEE 25th International Conference on Data Engineering*, IEEE, 2009 222 (cit. on p. 90).
- [188] C. Stadler, J. Lehmann, K. Höffner and S. Auer, *Linkedgeodata: A core for a web of spatial open data*, *Semantic Web* **3** (2012) 333 (cit. on p. 90).
- [189] D. Vrandečić and M. Krötzsch, *Wikidata: a free collaborative knowledgebase*, *Communications of the ACM* **57** (2014) 78 (cit. on p. 90).
- [190] V. Nguyen, H. Y. Yip, H. Thakkar, Q. Li, E. Bolton and O. Bodenreider, "Singleton property graph: Adding a semantic web abstraction layer to graph databases", *Proceedings of the 2nd International Semantic Web Conference (ISWC) Workshop on Contextualised Knowledge Graphs (CKG)*, 2019 (cit. on pp. 102, 185).

-
- [191] J. Hölsch and M. Grossniklaus, “An algebra and equivalences to transform graph patterns in Neo4j”, *EDBT/ICDT 2016 Workshops: EDBT Workshop on Querying Graph Structured Data (GraphQ)*, 2016 (cit. on pp. 103–105).
- [192] G. S. J. Marton, *Formalizing openCypher Graph Queries in Relational Algebra*, Published online on FTSRG archive (2017), URL: <http://docs.inf.mit.bme.hu/ingraph/> (cit. on pp. 103–105).
- [193] H. Thakkar, M. Dubey, G. Sejdin, A.-C. N. Ngomo, J. Debattista, C. Lange, J. Lehmann, S. Auer and M.-E. Vidal, *LITMUS: An open extensible framework for benchmarking RDF data management solutions*, arXiv preprint arXiv:1608.02800 (2016) (cit. on p. 103).
- [194] H. Thakkar, “Towards an Open Extensible Framework for Empirical Benchmarking of Data Management Solutions: LITMUS”, *14th Extended Semantic Web Conferences (ESWC 2017)*, 2017 (cit. on p. 103).
- [195] C. Li, K. C.-C. Chang, I. F. Ilyas and S. Song, “RankSQL: query algebra and optimization for relational top-k queries”, *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, ACM, 2005 131 (cit. on p. 105).
- [196] J. L. Reutter, *Graph Patterns: Structure, Query Answering and Applications in Schema Mappings and Formal Language Theory*, Edinburgh Research Archive (2013) (cit. on p. 105).
- [197] A. Gubichev and M. Then, “Graph Pattern Matching - Do We Have to Reinvent the Wheel?”, *Second International Workshop on Graph Data Management Experiences and Systems, GRADES 2014, co-located with SIGMOD/PODS 2014, Snowbird, Utah, USA, June 22, 2014*, 2014 8:1, URL: <http://doi.acm.org/10.1145/2621934.2621944> (cit. on p. 105).
- [198] R. Angles and C. Gutierrez, “The expressive power of SPARQL”, *International Semantic Web Conference*, Springer, 2008 114 (cit. on p. 108).
- [199] S. Harris, N. Lamb and N. Shadbolt, “4store: The design and implementation of a clustered RDF store”, *5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009)*, 2009 94 (cit. on p. 120).
- [200] N. Martinez-Bazan, S. Gomez-Villamor and F. Escale-Claveras, “DEX: A high-performance graph database management system”, *Data Engineering Workshops (ICDEW), 2011 IEEE 27th International Conference on*, IEEE, 2011 124 (cit. on p. 121).
- [201] H. Thakkar, R. Angles, M. Rodriguez, S. Mallette and J. Lehmann, *Gremlinator (sparql-gremlin) resources*, (2019), URL: https://figshare.com/articles/Gremlinator_sparql-gremlin_resources_txt/8187110 (cit. on pp. 122–124, 185).
- [202] H. P. Karanam, S. Neelam et al., “Scalable Reasoning Infrastructure for Large Scale Knowledge Bases”, *Proceedings of the ISWC 2018 Posters & Demo*. 2018, URL: <http://ceur-ws.org/Vol-2180/paper-30.pdf> (cit. on pp. 126, 131, 161).

- [203] J. Lehmann, G. Sejdou, L. Bühmann, P. Westphal, C. Stadler, I. Ermilov, S. Bin, N. Chakraborty, M. Saleem, A.-C. N. Ngomo and H. Jabeen, “Distributed semantic analytics using the sansa stack”, *International Semantic Web Conference*, Springer, 2017 147 (cit. on p. 132).
- [204] V. Nguyen, O. Bodenreider and A. Sheth, “Don’t like RDF reification?: making statements about statements using singleton property”, *Proc. of the 23rd International Conference on World Wide Web*, ACM, 2014 759 (cit. on p. 132).
- [205] C. Bizer and A. Schultz, *The Berlin SPARQL Benchmark*, *Int. J. of Semantic Web Inf. Syst.* **5** (2009) (cit. on p. 137).
- [206] G. Aluç, O. Hartig, M. T. Özsu et al., “Diversified stress testing of RDF data management systems”, *International Semantic Web Conference*, 2014 (cit. on p. 137).
- [207] J. D. Hunter, *Matplotlib: A 2D graphics environment*, *Computing In Science & Engineering* **9** (2007) 90 (cit. on p. 143).
- [208] J. M. Giménez-García, H. Thakkar and A. Zimmermann, “Ranking the Web of Data”, *3rd International Workshop on Dataset PROFiling and fEderated Search for Linked Data (PROFILES) 2016*, 2016 (cit. on pp. 146, 147, 185).
- [209] J. Debattista, S. Auer and C. Lange, “Luzzu – A Framework for Linked Data Quality Analysis”, *2016 IEEE International Conference on Semantic Computing, Laguna Hills*, 2016 (cit. on p. 146).
- [210] H. Thakkar, D. Punjani, S. Auer and M.-E. Vidal, “Towards an Integrated Graph Algebra for Graph Pattern Matching with Gremlin”, *28th International Conference on Database and Expert Systems Applications - DEXA 2017, Lyon, France*, vol. 10438, Springer, 2017 81 (cit. on p. 157).
- [211] R. Angles, P. A. Boncz et al., *The linked data benchmark council: a graph and RDF industry benchmarking effort*, *SIGMOD Record* **43** (2014) (cit. on p. 159).
- [212] C. Unger, C. Forascu et al., “Question Answering over Linked Data (QALD-5)”, *Working Notes of CLEF 2015, Toulouse, France*, 2015 (cit. on p. 161).
- [213] S. Shekarpour, K. M. Endris, A. Jaya Kumar, D. Lukovnikov, K. Singh, H. Thakkar and C. Lange, “Question answering on linked data: Challenges and future directions”, *Proceedings of the 25th International Conference Companion on World Wide Web*, International World Wide Web Conferences Steering Committee, 2016 693 (cit. on pp. 161, 185).
- [214] M. Dubey, D. Banerjee, D. Chaudhuri and J. Lehmann, “EARL: joint entity and relation linking for question answering over knowledge graphs”, *International Semantic Web Conference*, Springer, 2018 108 (cit. on p. 161).
- [215] D. Lukovnikov, A. Fischer, J. Lehmann and S. Auer, “Neural network-based question answering over knowledge graphs on word and character level”, *Proceedings of the 26th international conference on World Wide Web*, 2017 1211 (cit. on p. 161).

-
- [216] K. Singh, *Towards dynamic composition of question answering pipelines*, PhD thesis: Ph. D. thesis, University of Bonn, Germany, 2019 (cit. on p. 161).
- [217] J. M. Giménez-García, H. Thakkar and A. Zimmermann, “Assessing trust with pagerank in the web of data”, *European Semantic Web Conference*, Springer, 2016 293 (cit. on p. 185).
- [218] S. Gupta, V. Desai and H. Thakkar, *Social Data Analysis: A Study on Friend Rating Influence*, arXiv preprint arXiv:1702.07651 (2017) (cit. on p. 185).
- [219] K. Singh, I. Lytra, M.-E. Vidal, D. Punjani, H. Thakkar, C. Lange and S. Auer, “QAestro–Semantic-based Composition of Question Answering Pipelines”, *28th International Conference on Database and Expert Systems Applications - DEXA 2017, Lyon, France, 2017* (cit. on p. 185).
- [220] R. Samavi, M. P. Consens, S. Khatchadourian, V. Nguyen, A. P. Sheth, J. M. Giménez-García and H. Thakkar, eds., *Proceedings of the Blockchain enabled Semantic Web Workshop (BlockSW) and Contextualized Knowledge Graphs (CKG) Workshop co-located with the 18th International Semantic Web Conference, BlockSW/CKG@ISWC 2019, Auckland, New Zealand, October 27, 2019*, vol. 2599, CEUR Workshop Proceedings, CEUR-WS.org, 2020, URL: <http://ceur-ws.org/Vol-2599> (cit. on p. 185).
- [221] E. Kacupaj, J. Plepi, K. Singh, H. Thakkar, J. Lehmann and M. Maleshkova, “Conversational Question Answering over Knowledge Graphs with Transformer and Graph Attention Networks”, *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume, EACL 2021, Online, April 19 - 23, 2021*, ed. by P. Merlo, J. Tiedemann and R. Tsarfaty, Association for Computational Linguistics, 2021 850, URL: <https://www.aclweb.org/anthology/2021.eacl-main.72/> (cit. on p. 185).
- [222] H. Thakkar, D. Punjani, S. Auer and M.-E. Vidal, *Towards an integrated graph algebra for graph pattern matching with gremlin (extended version)*, arXiv preprint arXiv:1908.06265 (2019) (cit. on p. 185).
- [223] J. Plepi, E. Kacupaj, K. Singh, H. Thakkar and J. Lehmann, *Context Transformer with Stacked Pointer Networks for Conversational Question Answering over Knowledge Graphs*, CoRR **abs/2103.07766** (2021), arXiv: [2103.07766](https://arxiv.org/abs/2103.07766), URL: <https://arxiv.org/abs/2103.07766> (cit. on p. 185).

Full Results of the SPARQL - Gremlin Performance Comparison

Here, we report the detailed query-wise runtime results in tabular format corresponding to the plots reported previously in Figures [5.7](#) and [5.8](#) as shown in Section [5.5.3](#) of Chapter [5](#).

Query	Gremlin Traversal Execution Time (ms, without indexes)						SPARQL Query Execution Time (ms, with indexes)						Gremlin Traversal Execution Time (ms, with indexes)					
	Tinker (cold)	Tinker (warm)	Neo4j (cold)	Neo4j (warm)	Spaksee (cold)	Spaksee (warm)	Virtuoso (cold)	Virtuoso (warm)	Jena (cold)	Jena (warm)	4store (cold)	4store (warm)	Tinker (cold)	Tinker (warm)	Neo4j (cold)	Neo4j (warm)	Spaksee (cold)	Spaksee (warm)
C1	220.12	136.7	177.60	152.72	306.8	272.67	458.15	167.25	1652	1614	260	152.5	191.6	107.69	157.3	80.05	253.82	184
C2	302.6	187.3	272.67	231.36	340.84	224.48	61.25	21.5	390	361.25	232.5	65	238.1	100.52	218.46	97.14	279.01	203.7
C3	17.7	15.4	18.50	15.29	38.86	13.81	271	70.25	404.63	384	227.5	70	1.5	0.33	9.43	2.86	29.5	3.68
F1	16.32	15.4	19.72	16.40	32.66	22.08	172.25	45.5	448	417.5	275.25	72.5	17.83	15.27	11.93	4.75	26.2	3.49
F2	45.6	33.5	67.06	53.17	102.61	73.70	84.15	23.75	945.5	871.5	632	148.25	1.12	0.9	34.67	16.3	49.7	23.82
F3	22.8	20.1	29.37	25.81	44.93	25.34	43.3	15	812.75	904.3	655	132.5	1.15	0.72	2.4	1.15	29	3.77
L1	17.6	16	34.73	33.09	39.20	19.97	14.45	7.25	379.15	369.25	245	72.5	0.246	0.169	3.85	1.74	17.8	5.3
L2	107.17	56.57	71.37	32.05	105.26	68.42	44.5	21.25	456.75	416.5	287.5	55.3	77.9	49.41	45.26	17.98	43.8	21.52
L3	6.84	6.4	18.80	14.77	9.09	8.53	8.3	4.75	433	429.75	382.25	175	0.79	0.63	3.25	1.72	12.75	4.29
G1	42.57	28.87	50.14	22.49	48.11	43.54	61.45	22.5	811.34	544.21	465.19	132.75	18.52	8.01	28.3	4.18	29.3	9.29
G2	19.15	16.23	35.70	16.58	20.76	18.87	272.3	69.25	434.3	383.75	212.5	60.34	18.5	16.5	18.2	3.98	11.67	3.89
G3	260.47	149.1	248.53	156.58	306.06	287.54	77.25	50.5	916.5	858	682.5	130	269.5	163.3	133.4	54.52	212.82	174.25
Gc1	24.6	22.36	15.85	11.27	29.59	25.2	289	171.25	1535	1450	425	305	0.33	0.268	4.92	2.08	7.92	3.05
Gc2	23.47	21.36	21.49	17.09	27.04	24.85	110.3	94.25	2809.75	2717.3	2727.5	1110.25	0.737	0.679	6.07	2.34	5.88	3
Gc3	23.98	21.6	19.27	15.62	28.78	25.01	102	30	1063	1045.75	287.5	120.75	0.762	0.667	5.19	2.25	6.31	2.79
O1	328.57	192.77	238.3	173.95	461.59	402.64	183.5	164.7	1234.25	1200	207.45	130.5	294.68	193.44	210.2	77.75	331.9	227.24
O2	20	17.3	32.93	25.32	56.64	35.68	44.5	15.3	530	403.25	278	87.5	4.56	2.63	8.12	3.96	26.7	3.39
O3	525.81	357.98	489.63	369.2	612.04	483.51	604	519	2645.5	2109	650.9	321.65	205.1	158.18	196.87	153.76	338.74	179.82
U1	45.72	23.17	53.15	48.6	111.62	58.78	44.89	13.25	498.5	504	207.5	90	18.98	9.42	28.51	13.49	47.5	15.59
U2	373.95	203.61	402.40	378.1	503	286.24	34	10.75	444.5	403	190	55	257.5	173.2	159.01	68.63	307.53	152.7
U3	278.89	173.01	328.60	287.29	565.19	329.1	151.4	40.75	1111	1169	507.5	190	218.73	153.9	141.52	47.14	224.1	72.67
M1	1221.7	765.7	246.25	116.57	453.82	367.93	74.15	58.5	3445.15	3417.5	1535	498	899	618.25	126.33	75.79	226.23	130.7
M2	806.1	537.4	283.76	161.43	327.01	276.04	120.6	98.25	3513.75	3418.25	2857.5	1412.5	657.6	451	143.61	87.81	139.6	55.84
M3	551.6	309.32	161.72	81.63	261.92	168.91	84	75.5	3384.25	3367	1080	343	402.7	235.9	66.48	34.36	102.35	39.37
S1	20.75	15.5	25.38	16.72	38.83	32.79	388.15	108.5	593	630	253	70	1.55	0.79	2.54	1.97	12.91	3.94
S2	17.5	15.4	21.16	14.8	26.45	21.89	332	89.5	442	413	202.5	80	1.45	0.82	2.39	1.71	10.18	3.42
S3	34.02	26.65	40.75	31.54	52.57	40.2	40641	40329	22663	21180	2175	1730	7.02	4.46	15.92	8.02	31.2	6.84

Table A.1: Performance comparison of SPARQL queries vs the translated Gremlin traversals in both cold and warm caches for the BSBM dataset.

Query	Gremlin Traversal Execution Time (ms, without indexes)						SPARQL Query Execution Time (ms, with indexes)						Gremlin Traversal Execution Time (ms, with indexes)					
	Tinker (cold)	Tinker (warm)	Neo4j (cold)	Neo4j (warm)	Spaksee (cold)	Spaksee (warm)	Virtuoso (cold)	Virtuoso (warm)	Jena (cold)	Jena (warm)	4store (cold)	4store (warm)	Tinker (cold)	Tinker (warm)	Neo4j (cold)	Neo4j (warm)	Spaksee (cold)	Spaksee (warm)
C1	11.49	5.45	17.7	13.3	23.89	13.8	36.25	3.75	364	374	298	93	7.75	4.25	12.87	6.17	14.4	6.64
C2	20.757	11.8	28.3	21.2	31.13	15.7	25	2.75	395	389	290	80	11.52	6.14	18.4	5.26	23.9	10.52
C3	10.76	5.84	18.9	12.6	24	11.4	52	4.25	399	392	318	90	7.5	4.25	12.38	4.58	15.17	7.1
F1	0.77	0.8	3.49	2.7	4.45	2.4	12	4.33	363	369	273	115	0.53	0.29	2.16	1.03	2.45	1.32
F2	15.45	8.47	24.5	12.09	34.3	17.62	6	2	370	427	270	124	9.6	6.25	19.52	9.26	24.4	11.08
F3	2.4	1.24	6.68	5.41	11.28	9.7	5	2.65	470	373	252	115	1	0.72	8.15	3.38	7.6	3.46
L1	1.5	0.73	4.49	3.6	5.3	4.09	7.75	5	439	349	280	112	0.55	0.4	3.01	1.39	4.5	2.16
L2	2.96	0.47	7.43	4.6	4.75	2.8	12	2.5	364	364	295	113	0.39	0.38	4.12	1.83	4.57	2.25
L3	17.01	8.98	29.1	18.42	37.5	24.2	4	2.5	410	378	279	108	11.43	6.42	21.68	8.09	27.1	15.32
G1	6	2.15	12.16	8.81	4.39	3.2	64	18.25	433	461	245	103	1.46	1.11	3.22	1.26	3.04	2.27
G2	5.14	1.8	10.6	5.5	3.62	2.7	27	9.75	348	369	285	92	2.29	0.9	3.72	2.35	2.8	2.19
G3	6.32	1.5	13.27	9.76	4.86	3.8	2.75	2	471	476	328	157	1.34	0.85	5.25	3.99	3.95	3.04
Gc1	10.2	5.45	17.3	13.6	15.49	9.52	9	6.33	440	467	247	88	7.25	3.9	10.83	7.05	9.66	4.6
Gc2	25	14.09	38.1	23.9	36.6	15.67	6	2.75	385	354	250	105	15.92	9.25	23.62	9.84	18.9	7.01
Gc3	0.783	0.68	6.89	3.7	4.61	3.8	47.75	4.25	753	775	248	118	0.52	0.47	3.42	3.33	3.52	3.43
O1	18.25	12.23	28	17.4	40.6	25.3	4.5	2.25	446	483	267	125	12.32	7.5	19.1	8.44	27.46	12.2
O2	28	17.4	39.2	23	54.3	27.81	20	8.25	321	354	270	95	19.2	11.93	22.3	6.85	35.49	13.8
O3	18.45	10.44	28.3	18.02	42.78	24.64	3.75	2.25	545	573	300	120	10.65	6.6	18.4	5.41	28.27	11.3
U1	67.25	36.75	70.6	46.25	81.72	39.78	27.75	3.75	377	368	240	115	43.75	24.11	35.64	15.46	45.34	28.76
U2	68.5	35.45	89.8	42.61	107.76	48.28	16.5	2.75	529	536	252.5	130	40	23.25	53.21	29.71	69.2	33.71
U3	75.5	42.6	98.5	54.4	157.6	62.91	31.5	4.5	346	434	260	112.5	44.5	23.58	65.31	28.33	120.49	73.79
M1	22.45	12.4	37.8	23.4	49.14	24.5	12	6.25	457	473	325	140	12.87	6.9	18.09	7.97	24.1	13.8
M2	2.56	1.65	6.94	3.67	51.2	26.3	8	3	373	423	301	118	1.6	1.13	6.02	3.26	28.6	15.36
M3	2.5	1.92	7	4.16	50.8	25.4	35	14.5	563	547	298	110	1.8	1.25	4.07	3.38	27.8	15.59
S1	35.8	28.4	42.6	20.9	35.6	15.9	176	67.5	451	372	278	135	32.5	17.3	26.47	12.15	21.4	10.63
S2	7.85	5.03	11.8	5.3	21.4	10.8	50	24.75	328	369	253	98	4.09	3.25	2.25	2.02	18.9	8.51
S3	4.23	1.42	6.9	3.8	17.6	8.3	902	824	362	406	250	80	1.25	0.82	9.15	5.01	11.9	6.02

Table A.2: Performance comparison of SPARQL queries vs the translated Gremlin traversals in both cold and warm caches for the Northwind dataset.

Complete List of Publications

Following is the complete list of publications and articles¹ produced during the development of this Ph.D. thesis (07-2015 to 05-2020).

Journal Papers (peer reviewed):

1. Renzo Angles, **Harsh Thakkar**, and Dominik Tomaszuk. *Mapping RDF Databases to Property Graphs*. In IEEE Access, Vol. 8, 2020. DOI: [10.1109/ACCESS.2020.2993117](https://doi.org/10.1109/ACCESS.2020.2993117)
2. Dominik Tomaszuk, Renzo Angles, and **Harsh Thakkar**. *PGO: Describing Property Graphs in RDF*. In IEEE Access, Vol. 8, 2020. DOI: [10.1109/ACCESS.2020.3002018](https://doi.org/10.1109/ACCESS.2020.3002018)

Conference Papers (peer reviewed):

3. **Harsh Thakkar**, Renzo Angles, Marko Rodriguez, Stephen Mallette, and Jens Lehmann. *Let's build Bridges, not Walls: SPARQL Querying of TinkerPop Graph Databases with Sparql-Gremlin*. In Proceedings of the IEEE 14th International Conference on Semantic Computing (ICSC), pp. 408-415, San Diego, USA, 2020. DOI: [10.1109/ICSC.2020.00080](https://doi.org/10.1109/ICSC.2020.00080) [**Best Paper Award**]
4. **Harsh Thakkar**, Dharmen Punjani, Sören Auer, and Maria-Esther Vidal. *Towards an Integrated Graph Algebra for Graph Pattern Matching with Gremlin*. In Proceedings of the 28th International Conference on Database and Expert Systems Applications (DEXA 2017), Lyon, France, pp. 81-91. Springer, 2017. DOI: [10.1007/978-3-319-64468-4_6](https://doi.org/10.1007/978-3-319-64468-4_6)
5. **Harsh Thakkar**, Yashwant Keswani, Mohnish Dubey, Jens Lehmann, and Sören Auer. *Trying Not to Die Benchmarking – Orchestrating RDF and Graph Data Management Solution Benchmarks using LITMUS*. In Proceedings of the 13th International Conference on Semantic Systems (SEMANTiCS 2017), Amsterdam, Nederland, pages 120-127. ACM, 2017. [**Best Paper Award**] DOI: [10.1145/3132218.3132232](https://doi.org/10.1145/3132218.3132232)

¹ Both peer reviewed [34, 35, 38–40, 42, 43, 45–48, 104, 106, 190, 208, 213, 217–221] (such as conference, workshop, journal, poster & demo papers) and not peer reviewed [37, 41, 104, 105, 201, 222, 223] (such as ArXiv and technical reports [Apache TinkerPop [18] sparql-gremlin documentation])

6. Kemele M Endris, Josè M. Gimenez-García, **Harsh Thakkar**, Elena Demidova, Antoine Zimmermann, Christoph Lange, and Elena Simperl. *Dataset Reuse: An Analysis of References in Community Discussions, Publications and Data*. DOI: [10.1145/3148011.3154461](https://doi.org/10.1145/3148011.3154461)
7. **Harsh Thakkar**. *Towards an Open Extensible Framework for Empirical Benchmarking of Data Management Solutions: LITMUS*. In Proceedings of the 14th Extended Semantic Web Conferences (ESWC 2017), 2017. DOI: [10.1007/978-3-319-58451-5_20](https://doi.org/10.1007/978-3-319-58451-5_20)
8. **Harsh Thakkar**, Kemele M. Endris, Josè M. Gimenez-García, Jeremy Debattista, Christoph Lange, and Sóren Auer. *Are Linked Datasets Fit for Open-domain Question Answering? A Quality Assessment*. In Proceedings of the 6th International Conference on Web Intelligence, Mining and Semantics (WIMS 2016), Nîmes, France, June 13-15, pages 1-12, 2016. DOI: [10.1145/2912845.2912857](https://doi.org/10.1145/2912845.2912857)

Workshop Papers (peer reviewed):

9. **Harsh Thakkar**, Maria-Esther Vidal, Sóren Auer. *Formalizing Gremlin Pattern Matching Traversals in an Integrated Graph Algebra (extended version)*. In Proceedings of the 2nd International Semantic Web Conference (ISWC) Workshop on Contextualised Knowledge Graphs (CKG), New Zealand, 2019. URL: http://ceur-ws.org/Vol-2599/CKG2019_paper_2.pdf
10. Renzo Angles, **Harsh Thakkar**, and Dominik Tomaszuk. *RDF and Property Graphs Interoperability: Status and Issues*. In Proceedings of the 13th Alberto Mendelzon International Workshop on Foundations of Data Management, Asunción (AMW 2019), Paraguay, June 3-7, 2019. URL: <http://ceur-ws.org/Vol-2369/paper01.pdf>
11. Vinh Nguyen, Hong Yung Yip, **Harsh Thakkar**, Qingliang Li, Evan Bolton, and Olivier Bodenreider. *Singleton property graph: Adding a semantic web abstraction layer to graph databases*. In Proceedings of the 2nd International Semantic Web Conference (ISWC) Workshop on Contextualised Knowledge Graphs (CKG), New Zealand, 2019. URL: http://ceur-ws.org/Vol-2599/CKG2019_paper_4.pdf
12. Felipe Quécole, Romao Martines, Josè M. Gimenez-García, and **Harsh Thakkar**. *Towards Capturing Contextual Semantic Information about Statements in Web Tables*. In Joint Proceedings of the International Workshops on Contextualized Knowledge Graphs, and Semantic Statistics (CKGSemStats) co-located with 17th International Semantic Web Conference (ISWC 2018), USA, 2018. URL: <http://ceur-ws.org/Vol-2317/article-09.pdf>
13. Kuldeep Singh, Ioanna Lytra, Maria-Esther Vidal, Dharmen Punjani, **Harsh Thakkar**, Christoph Lange, and Sóren Auer. *Qaestro - Semantic-based Composition of Question Answering Pipelines*. In 28th International Conference on Database and Expert Systems Applications (DEXA 2017), France, 2017. DOI: [10.1007/978-3-319-64468-4_2](https://doi.org/10.1007/978-3-319-64468-4_2)
14. Saeedeh Shekarpour, Kemele M Endris, Ashwini Jaya Kumar, Denis Lukovnikov, Kuldeep Singh, **Harsh Thakkar**, and Christoph Lange. *Question Answering on Linked Data: Challenges and Future Directions*. In Companion Proceedings of the 25th International

Conference Companion on World Wide Web (WWW), pages 693-698. 2016. DOI: [10.1145/2872518.2890571](https://doi.org/10.1145/2872518.2890571)

15. Josè M. Gimenez-Garçia, **Harsh Thakkar**, and Antoine Zimmermann. *Assessing Trust with Pagerank in the Web of Data*. In Proceedings of the 3rd International Workshop on Dataset PROFiling and fEderated Search for Linked Data (PROFILES '16) co-located with the 13th ESWC 2016 Conference, Greece, 2016. [**Best Paper Award**] DOI: [10.1007/978-3-319-47602-5_45](https://doi.org/10.1007/978-3-319-47602-5_45)

Poster & Demo Papers (peer reviewed):

16. **Harsh Thakkar**, Dharmen Punjani, Jens Lehmann, and Sörenen Auer. *Two for one: Querying Property Graph Databases using SPARQL via GREMLINATOR*. In Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and NetworkData Analytics (NDA), page 12, ACM, USA, 2018. DOI: [10.1145/3210259.3210271](https://doi.org/10.1145/3210259.3210271)
17. Yashwant Keswani, **Harsh Thakkar**, Mohnish Dubey, Jens Lehmann, and Sören Auer. *The LITMUS Test: Benchmarking RDF and Graph Data Management Systems*. In Proceedings of the CEUR-WS (Poster & Demo), SEMANTiCS 2017, Nederland, 2017.

Pre-prints (not peer reviewed):

18. **Harsh Thakkar**, Renzo Angles, Dominik Tomaszuk, and Jens Lehmann. *Direct Mappings between RDF and Property Graph Databases*. Pre-print arXiv preprint arXiv:1912.02127, 2019. URL: <http://arxiv.org/abs/1912.02127>
19. Mohamed Nadjib Mami, Damien Graux, **Harsh Thakkar**, Simon Scerri, Sören Auer, and Jens Lehmann. *The Query Translation Landscape: A Survey*. Pre-print arXiv:1910.03118, pp. 1-25, 2019. URL: <http://arxiv.org/abs/1910.03118>
20. **Harsh Thakkar**, Dharmen Punjani, Yashwant Keswani, Jens Lehmann, and Sören Auer. *A Stitch in Time Saves Nine – SPARQL Querying of Property Graphs using Gremlin Traversals*. Pre-print arXiv:1801.02911, pp. 1-24, 2018. URL: <http://arxiv.org/abs/1801.02911>
21. **Harsh Thakkar**, Dharmen Punjani, Jens Lehmann, and Sörenen Auer. *Killing Two Birds with One Stone - Querying Property Graphs using SPARQL via GREMLINATOR*. Pre-print arXiv:1801.09556. URL: <http://arxiv.org/abs/1801.09556>
22. **Harsh Thakkar**, Mohnish Dubey, Gezim Sejdiu, Axel-Cyrille Ngonga Ngomo, Jeremy Debattista, Christoph Lange, Jens Lehmann, Sörenen Auer, and Maria-Esther Vidal. *LITMUS: An Open Extensible Framework for Benchmarking RDF Data Management Solutions*. Pre-print arXiv:1608.02800, 2016. URL: <http://arxiv.org/abs/1608.02800>
23. **Harsh Thakkar**, Maria-Esther Vidal, Sören Auer. *Towards an Integrated Graph Algebra for Graph Pattern Matching with Gremlin (Extended Version)*. Pre-print arXiv:1908.06265. URL: <http://arxiv.org/abs/1908.06265>

Edited Volumes & Online Resources:

24. Reza Samavi, Mariano P. Consens, Shahan Khatchadourian, Vinh Nguyen, Amit P. Sheth, José M. Giménez-García, and **Harsh Thakkar**. *Proceedings of the Blockchain enabled Semantic Web Workshop (BlockSW) and Contextualized Knowledge Graphs (CKG) Workshop* co-located with the 18th International Semantic Web Conference, BlockSW/CKG@ISWC, Auckland, New Zealand, 2019. URN: [urn:nbn:de:0074-2599-1](https://nbn-resolving.org/urn:nbn:de:0074-2599-1)
25. **Harsh Thakkar**, Renzo Angles, Marko Rodriguez, Stephen Mallette, Dharmen Punjani, Jens Lehmann, Sören Auer. "Gremlinator (sparql-gremlin) Resources." available online at <https://doi.org/10.6084/m9.figshare.8187110.v3>, 2019.
26. Renzo Angles, **Harsh Thakkar**, and Dominik Tomaszuk. "RDF2PG Experimental Datasets." available online at <https://doi.org/10.6084/m9.figshare.12021156.v10>, 2020.

Working Papers:

27. **Harsh Thakkar**, Renzo Angles, Marko Rodriguez, Sören Auer. *GREMLINATOR: SPARQL Querying of Property Graph Databases using Gremlin Traversals*. IEEE Access submission, 2020. (in progress)
28. **Harsh Thakkar**, Renzo Angles, and Dominik Tomaszuk. *RDF2PG: Automatic Transformation of RDF to Property Graphs*. Demo paper, *Venue TBD*, 2020. (in progress)

Best Paper Awards

Following is the list of the Best Paper Awards that were conferred to a part of the contributions made during the term of this dissertation (07-2015 to 04-2020).

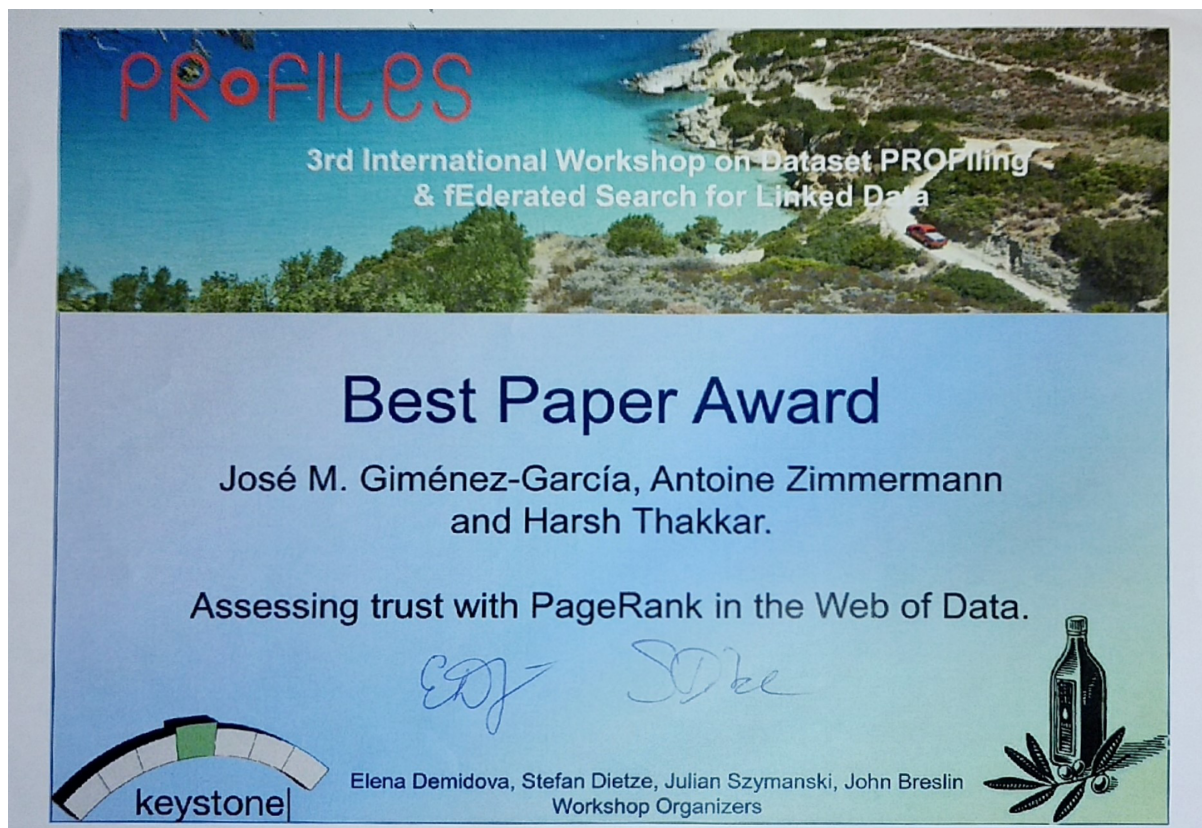


Figure C.1: Best Paper Award at the 3rd International Workshop on Dataset PROFiling and fEderated Search for Linked Data (PROFILES '16) at the 13th ESWC 2016 Conference, Greece, 2016.



Figure C.2: Best Paper Award – Research & Innovation Track at the 13th International Conference on Semantic Systems (SEMANTiCS 2017), Amsterdam, Netherlands, 2017.

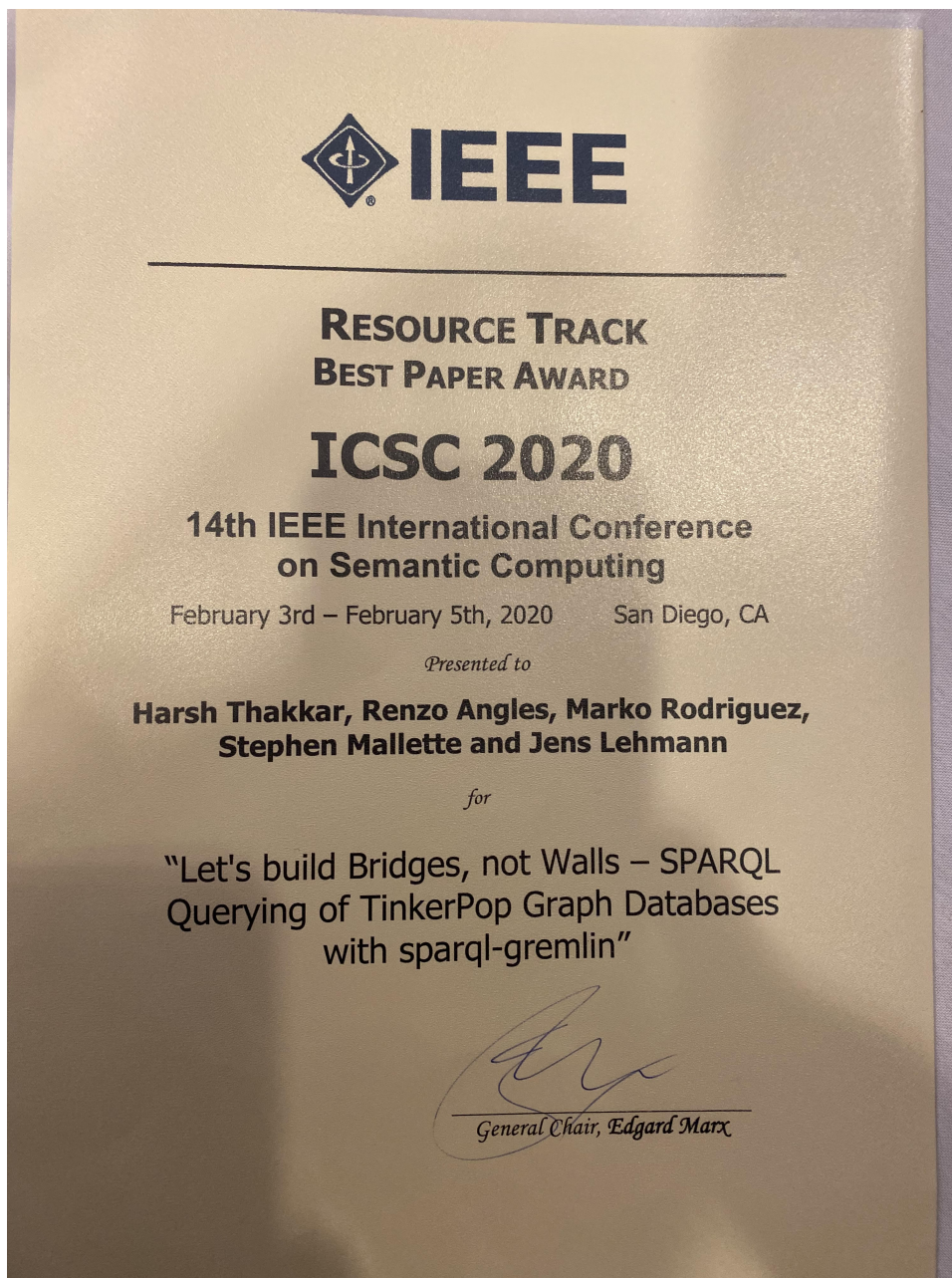


Figure C.3: Best Paper Award – Resource Track at the the IEEE 14th International Conference on Semantic Computing (ICSC 2020) San Diego, USA, 2020.

List of Figures

1.1	Layered Cake Diagram.	8
1.2	General Framework.	9
1.3	Research Questions.	10
1.4	Data Mapping.	12
1.5	Query Mapping.	13
1.6	Automated Benchmarking RDF and Property graph Databases.	14
2.1	Five Star Linked Open Data Principle	22
2.2	Example of an RDF graph.	24
2.3	The evaluation of a SPARQL SELECT query	27
2.4	Ranked list of various RDF Triplestores	29
2.5	Ranked list of various Graph databases	30
2.6	Example of a Property graph.	32
2.7	The Apache TinkerPop stack.	36
2.8	The TinkerPop-enabled graph systems.	37
2.9	The adoption of Knowledge Graphs.	41
2.10	The most value creating companies using Knowledge Graphs.	42
3.1	The <i>data interoperability</i> State-of-the-Art diagram	49
3.2	The <i>query interoperability</i> State-of-the-Art diagram	54
4.1	A graphical illustration of an RDF graph describing information about Elon Musk and Tesla Incorporation. We use abbreviated IRIs throughout this chapter.	66
4.2	The Schema of the RDF graph shown in Figure 4.1.	70
4.3	A graphical representation of a Property Graph.	72
4.4	The Schema of the Property Graph as shown in Fig. 4.3.	74
4.5	Property graph obtained after applying the instance mapping \mathcal{IM}_1 to the RDF graph shown in Figure 4.1.	78
4.6	A generic Property graph schema.	80
4.7	Property graph obtained after applying the instance mapping \mathcal{IM}_2 to the RDF graph shown in Figure 4.1.	82
4.8	Property graph obtained after applying the instance mapping \mathcal{IM}_3 to the RDF graph shown in Figure 4.1.	86
4.9	Scalability of the database mappings with respect to the size of the input data. The X-axis has graph sizes and the Y-axis has runtimes (in log scale). The “Base” line indicates the sizes of the input graphs in the log scale.	94
4.10	Scalability of the simple database mapping with respect to the hardware. The X-axis has virtual machines and the Y-axis has runtimes (in log scale).	95

4.11	Scalability of the generic database mapping with respect to the hardware. The X-axis has virtual machines and the Y-axis has runtimes (in log scale).	96
4.12	Scalability of the complete database mapping with respect to the hardware. The X-axis has virtual machines and the Y-axis has runtimes (in log scale).	97
4.13	Graphical representation of the property graph produced by applying the simple database mapping over the RDF graph G2.	98
5.1	An example of a Property graph from the Apache TinkerPop project network. . .	107
5.2	Graphical illustration of BGPs on PGs	108
5.3	Conceptual architecture for formalizing a Gremlin traversal using graph relation algebra.	110
5.4	Transformation Example 1. Illustration of the transformation of an input SPARQL query (Q) to the corresponding Gremlin pattern matching traversal (Ψ).	115
5.5	Transformation Example 2. Illustration of the transformation of an input SPARQL query (Q) to the corresponding Gremlin pattern matching traversal (Ψ).	116
5.6	The GREMLINATOR (<code>sparql-gremlin</code>) query translation pipeline.	118
5.7	Performance comparison of SPARQL queries vs the translated Gremlin traversals for BSBM dataset with respect to RDF and Graph DMSs in different configuration settings.	127
5.8	Performance comparison of SPARQL queries vs the translated Gremlin traversals for Northwind dataset with respect to RDF and Graph DMSs in different configuration settings.	128
6.1	The architectural overview of the LITMUS BENCHMARK SUITE.	137
6.2	The component driven architecture of the first working prototype of LITMUS BENCHMARK SUITE.	139
6.3	The graphical user interface of the LITMUS BENCHMARK SUITE.	139
6.4	CPU Migrations for Query 20 in warm cache.	151
6.5	CPU Migrations for Query 20 in cold cache.	151
6.6	Page Faults - loading the Northwind dataset.	152
6.7	Instructions - loading the Northwind dataset.	152
7.1	The three sub-research questions.	156
C.1	Best Paper Award at the 3 rd International Workshop on Dataset PROFiling and Federated Search for Linked Data (PROFILES '16) at the 13 th ESWC 2016 Conference, Greece, 2016.	189
C.2	Best Paper Award – Research & Innovation Track at the 13 th International Conference on Semantic Systems (SEMANTiCS 2017), Amsterdam, Netherlands, 2017.	190
C.3	Best Paper Award – Resource Track at the the IEEE 14th International Conference on Semantic Computing (ICSC 2020) San Diego, USA, 2020.	191

List of Tables

3.1	A consolidated summary of related work supporting <i>data interoperability</i> between RDF and Property graphs. Here, B.N. refers to whether the approach supports Blank Nodes, Reif. refers to whether the approach supports RDF reification, I.P. refers to whether the approach is Information Preserving, and the “-” refers to the lack of evidence in the respective work. The type of the arrow in the column “ Target ” represents whether the proposed transformation is omni-directional or bi-directional.	46
3.2	A consolidated summary of the SPARQL language features supported in various SPARQL \leftrightarrow X query translation approaches. Here, ✓ refers to features that are supported, ✗ implies features that are not supported, and ? implies features that have not been (clearly) mentioned in the respective study. The <i>Others</i> column reports the features provided only by individual works.	56
3.3	A consolidated summary of the State-of-the-Art in benchmarking frameworks for Relational, RDF and Graph Data Management Systems.	59
4.1	Datasets used in the experimental evaluation.	90
4.2	RDF Graphs used in the experimental evaluation.	90
4.3	Virtual Machines (Google Cloud Platform) used in the experimental evaluation.	91
4.4	RDF Schemas used in the experimental evaluation. This table shows the number of resource classes, property classes, and datatype definitions.	91
4.5	Runtimes (in milliseconds) for the simple data mapping . Undefined runtimes are represented with “?”.	92
4.6	Runtimes (in milliseconds) for the generic data mapping . Undefined runtimes are represented with “?”.	92
4.7	Runtimes (in milliseconds) for the complete data mapping . Undefined runtimes are represented with “?”.	93
4.8	Size (in bytes) of the output files produced during the experimental evaluation of the database mappings (SDM, GDM and CDM). PG and PGS mean property graph and property graph schema respectively.	93
5.1	A consolidated list of graph relational algebra operators with their corresponding instruction steps in the Gremlin traversal language.	105
5.2	Correspondence between the SPARQL triple patterns and single step traversals (SSTs) from the Gremlin instruction library. Each of the SPARQL triple pattern can be mapped to a particular Gremlin single step traversal.	113
5.3	A consolidated summary of the SPARQL constructs and keywords along with their corresponding Gremlin constructs and instruction steps.	115

5.4	Northwind RDF and Property graph dataset statistics	121
5.5	BSBM RDF and Property graph dataset statistics	122
5.6	A description of the feature compositions in each of the query dataset.	123
5.7	Comparison of results of a subset of SPARQL queries and their corresponding Gremlin traversals for the BSBM dataset.	125
6.1	Feature distribution in preset queries provided with the LITMUS BENCHMARK SUITE.	143
6.2	Linked Open Data quality assessment dimensions and metrics relevant to open domain Question Answering systems.	146
6.3	The loading time (in seconds) performance comparison for Northwind (respective versions) in all the DMSs. The highest and lowest values for mean are shown using the bold and <i>italic</i> fonts respectively.	148
6.4	The warm cache execution time (in seconds) performance comparison for running Query 14 (C1) (respective version) on all DMSs. The highest and lowest values for mean are shown using the bold and <i>italic</i> fonts respectively.	149
6.5	The cold cache execution time (in seconds) performance comparison for running Query 14 (C1) (respective version) on all DMSs. The highest and lowest values for mean are shown using the bold and <i>italic</i> fonts respectively.	149
A.1	Performance comparison of SPARQL queries vs the translated Gremlin traversals in both cold and warm caches for the BSBM dataset.	182
A.2	Performance comparison of SPARQL queries vs the translated Gremlin traversals in both cold and warm caches for the Northwind dataset.	183

List of Algorithms

1	GREMLINATOR SPARQL-to-Gremlin query mapping algorithm	118
---	---	-----

Listings

2.1	An Example of SPARQL SELECT query	26
2.2	An Example of SPARQL ASK query	26
2.3	Gremlin query (<i>imperative</i>) for the question “Elon Musk is the CEO of which organisation?”	34
2.4	Gremlin query (<i>declarative</i>) for the question “Elon Musk is the CEO of which organisation?”	34
2.5	A snippet of the triples describing the city of Bonn (http://dbpedia.org/resource/Bonn)	40
5.1	Return the age of the oldest person marko knows	106
5.2	Gremlin traversal for "What is created by Marko?"	107
5.3	This traversal returns the names of people who created a project named 'lop' that was also created by someone who is 30 years old.	108
5.4	This traversal returns the list all the persons in the ascending order of the age.	111
5.5	This traversal returns the list of all the people who have collaboratively created a software.	112
6.1	Query 14 (C1) in SPARQL.	149
6.2	Query 14 (C1) in Gremlin.	149
6.3	Query 20 (Gc2) in SPARQL.	150
6.4	Query 20 (Gc2) in Gremlin.	150