High-performance Query Processing over Knowledge Graphs

Masoud Salehpour

A thesis submitted to fulfil requirements for the degree of

Doctor of Philosophy

School of Computer Science

Faculty of Engineering

The University of Sydney

May 2022

Statement of Originality

This is to certify that to the best of my knowledge, the content of this thesis is my own work. This thesis has not been submitted for any degree or other purposes.

I certify that the intellectual content of this thesis is the product of my own work and that all the assistance received in preparing this thesis and sources have been acknowledged.

Masoud Salehpour
May 04, 2022

Authorship Attribution

The results presented in this dissertation were published in the following publications and can be found in the relevant chapters:

(1) Chapter 2 of this thesis contains material under submission and available as preprints [1, 2].

I was the primary author who proposed the concept, designed the main techniques, implemented the system, designed and conducted the evaluation, and wrote the drafts for the submission.

[1] Masoud Salehpour, Joseph G. Davis, "Knowledge Graphs for Processing Scientific Data: Challenges and Prospects", In: *CoRR abs/2004.06203 (2020). arXiv: 2004.06203.* url: https://arxiv.org/abs/2004.06203.

[2] Masoud Salehpour, Joseph G. Davis, "The Effects of Different JSON Representations on Querying Knowledge Graphs", In: *CoRR abs/2004.04286 (2020). arXiv: 2004.04286.* url: https://arxiv.org/abs/2004.04286.

(2) Chapter 3 of this thesis contains material under submission and available as a preprint [1]. It also contains material which is published in [3, 4]. It is also available as a preprint [5].

I was the primary author who proposed the concept, designed the main techniques, implemented the system, designed and conducted the evaluation, and wrote the drafts for the submission.

[1] Masoud Salehpour, Joseph G. Davis, "Knowledge Graphs for Processing Scientific Data: Challenges and Prospects", In: *CoRR abs/2004.06203 (2020). arXiv: 2004.06203.* url: https://arxiv.org/abs/2004.06203.

[3] Masoud Salehpour, Joseph G. Davis, "A Comparative Analysis of Knowledge Graph Query Performance," In: Proc.

of Int. Conf. on Transdisciplinary AI (TransAI). 2021, pp. 33-40, doi:10.1109/TransAI51903.2021.00014.

[5] Masoud Salehpour, Joseph G. Davis, "A Comparative Analysis of Knowledge Graph Query Performance", In: *CoRR abs/2004.05648 (2020). arXiv: 2004.05648*. url: https://arxiv.org/abs/2004.05648.

[4] Masoud Salehpour, Joseph G. Davis, "Towards Diversity-Tolerant RDF-Stores", In: Proc. of the ACM Symposium on Applied Computing (SAC). 2022, pp. *to-appear*.

(3) Chapter 4 of this thesis contains material which is published in [6]. It also contains material from this preprint [5].

I was the primary author who proposed the concept, designed the main techniques, implemented the system, designed and conducted the evaluation, and wrote the drafts for the submission.

[5] Masoud Salehpour, Joseph G. Davis, "A Comparative Analysis of Knowledge Graph Query Performance", In: *CoRR abs/2004.05648 (2020). arXiv: 2004.05648*. url: https://arxiv.org/abs/2004.05648.

[6] Masoud Salehpour, Joseph G. Davis, "SymphonyDB: A Polyglot Model for Knowledge Graph Query Processing," In: Proc. of Int. Conf. on Transdisciplinary AI (TransAI). 2021, pp. 25-32, 10.1109/TransAI51903.2021.00013.

(4) Chapter 5 of this thesis contains material from [3, 5].

[3] Masoud Salehpour, Joseph G. Davis, "A Comparative Analysis of Knowledge Graph Query Performance," In: Proc. of Int. Conf. on Transdisciplinary AI (TransAI). 2021, pp. 33-40, doi:10.1109/TransAI51903.2021.00014.

[5] Masoud Salehpour, Joseph G. Davis, "A Comparative Analysis of Knowledge Graph Query Performance", In: *CoRR abs/2004.05648 (2020). arXiv: 2004.05648*. url: https://arxiv.org/abs/2004.05648.

In addition to the statements above, in cases where I am not the corresponding author of a published item, permission to include the published material has been granted by the corresponding author.

Masoud Salehpour

Date: May 04, 2022

As supervisor for the candidature upon which this thesis is based, I can confirm that the authorship attribution statements above are correct.

Prof. Joseph G. Davis

Date: May 04, 2022

In Memory of My Deeply Missed Father
Dedicated to My Wonderful Mother

Acknowledgements

Every day in my PhD was an adventure and an exciting opportunity to widen my experience and deepen my understanding of problems and solutions. Although unconventional, I am extremely blessed to run out of space writing quite a long list of acknowledgments.

I am truly indebted to my research adviser, Joseph G. Davis who took me under his wing from the first day, encouraging me to attack real-world practical problems, helping me solidify hypotheses, and guiding me to conduct research independently. Although there are not enough words to show my appreciation, I wish you a lifetime of happiness, Joseph. Thank you also to my reviewers for their efforts and detailed reviews.

I would also like to thank other senior researchers of our school: Albert Zomaya, Alan Fekete, Joachim Gudmundsson, Bernhard Scholz, and Simon Poon with whom I have had many intellectually stimulating conversations over the last few years.

Thanks especially to Asadollah Shahbahrami who introduced me to the world of research, serving as my master research adviser; to all other friends and professors who helped me when I was doing my undergraduate and master.

I would also like to thank all the former and current members of the following research groups: DataBase Research Group (DBRG), Center for Distributed and High-Performance Computing, Knowledge Discovery and Management Research Group (KDMRG), Sydney Algorithms and Computing Theory (SACT), Concurrent Systems, Systems Lab, Programming Language Research, Computational Logic for Artificial Intelligence (LOGIC-AI), and Sensors, Clouds, and Services Laboratory (SCSLab). I learned a lot from each of them and enjoyed my diverse conversations with them.

Thanks especially to the former and current outstanding young researchers of our school: Azadeh Ghari Neiat, M.Reza Hoseinyfarahabady, Rouzbeh Meymandpour, Niku Gorji, Natalie Tridgell, Tyler Crain, Gauthier Voron, Florin Dinu, Baptiste Lepers, and Martin Seybold who helped me to learn more about a variety of areas rang-

ing from incentive models and algorithms to compilers and efficient virtualization of NUMA architectures.

I would also like to thank Evelyn Riegler, Greg Ryan, William Calleja, Julia Ashworth, Irena Koprinska, Jinman Kim, Tom Cai, Basem Suleiman, and the whole members of our school for their academic and administrative support.

Thanks especially to all my wonderful friends in the School of Computer Science, School of Civil Engineering, and School of Electrical and Information Engineering as well as all my great friends at The Sydney Uni Sports & Aquatic Center, The Sydney University Gymnastics Club, and all others at Sydney Uni Sport & Fitness center.

Thanks to The Australian Government, Pawsey Supercomputing Center, The Faculty of Engineering, and the School of Computer Science who funded the majority of my research; both directly through a Research Scholarship and indirectly by providing me with computational support and a nice working environment.

Lack of good references can easily render the PhD into a Sisyphean task full of mental gymnastics. My research style, philosophy, and approach have been enormously influenced by reading comprehensive books, amazing surveys, and seminal papers written by a number of excellent researchers who helped me to gain a deeper understanding of my research problems by following them from their inception. Thanks to them. I will never be able to repay them but I hope to be able to follow in their footsteps.

I still cannot believe how fortunate I am to have found a friend like Christopher Natoli. I am deeply influenced by his kindness, selflessness, skillfulness, wisdom, and politeness. You are a lifesaver. I really enjoyed spending time in "Hawaii" with you. *Grazie mille, amico mio*!

Saving the best for last, I will always be grateful to my supportive, smart, hardworking, incredible, and loving family. *Balamisar*!

Masoud (Pouya) Salehpour

Sydney, May 04, 2022

Abstract

The label "Knowledge Graph" (KG) has been used in the literature for over four decades, typically to refer to a collection of information about real-world *entities* and their *inter-relationships*. The proliferation of KGs in recent times opens up exciting opportunities for a broad range of semantic applications such as *recommendations*. However, unlocking the full potential of KGs in response to the growing deployment requires data platforms to efficiently store and process the content to support various applications.

What began with extensions of relational database systems to store the content of KGs led to the design and development of a number of new specialized data management systems. Although progress has been made around building efficient KG data management systems, developing *high-performance* systems continues to pose research challenges. In this research, we studied the efficiency of existing systems for storing and processing KG content. Our results pointed to performance inconsistencies in representative systems across diverse query types. We address this by introducing a polyglot model of KG query processing to analyze each query and match it to the best-performing available systems. Experimental evaluation highlighted that our proposed approach provides consistently high performance.

Finally, we investigated leveraging emerging hardware and its benefits to RDF data management and performance. To this end, we introduced a novel index structure, RDFix, that utilizes Persistent Memory (PM) to outperform existing read-optimized indexes as shown experimentally.

Contents

List of Figures **xiv**

List of Tables **xviii**

1 Introduction **1**

1.1 Motivation 2
1.2 Foundations 3
1.2.1 Data Model 3
1.2.1.1 Resource Description Framework . . 3
1.2.1.2 Widespread Acceptance of RDF . . 5
1.2.2 Query Language 7
1.2.3 Example Application Use Cases 8
1.2.3.1 Semantic Search 8
1.2.3.2 Recommender Systems 9
1.2.3.3 Text Analytics 10
1.3 Challenges 11
1.3.1 Variety 12
1.3.2 Volume 14
1.4 Research Questions 16
1.5 Research Contributions 17
1.5.1 An Extensive Performance Study of RDF-stores 17
1.5.2 Introducing A Polyglot Model of Query Processing 18
1.5.3 Illustrating An Efficient Indexing Architecture with Persistent Memory 19
1.6 Research Scope 20

1.7 Thesis Structure 21
1.7.1 Chapter 2: Background and Review of the Related Literature 22
1.7.2 Chapter 3: Research Approach and Methods 22
1.7.3 Chapter 4: High-performance Knowledge Graph Query Processing: A Comparative Analysis . 23
1.7.4 Chapter 5: High-performance Knowledge Graph Query Processing: A Polyglot Model-based Approach 23
1.7.5 Chapter 6: High-performance Knowledge Graph Query Processing: A Persistent Memory-based Approach 24
1.7.6 Chapter 7: Conclusion 24

2 Background and Review of the Related Literature 25
2.1 Knowledge Graph 26
2.1.1 Definition 26
2.1.2 Historical Overview 29
2.1.3 Knowledge Graphs in Practice 30
2.1.3.1 Galleries, Libraries, Archives, and Museums 30
2.1.3.2 Scholarly Knowledge 31
2.1.3.3 Entity Identification and Science Stories 31
2.1.3.4 Biomedicine and Health 32
2.1.3.5 Checking Credibility 32
2.1.4 Key Requirements to Unlock the Full Potential of KGs 32
2.2 RDF Data Model 33
2.2.1 Foundational Elements 34
2.2.2 Design Principles 38
2.2.2.1 Triples 38
2.2.2.2 Basic Terms 39
2.2.3 RDF Structuredness 42
2.3 SPARQL Query Language 43

2.3.1 Syntax 43
2.3.2 Query Clause 46
2.3.3 Result Modifiers 48
2.4 Knowledge Graph Query Types 49
2.4.1 Single Triple Pattern 50
2.4.2 Star-shaped Queries (Subject-subject Joins) . 51
2.4.3 Chain-like Queries (Subject-object Joins) . . 51
2.4.4 Tree-like Queries 52
2.4.5 Optional Joins 52
2.5 RDF Data Management Systems 53
2.5.1 Overview 53
2.5.2 Classification of Centralized RDF-stores . . . 54
2.5.3 Major Non-native RDF-stores 57
2.5.3.1 Overview 57
2.5.3.2 Statement Table-based 59
2.5.3.3 Property Tables-based 60
2.5.3.4 Cluster-Property Tables 62
2.5.3.5 Summary 63
2.5.4 Major Native RDF-stores 65
2.5.4.1 Overview 65
2.5.4.2 Aggressive Indexing Strategy 65
2.5.4.3 Dictionary Encoding 66
2.5.4.4 In-memory Approaches 66
2.5.4.5 Disk-based Approaches 70
2.5.4.6 Summary 71
2.6 SPARQL Query Optimization 74
2.6.1 Overview 74
2.6.2 Join Query Graph Construction 74
2.6.3 Join Ordering and Plan Enumerations 76
2.6.4 Cardinality Estimation 78
2.7 Virtual Integration of RDF-stores 80
2.7.1 Overview 80
2.7.2 Mediator-based Systems 80
2.7.3 Federated Systems 81

2.8 Polygloty: A Multi-database Solution for Query Processing . . . 82
2.9 Benchmarking Efforts . . . 83
2.10 Conclusion . . . 84

3 Research Approach and Methods 85
3.1 Introduction . . . 85
3.2 Definition of Diversity-tolerance . . . 86
3.2.1 Remarks on the Impacts of Structuredness (S) on Performance (P) . . . 87
3.2.2 Remarks on the Impacts of Query features (Q) on Performance (P) . . . 88
3.2.3 Remarks on the of Impacts of Volume on Performance . . . 89
3.3 Evaluation Approach: Controlled Experiments . . . 89
3.3.1 Overview . . . 89
3.3.2 Basis of Evaluation . . . 90
3.3.3 Robustness of Measurements . . . 90
3.4 Conclusion . . . 90

4 High-performance Knowledge Graph Query Processing: A Comparative Analysis 92
4.1 Introduction . . . 92
4.2 Remarks on Factors Influencing Performance . . . 93
4.3 Experimental Context . . . 95
4.3.1 Benchmark Datasets . . . 95
4.3.2 Knowledge Graph Queries . . . 96
4.3.3 Computational Environment . . . 99
4.3.4 Experimental Platforms . . . 99
4.3.4.1 Configurations . . . 100
4.3.4.2 Indexes . . . 100
4.3.4.3 Remarks on Bulk Loading and Measurement . . . 101
4.4 Exploratory Analysis . . . 102
4.4.1 Results . . . 103

4.4.2 Analysis 105
4.4.2.1 Mapping Query Types and DMS types 105
4.4.2.2 Optional Joins and DMSs 107
4.4.2.3 The effects of scale 107
4.5 Lessons Learned 108
4.5.1 Limitations 109
4.6 Conclusion 110

5 High-performance Knowledge Graph Query Processing: A Polyglot Model-based Approach 111
5.1 Introduction 112
5.2 SymphonyDB: A Polyglot Model for Query Processing 113
5.2.1 Polyglot Database Management System Layer: Storing KGs 114
5.2.2 Polyglot Access Management: Query Labeling and Execution 116
5.2.3 Polyglot Access Management: Query Translation 118
5.3 Experimental Context and Platform 121
5.3.1 Evaluation Datasets and Queries 121
5.3.2 Evaluation Platform 122
5.4 Results 124
5.4.1 Discussion 126
5.4.1.1 Analysis: Performance Consistency . 126
5.4.1.2 Insight: SymphonyDB as Common Ground 126
5.4.1.3 Limitations 128
5.5 Conclusion 129

6 High-performance Knowledge Graph Query Processing: A Persistent Memory-based Approach 130
6.1 Introduction 131
6.2 RDFix: Design and Architecture 133
6.2.1 Operations 136
6.2.1.1 Insert 136
6.2.1.2 Lookup 138

6.2.1.3 Range Scan . . . 139
6.2.1.4 Delete . . . 139
6.2.1.5 Updates . . . 140
6.2.2 Rationale . . . 141
6.3 Experiments . . . 142
6.3.1 Setup . . . 142
6.3.2 Measurement and Results . . . 145
6.3.2.1 Insert . . . 146
6.3.2.2 Lookup . . . 148
6.3.2.3 Range Scan . . . 152
6.3.2.4 Delete . . . 153
6.3.3 Space-Time Tradeoffs . . . 154
6.4 RDFix Integration into SymphonyDB . . . 156
6.5 Conclusion . . . 157

7 Conclusion **159**
7.1 Introduction . . . 160
7.2 Summary of Contributions and Results . . . 161
7.2.1 Definition of Diversity-tolerant RDF Data Management . . . 161
7.2.2 Returning to Research Question 1 . . . 161
7.2.2.1 Factors Influencing Query Performance 161
7.2.2.2 Exploring Efficiency of Major RDF-stores . . . 162
7.2.2.3 Key Findings . . . 162
7.2.3 Returning to Research Question 2 . . . 163
7.2.3.1 Key Findings . . . 163
7.2.4 Returning to Research Question 3 . . . 163
7.2.4.1 Key Findings . . . 164
7.3 Limitations . . . 164
7.3.1 Redundant Information . . . 164
7.3.2 Efficient Query Translation . . . 165
7.3.3 Maintenance and Integration . . . 165
7.3.4 License . . . 165
7.3.5 Parallel Operations . . . 166

7.4 Future Work 166
7.4.1 Update Queries 166
7.4.2 Programming Languages Effects on Performance 166
7.4.3 Knowledge Graph Query Typology 167
7.4.4 Energy Consumption 167
7.4.5 Cloud-based Non-monolithic RDF-stores . . . 167
7.5 Concluding Remarks 168

Bibliography **169**

A Short List of Acronyms **192**

List of Figures

1.1 An informal example of a simple KG describing the Opera House, a heritage site located in Sydney. . . . 5
1.2 The Linked Open Data (LOD) diagram (source: https://lod-cloud.net/#diagram) 6
1.3 An informal example of a SPARQL Query 8
1.4 RDF triples (informally expressed in pseudocode) representing a sample recipe description 9
1.5 An illustration of a KG-based recommendation based on [32] 10
1.6 An example of a KG with *high-structuredness* (based on an example from [50]) 12
1.7 *High-structuredness* of the KG in Fig. 1.6 with respect to `type1` (based on an example from [50]) 14
1.8 An example of a KG with *low-structuredness* (based on an example from [50]). 15
1.9 *Low-structuredness* of the KG in Fig. 1.8 with respect to `type2` (based on an example from [50]) 16
1.10 Logical organization of the thesis 21

2.1 Una Osili's information in the University of Wisconsin–Madison KG (source: [77]) 30
2.2 Absolute IRIs are used to represent an example triple. 35
2.3 Using prefixed names to represent the same triple in Fig. 2.2. 35
2.4 A subset of RDF triples from DBpedia describing Elon Musk. 36
2.5 The RDF schema information corresponding to the RDF triples in Fig. 2.4. 37

2.6 An example KG informally modeled in RDF to represent companies offering flights between two cities (based on an example from [16]) 41
2.7 The KG in Fig. 2.6 is modeled in a property graph (based on an example from [16]) 41
2.8 An example SPARQL query 45
2.9 The logical database design of 3store [38] 59
2.10 An example illustration of vertical partitioning [90] . 62
2.11 An example illustration of clustered-property tables [91] 63
2.12 Indexing RDF triples in all six possible orders 65
2.13 An example illustration of SPO index in Hexastore [60] 67
2.14 An example illustration of the Kowari system's AVLtree-based indexing [117] 69
2.15 Evolution of native RDF-stores [86]. Edges indicate influences. 72
2.16 Query graphs of an example SPARQL query (based on [81, 92]) 75
2.17 An example SPARQL query (based on [81, 92]) . . . 78
2.18 Optimal and suboptimal query plans for the star-shaped query in Fig. 2.17 (based on [81, 92]) 78

4.1 Impacts of subject-subject join queries on the DMSs (cold- and warm-run). X axes show DMSs and Y axes show the execution time of each query in milliseconds (using log scale). 102
4.2 Impacts of subject-object join queries on the DMSs (cold- and warm-run). 104
4.3 Impacts of tree-like join queries on the DMSs (cold- and warm-run). 105
4.4 Impacts of optional join queries on the DMSs (cold- and warm-run). 106

5.1 A schematic view of SymphonyDB's architecture . . 113
5.2 (a) An example of a subject-subject query pattern, (b) an example of a subject-object query pattern, and (c) an example of a tree-like query pattern 115

5.3 A simplified example in which an incoming SPARQL query and its corresponding abstract syntax tree is depicted. 116
5.4 The query translation logic flow 118
5.5 Execution time of each query in milliseconds (**log scale**). 125
5.6 An example SPARQL Query whose predicate is replaced by a variable. 127

6.1 Overview of RDFix's structure for *spo* indexing order. Layouts of the arrays and each list node are also shown in the red boxes. 134
6.2 Space consumption of RDFix for *spo* indexing order. 135
6.3 Process of *insert* operation. 136
6.4 Process of *lookup* operation for a given single key <v1>.138
6.5 Process of *delete* operation. 140
6.6 Execution times (log scale) of *insert* (aka, *put*) operations for the different datasets and scales. 145
6.7 Execution times (log scale) of 1M, 10M, and 100M individual *lookup* (aka, *get*) operations for randomly generated <v1,v2> pairs at scale. 149
6.8 Execution times (log scale) of 1M, 10M, and 100M individual *lookup* (aka, *get*) operations for randomly generated <v1> at scale. 151
6.9 Execution times (log scale) of 100M individual *range scan* operations for randomly generated range [k1, k1+1000] at scale. 153
6.10 Execution times (log scale) of 10K individual *delete* operations for randomly generated <v1> at scale. . . 154
6.11 Space-time tradeoff curve to perform 100M individual lookup operations over the different datasets with 10M triples for randomly generated <v1>. 155
6.12 A schematic view of integrating RDFix into SymphonyDB's architecture as a stand-alone indexing structure156

6.13 A schematic view of integrating RDFix into SymphonyDB's architecture as an embedded indexing structure 157
6.14 A schematic view of integrating RDFix into SymphonyDB's architecture as both a stand-alone indexing structure and an embedded indexing structure 158

List of Tables

2.1 Selected definitions of KG (source: [74]) 28
2.2 Various classifications of RDF-stores 56
2.3 Centralized non-native RDF-stores 64
2.4 Centralized native RDF-stores 73

4.1 Statistics of the Benchmark datasets 98

5.1 SPARQL expressions representation and their equivalent MQL expressions 119
5.2 Sample rules used to translate SPARQL queries to MQL 120
5.3 Characteristics of the KGs that were used to run the experiments 121
5.4 Types of the queries. SS^{a*}: Subject-subject join, SO^{b*}: Subject-object join, Co^{c*}: combination of SS and SO, OPT^{d*}: Optional pattern, Fil^{e*}: Filter, ORD^{f*}: Order by, Lim^{g*}: Limit, OFF^{h*}: Offset, STP^{i*}: Single triple pattern (no join) 123

6.1 Description of the evaluation platform. 143
6.2 Statistics of the benchmark datasets. Sub stands for subjects, Pre: predicates, Obj: objects, Tri: triples, and Stru: structuredness. 144

Chapter 1

Introduction

Strengthening computer systems with comprehensive knowledge of the real-world has been a long-standing research goal in computer science [7, 8, 9]. Attaining this elusive goal is becoming more plausible today in light of the tremendous progress in a wide spectrum of research fields ranging from *knowledge harvesting* and *artificial intelligence* to *databases* and *natural language processing* as well as the *Semantic Web* and *information systems*. Advances in these fields have enabled us to extract information from knowledge sources to create large-scale *knowledge bases*, also known as Knowledge Graphs (KGs), i.e., collections of information about real-world entities (such as people, places, organizations, movies, books, music albums, proteins, genes, drugs, etc.) and their interconnections [7]. For instance, many large-scale KGs have been deployed both by private enterprises and in the public domain over the last ten years such as *Google Knowledge Graph* [10], *Microsoft Satori* [11], *IBM Watson* [12], and *Amazon Product Graph* [13, 14] as well-known examples of private KGs alongside *Wikidata*[1], *BabelNet*[2], *DBpedia*[3], and *YAGO*[4] as salient examples of KGs with publicly accessible content containing billions of entities and millions of their interconnections. The success of KG-based applications relies on managing, accessing,

[1]Available Online: http://wikidata.org
[2]Available Online: http://babelnet.org
[3]Available Online: http://dbpedia.org
[4]Available Online: http://yago-knowledge.org

and analyzing the content, requiring many crucial properties like efficient query processing and fast data access times. This chapter focuses on the significance of these properties for KG-based applications and presents the research questions that we address in this thesis related to the development of high-performance KG query processing systems.

1.1 Motivation

Processing queries over KGs enables or enhances a wide variety of applications, semantic or otherwise, such as similarity analysis, recommendation provision, and reasoning. As a response to the growing number of deployments, unlocking the full potential of KGs requires *data models* to represent KG content effectively and *data platforms* to efficiently store the content of KGs. However, querying large-scale KGs efficiently is still a challenge.

Throughout this thesis, we discuss this research challenge in detail. We present an extensive study of the efficacy and efficiency of the services offered by major existing Data Management Systems (DMSs) to query KGs. Through our experiments, we illustrate the performance limitations that exist in the current generation of these systems. In response to these limitations, we propose high-performance solutions to execute a large number of diverse queries over KGs with a large volume of diverse content. We evaluate the effectiveness of our solution through experimental analysis by using standard benchmark datasets and queries.

In this chapter, we confine our focus to highlighting the foundational requirements to unlock the full potential of KGs. It begins with the introduction of KG data models, query languages, and the performance challenges we face to efficiently process KG queries. We then present the research questions that we address in this thesis in relation to high-performance query processing over KGs. Finally, the structure and logical organization of this research are presented (see Section 1.7).

1.2 Foundations

1.2.1 Data Model

The Resource Description Framework (RDF)[5] has widely been accepted for representing the content of KGs. It is a standardized graph-like data model whereby the graph nodes represent entities of a domain of interest, and the edges represent interconnections between them [15, 16]. For example, if we use RDF to represent the content of a KG about books, nodes can represent authors and books, and a directed edge from a node α to another node β can be properly labeled to represent that α is the author of β. Well-known general-purpose KGs like YAGO and DBpedia typically use this data model to represent their content [7, 17].

Other graph-structured data models, such as *property graphs*, have also been proposed to represent KG content whereby additional meta-information can be assigned to the graph edges or nodes in the form of a set of property-value pairs, known as *attributes* [15]. The use of property graphs can provide additional flexibility when compared to the RDF data model. However, the RDF model provides a more minimal representation [15]. The use of tree-like data models, such as Extensible Markup Language (XML), is also feasible in most cases, yet the RDF model is generally more flexible without the need to organize the data *hierarchically* (e.g. parents, children, siblings, etc.) [16]. With these properties, it is no surprise that the RDF model and data are widely used in studies and real-world applications of KGs. We, therefore, focus our research on the RDF data model, providing extensive and in-depth analyses of the performance of queries on KGs represented using the RDF data model.

1.2.1.1 Resource Description Framework

Each RDF dataset can be viewed as a collection of *statements* about entities and their interrelations, called triples, of the form $<s,p,o>$ where s is a subject, p is a predicate, and o is an object [16, 17,

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

18, 19, 20, 21, 22]. Entities are depicted by subjects and objects, whereas the relationships between them are represented as predicates. RDF triples can be used to represent almost anything that needs to be modeled, such as people, organizations, places, real-world objects, or even abstract concepts (see Section 2.2 for more details). For example, Fig. 1.1 depicts a KG to represent the fact that the Opera House is a heritage site located in Sydney. Different entities are nodes in the graph and relationships between them are represented as labeled edges. The content can be also represented by the following triples:

```
1     <OperaHouse>  <located_in>   <Sydney>.
2     <OperaHouse>  <instance_of>  <landmark>.
3     <OperaHouse>  <instance_of>  <heritage site>.
4     <OperaHouse>  <instance_of>  <tourist attraction>.
5     <OperaHouse>  <style>        <expressionist>.
6     <OperaHouse>  <opening_date> <20 Oct. 1973>.
7     <Sydney>      <located_in>   <Australia>.
8     <Sydney>      <instance_of>  <city>.
9     <Sydney>      <instance_of>  <capital>.
10    <Sydney>      <instance_of>  <metropolis>.
```

For instance, one way to represent the statement "Opera House is located in Sydney" in RDF is as the triple: <OperaHouse> <located_in> <Sydney>. A single RDF triple is the atomic unit of information in the RDF data model [23]. Note that all the above triples are informally expressed in pseudocode (similar to the RDF 1.1 Primer [24]). However, Internationalized Resource Identifier (IRIs) [25], which look like URLs and often include unique sequences of characters and numbers, are generally used to represent triples. More specifically, in a triple, the subject is an IRI or a blank node (i.e., an entity without a global identifier [24]). The predicate is an IRI and the object is an IRI, a literal (i.e., basic values that are not IRIs [24]), or a blank node. More details about the RDF data model are presented in Section 2.2 of Chapter 2.

Figure 1.1: An informal example of a simple KG describing the Opera House, a heritage site located in Sydney.

1.2.1.2 Widespread Acceptance of RDF

RDF is used in a wide range of domains [16]. The evidence of this can be clearly seen in the Linked Open Data (LOD) project where a large set of RDF collections (such as DBpedia, etc.) is interlinked at the entity level, constituting the Web of LOD. All the KGs of this project can be freely used and distributed with the vision of enabling the Internet to become a "global database". Under the LOD project, a cloud[6] of over 3,000 interlinked datasets has already been made available [26]. Fig. 1.2 attempts to illustrate the interlinked RDF datasets that have been published in the Linked Data format under the LOD project (source: `https://lod-cloud.net/#diagram`).

The wide acceptance of the RDF data model is not limited to the LOD project or data that can be freely used and distributed on the Web. In recent years, a number of government entities, most notably from the US[7] and UK[8], have adopted this data model for a number of datasets, including that of the Coronavirus/COVID-19[9].

[6] Available Online: `https://lod-cloud.net/`
[7] Available Online: `https://www.data.gov/`
[8] Available Online: `https://data.gov.uk/`
[9] Available Online: `https://www.coronavirus.gov/` (April 2021)

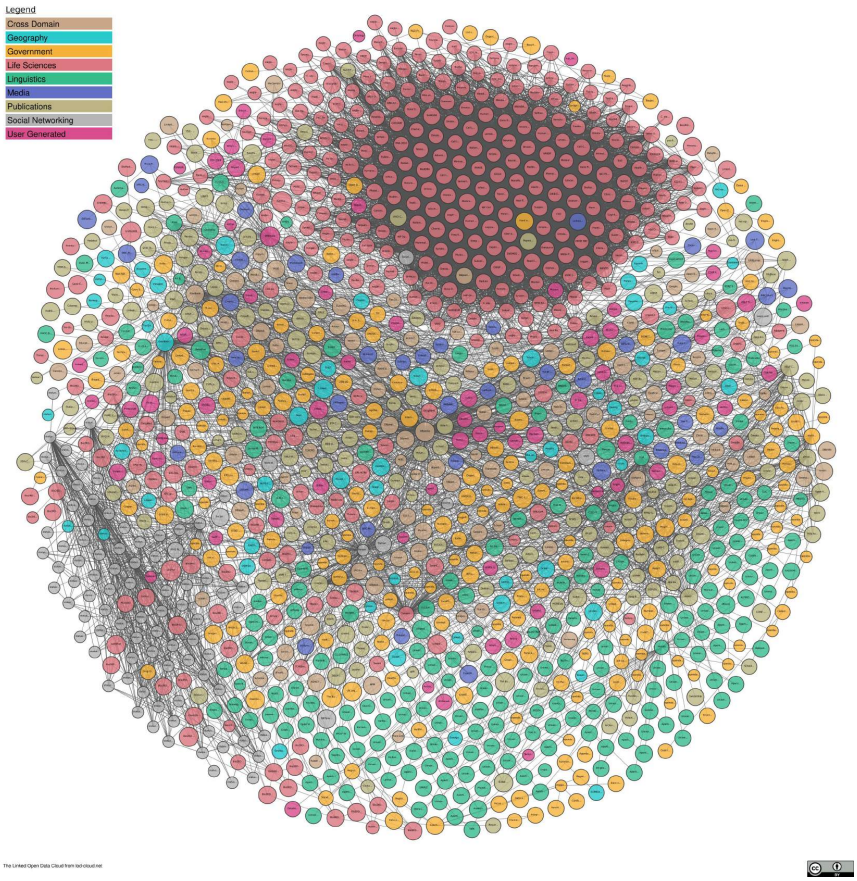


Figure 1.2: The Linked Open Data (LOD) diagram (source: https://lod-cloud.net/#diagram)

For example, the time-series dataset includes results from the viral COVID-19 Polymerase Chain Reaction (PCR) laboratory tests, showing results in an RDF representation from over 1,000 U.S. laboratories and testing locations, including commercial and reference laboratories, public health laboratories, hospital laboratories, and other testing locations[10]. Many large organizations and companies have also been using the RDF data model in recent years for busi-

[10] Available Online: https://healthdata.gov/dataset/COVID-19-Diagnostic-Laboratory-Testing-PCR-Testing/j8mb-icvb (July 2021)

ness data representation and data integration like Pfizer[11]. These, and other examples, can easily highlight the wide acceptance of the RDF data model in practice. In the next section, we discuss major languages for querying KGs.

1.2.2 Query Language

As discussed, RDF has widely been accepted as a standard for representing the content of KGs. While the representation of KG content is necessary, it is not sufficient as users require methods to query and utilize the data. To meet this need, a number of practical query languages have been proposed over the last few years, including SPARQL Protocol and RDF Query Language (SPARQL) as a query language specifically designed for the RDF data model as well as Cypher [27] and some other languages like Gremlin [28] and G-CORE [29] for querying property graphs. These query languages offer a set of essential *primitives* like *basic graph patterns*, *relational operators*, and *path expressions* for querying graphs. Details of the popularity of these query languages are discussed in [30]. Among these, SPARQL has become the common language of choice for querying KGs. Evidence to support this claim is shown by most, if not all, publicly accessible KG content (such as *WikiData*, *DBPedia*, etc.) are modeled in RDF while offering access to their content through SPARQL.

SPARQL is a *standard* that is acknowledged and recommended by the W3C. Its syntax holds many similarities to the widely known SQL. SPARQL queries are typically formulated based on *triple patterns*. Triple patterns are similar to RDF triples except that each subject, predicate, and object may be a variable [31]. An *informal* example of a SPARQL query is presented in Fig. 1.3. It asks for the subject "OperaHouse's" architectural style name from the graph in Fig. 1.1. In this query, *"`OperaHouse style ?styleName`"* is a triple

[11] Details Available Online: https://franz.com/agraph/cresources/white_papers/BioInform_mar3_Pfizer-Partners-with-IO-Franz_on-Semantic-Proof-of-Concept.pdf

```
1 SELECT ?styleName
2 WHERE {
3 OperaHouse style ?styleName .
4 }
```

Figure 1.3: An informal example of a SPARQL Query

pattern whose object is a variable. More details about the SPARQL query language are presented in Section 2.3 of Chapter 2.

Processing queries over KGs enables or enhances a wide variety of applications. For instance, KG query processing has found wide use for a variety of tasks in semantic search, recommendation systems, natural language processing, and text analytics. Some of these application use cases are discussed in the next section.

1.2.3 Example Application Use Cases

1.2.3.1 Semantic Search

A prominent use case where KGs have become a key asset is semantic search over the Web. Most, if not all, well-known search engines like Bing and Google typically execute queries over their KGs (as their important background resources) to return a set of accurate and concise results (often entities) rather than solely listing some hyperlinks to web pages whenever search queries precisely center around an entity such as books, singers, music albums, movies, theatres, hotels, restaurants, tourist attractions, flight tickets, retailers, products, sports events, etc. [7]. A knowledge panel (sometimes referred to as an information box) also appears on the result page when we search for entities to provide us with a quick snapshot of information.[12]

These and other major search engines make extensive use of KGs that are typically modeled in RDF to understand the content of web

[12] Available Online: https://support.google.com/knowledgepanel/answer/9163198?hl=en

```
1      <Page1>   <type>          <Recipe>.
2      <Page1>   <author>        <Author1>.
3      <Page1>   <datePublished> <2018-03-10>.
4      <Page1>   <description>   <This cake is awesome.>.
5      <Page1>   <name>          <Party Coffee Cake>.
6      <Page1>   <prepTime>      <PT20M>.
7      <Author1> <type>          <Person>.
8      <Author1> <name>          <Mary Stone>.
```

Figure 1.4: RDF triples (informally expressed in pseudocode) representing a sample recipe description

pages. As an example, a web developer can use RDF to provide explicit clues about the meaning of a recipe page to Google such as the title of the recipe, the author of the recipe, what are the ingredients, the cooking time and temperature, the calories, and other details.[13] A sample description can be represented by the RDF triples (informally expressed in pseudocode) as shown in Fig. 1.4.
Using the RDF data model to explicitly describe each individual element of the recipe enables the semantic search engine to understand the content of the page and formulate relevant RDF triples to add to its KGs. On this basis, users can search for the recipe by other attributes, such as ingredient, calorie count, cook time, and other details with the expectation of receiving accurate search results in most cases rather than merely hyperlinks to web pages. Indeed, it is a clear incentive for KG growth when big companies like Google are promoting the use of KGs and the RDF data model as part of their search engine.

1.2.3.2 Recommender Systems

Recommender systems can help users to select what interests them amongst a large number of choices [33]. Over the last few years, utilizing KGs to develop recommender systems has become more prevalent to improve the user experience in a broad range of scenarios, including music recommendations, movie recommendations,

[13] Details Available Online: https://developers.google.com/search/docs/guides/intro-structured-data

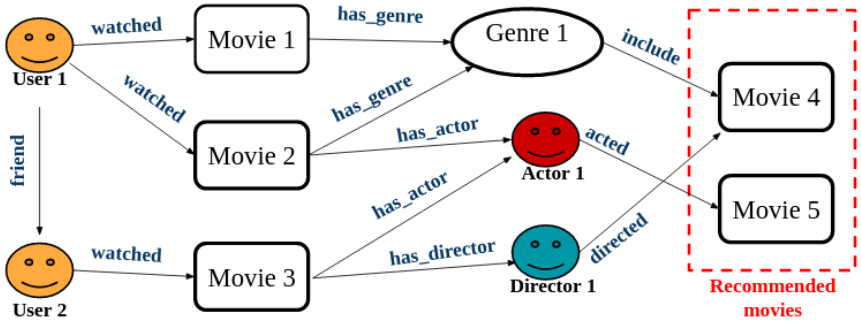


Figure 1.5: An illustration of a KG-based recommendation based on [32]

and online shopping. In these and other scenarios, KGs can provide better understanding of the mutual relations between entities of a domain of interest that is a key for developing recommender systems [32]. For instance, Fig. 1.5 depicts a KG-based movie recommendation scenario represented in [32]. This KG contains five entities, namely, users, movies, actors, directors, and genres as well as five interrelations between them, namely, interaction, belonging, acting, directing, and friendship. Based on this, "Movie4" can be recommended to User1 since it has the same genre as "Movie1" which was watched by User1 before.

1.2.3.3 Text Analytics

As discussed in [7], understanding entities and their interconnections in text play a major role in large-scale analytics over social media posts, news articles, political discussions, scientific papers, commercial advertisements, etc. A trending use case is to create KGs to represent mentions of product names along with all the consumer opinions that are associated with the products for performing in-depth and comparative studies like *filtering* and *grouping* of products based on their categories and the consumers' locations and opinions [7]. Another growing use case of KGs in semantic text analytics is to detect gender bias in news and similar online content as discussed in [34]. The core idea is to create a KG of people from text and execute queries over this KG to find out each person's gender.

Based on this, it is also feasible to compute gender-specific statistics like the number of males versus females in political offices and gender-associated earning differentials in an industry of interest (e.g., the movie industry).

1.3 Challenges

Developing a standard data model (RDF) to represent KGs as well as a standard language (SPARQL) to query their content were major steps in opening up exciting opportunities for a wide range of applications. However, these are not still sufficient since developing efficient data management systems customized for KGs (aka, RDF-stores) is also required to implement these standards to unlock the full potential of KGs. This is particularly critical as the size of KGs continues to explode.

In response, a number of RDF-stores have been developed in the past few years. It began with early works on FORTH RDF Suite [35, 36], Redland [37], 3store [38], Sesame [39], KAON [40], Jena [41], RStar [42], and DLDB [43], and continued through a number of data platforms that have relatively been introduced more recently such as RDF-3X [44]. Although there has been some progress in research efforts dealing with adopting a broad range of design choices and architectures to build RDF-stores, a number of challenges still persist when designing *high-performance* systems [45, 46, 47, 48]. There are two critical challenges that hinder the performance of KG implementations:

i) **Variety.** RDF can be used to represent diverse content ranging from *structured* to *unstructured* data. Thanks to this flexibility, users can collect data without a *schema-first* database design phase — a paradigm known as "*pay-as-you-go*" dataspaces [47, 49] and is sometimes referred to as the *schema-relaxable*, *schema-free*, or *schema-last* feature of the RDF data model [44]. However, this *flexibility* as well as the absence of

an *explicit schema* and the *heterogeneity* of RDF datasets pose challenges for RDF-stores [44, 45, 50].

ii) **Volume.** The proliferation of RDF data is rapidly accelerating. This is perhaps nowhere more evident than in the LOD project in which an RDF data cloud of over 3,000 interlinked datasets has already been available [26]. These datasets include more than 84 billion triples in total [26]. While the increased volume of RDF datasets is exciting for most semantic applications, it poses significant scalability demands on existing RDF data management systems [47, 51].

More information about these challenges (variety and volume) is presented in the next sections.

```
1    <person1>   <name>          <Natalie>.
2    <person1>   <home-phone>    <2468>.
3    <person1>   <cell-phone>    <1010>.
4    <person2>   <name>          <Zoe>.
5    <person2>   <home-phone>    <8642>.
6    <person2>   <home-phone>    <9753>.
7    <person2>   <cell-phone>    <2020>.
8    <person3>   <name>          <Dana>.
9    <person3>   <cell-phone>    <3030>.
```

Figure 1.6: An example of a KG with *high-structuredness* (based on an example from [50])

.

1.3.1 Variety

RDF can represent a variety of KG content across the full spectrum of structuredness (i.e., from *structured* to *unstructured* data) [50]. More specifically, each RDF dataset typically comes with a number of types as well as a number of instances of each type. The level of structuredness of an RDF dataset is typically determined by the sparsity (presence) of its predicates across instances of each type. Consider for example the KG represented in RDF triples (note

that similar to [24], we informally expressed triples in pseudocode) in Fig. 1.6. For simplicity, assume that all the instances (i.e., `person1` to `person3`) belong to the same type (i.e., `type1`). We also assume that this type (i.e., `type1`) has predicates `name`, `home-phone`, and `cell-phone`. If each instance (i.e., `person1` to `person3`) in the KG sets values for most (if not all) of the predicates of `type1`, then all the instances in the KG have a fairly similar structure that conforms to `type1`. In this case, we can say that the KG (represented using RDF triples) has high structuredness with respect to `type1`. Fig. 1.7 attempts to show the high-structuredness of the KG with respect to `type1`. The X-axis shows predicates of `type1` and the Y-axis shows the number of instances that set the value for each predicate. As can be seen, all the instances set values for almost all predicates. The only exception is `person3` whose `home-phone` is not set. Since all the instances in the KG have a fairly similar structure that conforms to `type1`, we can say that this KG has high structuredness. Following [50], we are not interested in the number of times a predicate is set for an instance (i.e, multi-valued predicates). Intuitively, this is because the level of structuredness of a KG dataset is affected by the sparsity (presence) of its predicates across instances, rather than the number of occurrences of the predicates [50].

Consider now the KG represented in RDF triples in Fig. 1.8. For illustration purposes, consider that all the instances (i.e., `person1` to `person6`) belong to the same type (i.e., `type2`). We also assume that this type (i.e., `type2`) has 5 predicates: `name`, `home-phone`, `cell-phone`, `bornOn`, and `postcode`. The KG represented in RDF triples in Fig. 1.8 has low structuredness with respect to `type2`. To see why this is the case, notice that all instances (i.e., `person1` to `person6`) in the KG do not set values for most of the predicates of `type2`. In other words, all the instances in the KG do not have a fairly similar structure that conforms to `type2`. The sparsity (presence) of its predicates across instances are shown in Fig. 1.9

Representing data across the full spectrum of structuredness can be directly viewed as a strength of RDF. However, this inherent ease of use and flexibility as well as the absence of an explicit schema

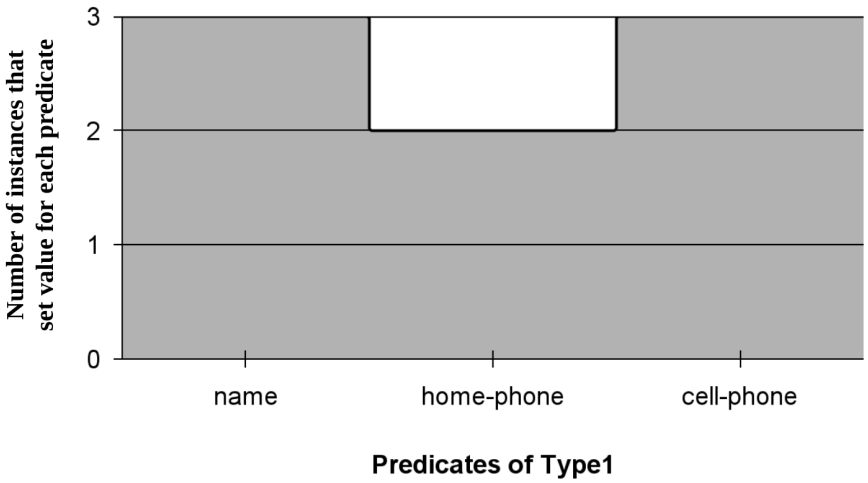


Figure 1.7: *High-structuredness* of the KG in Fig. 1.6 with respect to `type1` (based on an example from [50])

and heterogeneity of RDF datasets pose performance challenges to RDF-stores for *storing* and *querying* efficiently. This is due to RDF-stores typically being unable to make any priori assumptions about the level of structuredness of the datasets [50].

1.3.2 Volume

In general, the RDF data volume is measured by the number of triples in the data and is typically referred to as the data size. We hinted at increasing volumes in the previous sections of this chapter by referring to the use of the RDF data model to represent the massive content of KGs such as DBpedia[14] and YAGO[15] as well as the tendency of different scientific communities to represent their experiments and results using RDF (e.g., Bio2RDF[16] and Uniprot[17] as two major RDF collections representing scientific information like protein sequence). With this kind of proliferation, KGs can grow into very large datasets that contain hundreds of millions of RDF triples. While the increased volume of RDF datasets is appealing

[14] Available Online: https://www.dbpedia.org/
[15] Available Online: https://yago-knowledge.org/
[16] Details Available Online: https://bio2rdf.org/
[17] Details Available Online: https://www.uniprot.org/

```
1     <person1>   <name>          <Natalie>.
2     <person1>   <home-phone>    <2468>.
3     <person1>   <cell-phone>    <1010>.
4     <person2>   <name>          <Zoe>.
5     <person2>   <home-phone>    <8642>.
6     <person2>   <home-phone>    <9753>.
7     <person2>   <cell-phone>    <2020>.
8     <person3>   <name>          <Dana>.
9     <person3>   <cell-phone>    <3030>.
10    <person4>   <name>          <Charlie>.
11    <person4>   <bornOn>        <12 Aug 1999>.
12    <person4>   <postcode>      <951>.
13    <person5>   <name>          <Terry>.
14    <person5>   <postcode>      <159>.
15    <person6>   <name>          <Manny>.
16    <person6>   <postcode>      <456>.
```

Figure 1.8: An example of a KG with *low-structuredness* (based on an example from [50]).

for a wide range of semantic applications, it also poses expanding scalability demands on RDF data management systems [47, 51].

A larger part of the data typically needs to be scanned to return the query result as the data volume grows. Scanning more data typically infers more intermediate results that require management, leading to increased pressure on the computer hardware and software resources such as the relevant parts of the operating system, storage media, and other components. This inevitably requires the design and implementation of query processing algorithms with higher levels of complexity to achieve scalable performance. In other words, the data volume is a critical factor that can easily influence the essential aspects of the RDF query processing (e.g., see Section 2.6), and consequently the performance of RDF-stores [52]. More details about the impact of volume on performance can be found in Section 3.2.3 of Chapter 3.

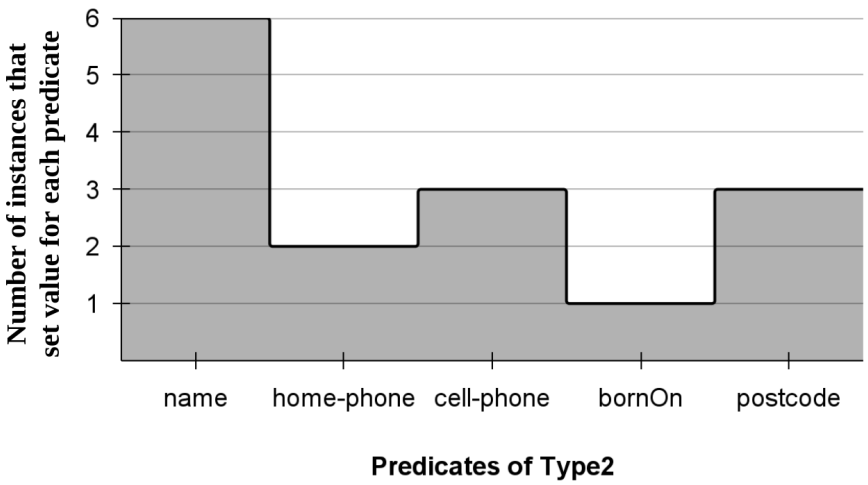


Figure 1.9: *Low-structuredness* of the KG in Fig. 1.8 with respect to `type2` (based on an example from [50])

1.4 Research Questions

The widespread adoption of RDF for the representation and exchange of data has highlighted the need for employing efficient RDF-stores. To address this need, a number of RDF-stores utilizing a broad range of techniques have been developed over the past few years. However, the absence of an explicit schema, and the increasing size of RDF datasets, still challenge RDF-stores designers [50, 53]. To tackle these challenges, the three broad, yet essential problems that are investigated in this research can be summarized as follows:

- ***Question 1***. *How efficiently do major existing RDF-stores perform to store diverse RDF datasets and execute archetypal query types over them*?

- ***Question 2***. *How to design and develop an RDF data management system that can reduce the negative effects of the variety of RDF data and diversity of queries on the performance*?

- ***Question 3***. *How can we leverage emerging hardware devices like Persistent Memories (PMs) to design high-performance RDF-stores*?

1.5 Research Contributions

1.5.1 An Extensive Performance Study of RDF-stores

As stated previously, KGs are gaining widespread momentum for use in different domains. Subsequently, storing their content and efficiently executing queries over them is becoming increasingly important. A range of Data Management Systems (DMSs) have been employed to process KGs. One of our contributions is to provide an in-depth analysis of query performance across diverse DMSs and KG query types. More specifically, we provide a fine-grained, comparative analysis of four major DMS types, namely, row-, column-, graph-, and document-stores, against major query types, namely, subject-subject, subject-object, tree-like, and optional joins (more details about query types can be found in Section 2.4). In particular, we analyzed the performance of row-store Virtuoso, column-store Virtuoso, Blazegraph (i.e., graph-store), and MongoDB (i.e., document-store) using five well-known benchmarks, namely, BSBM [54], WatDiv [45], FishMark [55], BowlognaBench [56], and BioBench-Allie [57]. A summary of our findings is as follows:

- No single RDF-store displays superior query performance across the four query types, however, there are significant interaction effects between different types of RDF-stores and query types.
- Taking advantage of data locality and efficient implementation of indexing data structures (such as B-trees) can contribute to better performance for executing subject-subject join queries.
- Suitable *cardinality estimation*, as well as efficient query optimization, offer a significant performance improvement on subject-object join queries.

- The simplicity of the underlying storage layout, increasing data locality, and suitable caching techniques can lead to performance advantages for tree-like join queries.

1.5.2 Introducing A Polyglot Model of Query Processing

As already discussed (see Section 1.3), *variety* and *size* of KG content pose challenges to DMSs to *store* and *query* KGs. As we have shown in Chapter 4, a single, *one-size-fits-all* DMS does not currently exist for efficient KG query processing [45]. A contribution of this thesis is to develop a solution to this problem that is inspired by Ashby's First Law of Cybernetics [58] and Stonebraker et. al. [59] which can be paraphrased in this context to state that the variety in the solution architecture should be greater than or at least equal to that of the variety displayed by the data and the queries. Based on this, our proposed solution is an architecture based on *polyglot* model of query processing and access languages supported by a design that can analyze individual queries and match each to the likely best-performing database engine. More specifically, we present *SymphonyDB*, a prototype that provides polyglot support for RDF query processing. It is a multi-database approach supported by an access management layer to provide a unified query interface for accessing the underlying DMSs. This layer receives the incoming workloads in the form of SPARQL queries and routes each of them to one (or more than one) of the more likely to be efficiently matched DMSs among Virtuoso, Blazegraph, RDF-3X, and MongoDB as representative DMSs that are included in our prototype at this time. The results of our experiments with the prototype over well-known KG benchmark datasets point to the efficiency and consistency of its performance across different query types and datasets.

1.5.3 Illustrating An Efficient Indexing Architecture with Persistent Memory

The current generation of RDF-stores is typically designed based on an aggressive indexing scheme specific to the *schema-relaxable* feature of the RDF data model based on which indexes are built over different permutations of the three dimensions that constitute an RDF triple [44, 60]. In this design, indexing architectures and their primary storage locations are critical to achieving good performance. In other words, the speed with which these indexes can be read from the storage devices plays a major role in the performance of RDF-stores to process the content of KGs. For the primary storage locations, disks (e.g., HDD or SSD) and main memory (DRAM) are the typical options. Disk-based devices only support bulk data transfers as blocks which tend to be slow [61, 62]. This means that even for reading a single byte of data stored on an SSD, a block of data (typically 4 KB) will be transferred. This adversely affects random access to the indexes. In contrast, a single byte can be quickly read from DRAM but all data is lost in the event of a power failure. Fortunately, emerging storage technologies such as Phase Change Memory are reducing the fundamental gaps between main memory and disk.

Specifically, Intel's Optane DC Persistent Memory Modules (Optane DC PMM) offers an appealing blend of the best properties of DRAM and disk. Persistent Memory (PM) is durable like disk as well as directly byte-addressable like DRAM [63]. PM's *price*, *capacity*, and *latency* lie between DRAM and SSD [64, 65]. This allows us to allocate larger spaces for holding indexes without the need for reconstruction after a crash [66]. To the best of our knowledge, efficient indexing architectures for KGs to better leverage PM technology are not available at this time.

Motivated by the advent of PMs, a contribution of this thesis is to introduce RDFix, a specialized architecture for indexing RDF data. Typical of most access methods, RDFix can be deployed as a stand-alone indexing structure, or embedded in an RDF-store. We

experimentally demonstrate the advantages of RDFix for indexing diverse RDF datasets as compared to major read-optimized architectures such as B-trees, sorted vectors, and hash maps over PM. Our experimental analyses show that RDFix outperforms these indexing structures by at least a factor of 3.

1.6 Research Scope

This thesis presents an extensive study of the performance of existing RDF data management systems. Through this study, we experimentally show major RDF-stores' performance bottlenecks. To address these, we introduce and evaluate two solutions:

- *SymphonyDB*, a prototype that provides polyglot support for RDF query processing. It is a multi-database approach supported by an access management layer to provide a unified query interface for accessing the underlying DMSs.

- *RDFix*, a specialized architecture for indexing RDF data from the ground-up in which all data persist on PM. It can perform different operations with the lowest overhead as compared to major read-optimized indexing architectures.

The solutions presented in this thesis are for efficient query execution across diverse datasets and queries and for KGs modeled using the RDF data model. It is possible to develop techniques for supporting other data models (like labeled property graphs) in the future but we confined our focus to RDF data at this stage.

This thesis focuses on the performance of query processing over RDF datasets since RDF databases tend to be *read-mostly* if not *read-only* [44, 60]. In this case, when RDF datasets are *bulk-loaded* in the proposed prototypes, *queries* are far more frequent than *updates*. Our future work will explore the performance of query processing over *update-intensive* KGs.

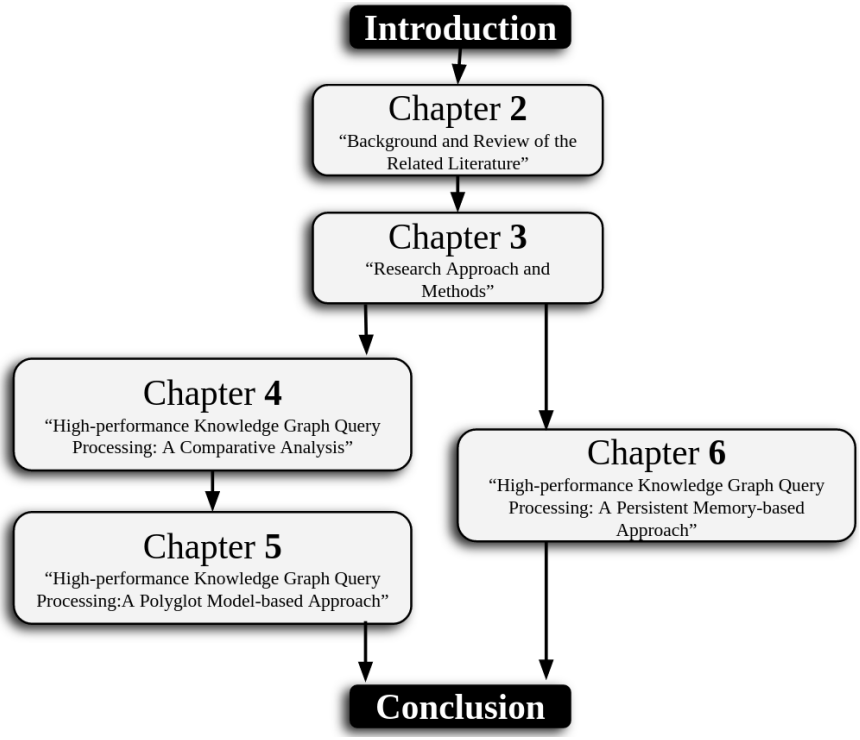


Figure 1.10: Logical organization of the thesis

1.7 Thesis Structure

Fig. 1.10 illustrates the logical organization of the thesis. An arrow from a chapter α to a chapter β indicates that the content of chapter α is a direct precondition of (and used within) the chapter β. Please note that all the dependencies between chapters are not marked, but the main logical outline is drawn.

Throughout this thesis, each chapter includes an introduction that briefly describes the main focus and context of the chapter and is concluded with a summary of the key takeaways of that chapter. An overview of each chapter is presented below.

1.7.1 Chapter 2: Background and Review of the Related Literature

While data management systems and KGs have been prevalent across a number of research disciplines, we begin by presenting an overview of the technical background and landscape as it exists today, providing insight into the state-of-the-art in related literature. It starts by introducing KGs in Section 2.1 with the goal of providing a conceptual background about them while including various important definitions, a brief history, and major application use cases. Key specifications of relevant technologies and standards, such as the RDF data model and its foundational elements, are discussed in Section 2.2. This is followed by providing background information on the SPARQL query language in Section 2.3. Archetypal KG query types are covered in Section 2.4. An in-depth discussion of KG data management systems and a review of major state-of-the-art approaches are provided in Section 2.5 with the goal of advancing our understanding of the design choices and architectures of modern KG data management systems. This is followed by providing background knowledge on query optimization in Section 2.6. Query processing in a distributed environment is discussed in Section 2.7. Finally, a summary of this chapter is presented in Section 2.10.

1.7.2 Chapter 3: Research Approach and Methods

Through this chapter, we discuss our approach to evaluating the efficacy and efficiency of services offered by the current generation of RDF-stores. We begin in Section 3.2 by clarifying the definition of diversity-tolerant RDF-stores. We use this definition to present an approach to evaluate the efficiency of RDF-stores. We proceed by determining factors that can affect the performance of an RDF-store with the goal of developing an understanding of the major role players in high-performance KG query processing. In Section 3.3, we give clear guidelines on how to detect potential inefficiencies of RDF-stores and present how to set up a control experiment for insightful comparative analyses of RDF-stores. Finally, we provide detailed

information about the basis of our evaluation, including the benchmark datasets and queries, computational environment, and system configurations.

1.7.3 Chapter 4: High-performance Knowledge Graph Query Processing: A Comparative Analysis

The main aim of our investigation in this chapter is to assess how efficiently major existing RDF-stores perform when storing diverse KG datasets and executing diverse queries over them. We begin by providing remarks concerning major factors contributing to query performance in Section 4.2) with the goal of facilitating better understanding of the rest of the chapter. This follows by presenting our experimental context in Section 4.3. We provide an exploratory analysis in Section 4.4 to discuss our experimental results. Our analyses can provide an important opportunity to advance the understanding of the performance of existing KG data management systems. We make some remarks on the lessons learned in Section 4.5 and conclude the chapter in Section 4.6.

1.7.4 Chapter 5: High-performance Knowledge Graph Query Processing: A Polyglot Model-based Approach

In this chapter, we build upon our findings from the previous chapter, proposing a solution to reduce the negative effects on performance as a result of high variance in RDF data and queries. The details of our prototype, *SymphonyDB*, including the polyglot access management and data persistence layers are presented in Section 5.2 with the intention of clarifying the way in which SymphonyDB executes diverse queries. Our experimental context and platform are included in Section 5.3. Results of the query processing and related analyses are presented in Section 5.4. We present our conclusions in Section 5.5.

1.7.5 Chapter 6: High-performance Knowledge Graph Query Processing: A Persistent Memory-based Approach

While system design inherently yields a high impact on performance gain, the way in which queries are executed, and the underlying structures they utilize, also provide numerous benefits. To this end, we propose a new index structure, RDFix, which exploits the high performance of persistent memory to persist index data with low overheads. In this chapter, Section 6.2 presents the design and architecture of RDFix. We proceed by providing experimental validation of RDFix where we experimentally demonstrate its advantages for indexing diverse RDF datasets as compared to major read-optimized architectures such as B-trees, sorted vectors, and hash maps. Our investigation led to the conclusion that index data structures with poor memory locality will typically be slower in practice than a static array-based structure with high locality, despite the additional space consumption. To further clarify this conclusion, a discussion of the space-time tradeoffs is also included to highlight that the superior performance of RDFix is achieved by trading off more memory (space).

1.7.6 Chapter 7: Conclusion

This chapter summarizes the primary contributions and core results presented throughout this study, returning to answer the questions posed in Section 1.4. During this study, we also note a number of caveats that are reviewed in Section 7.3. The findings of this study hold numerous implications for future practice that are discussed in Section 7.4. Finally, Section 7.5 presents concluding remarks with the hope to encourage the use of KGs and our proposed solutions for high-performance query execution.

Chapter 2

Background and Review of the Related Literature

A range of topics related to Knowledge Graphs (KGs) and Data Management Systems (DMS) is explored in this research. Providing preliminary background knowledge on related subjects and the state-of-the-art research in these areas mentioned above are essential, especially for readers outside of semantic knowledge management or the database communities. This chapter presents an overview of the technical background along with a review of the related literature. It starts by introducing KGs in Section 2.1 (where various definitions, a brief history, and major application use cases are included). Key specifications of relevant technologies and standards such as the RDF data model and its foundational elements are discussed in Section 2.2. This is followed by background information on SPARQL query language in Section 2.3. KG query types are covered in Section 2.4. An in-depth discussion of KG data management systems and a review of major state-of-the-art approaches are provided in Section 2.5. This is followed by an overview of query optimization in Section 2.6. Query processing in a distributed environment is discussed in Section 2.7. Finally, a summary of this chapter is presented in Section 2.10.

2.1 Knowledge Graph

2.1.1 Definition

Though the label KG has been used in the literature since at least four decades ago (e.g., [67]), the definition of a KG remains contentious as discussed in [16, 20]. A number of definitions have emerged over the last few years ranging from very specific technical definitions to slightly more general proposals. Some of these definitions are even conflicting. However, the following "*inclusive*" definition of *Hogan et. al.* [16] has recently received much more attention:

> "*a knowledge graph as a graph of data intended to accumulate and convey knowledge of the real world, whose nodes represent entities of interest and whose edges represent relations between these entities*"

In this definition, knowledge refers to "*something that is known*" in a particular domain [16]. Such knowledge can be also accumulated from external sources (i.e., harvesting), or extracted from the KG itself (i.e., inference). A similar definition for KGs is also presented by *Weikum et. al.* in [7]. More precisely, they first defined the phrase "*Knowledge Bases (KBs)*" as "*collections of machine-readable facts about the real world*" and then mentioned that "*large-scale KBs*" are also known as *KGs*. Following [7, 16], we define KGs for the purpose of this research as:

> large-scale collections of information to represent real-world entities and their interconnections

In addition, Table 2.1 outlines various other definitions of the term "Knowledge Graph" over time (in computer science).

KGs typically take a pragmatic approach by representing the central entities that match their *domain*, *scope*, and *purpose*. For instance, a KG on writers and their biographies may include *Fernando Pessoa* and one of his notable works *The Book of Disquiet*, but it may not include the book content like its *aphoristic paragraphs*. In contrast, a domain-specific KG that is particularly created for literature scholars (to analyze character relationships in literature content) may include all characters from Pessoa's works. A comprehensive KG that is constructed for libraries is more likely to include publishers' information, prices, distribution networks, notable awards, comments from experts, readers' ratings as well as many other items which are not directly relevant to the above-mentioned KG of literary scholars. Similarly, a scientific KG of drugs would include different types of drugs and their side effects, but perhaps not case histories of individuals. However, a KG that is constructed for personalized medicine is more likely to include data items related to individual patients, drugs prescribed, their clinical trials, diagnoses, treatments, and outcomes. Please note that each KG can also be enhanced with the representation of a schema (sometimes referred to as *ontologies*) to explicitly define the meaning of high-level terms (sometimes referred to as *vocabulary* or *terminology*) used in the KG (more details can be found inSection 2.2.1).

After covering the definitions and the pragmatic approach of KGs to represent real-world entities and their interconnections, we now provide some background information on the history of KG in the next section.

Source	Year	Definition
Marchi and Miquel [68]	1974	"A mathematical structure with vertices as knowledge units connected by edges that represent the prerequisite relation"
Vries [69]	1989	"A directed graph that distinguishes three types of asserted relationships: (1) an object has a certain property, (2) an object is an instance of another object, (3) a change in a property of an object leads to a change in another property of that object"
James [70]	1992	"A knowledge graph is a kind of semantic network... One of the essential differences between knowledge graphs and semantic networks is the explicit choice of only a few types of relations"
Zhang [71]	2002	"A new method of knowledge representation, [which] belongs to the category of semantic networks. In principle, the composition of a knowledge graph is including concept (tokens and types) and relationship (binary and multivariate relation)"
Popping [72]	2003	"A particular kind of semantic network"
Singhal [10]	2012	"A graph that understands real-world entities and their relationships to one another: things, not strings"
Ehrlinger and Wöß [20]	2016	"A knowledge graph acquires and integrates information into an ontology and applies a reasoner to derive new knowledge"
Krötzsch and Weikum [73]	2016	"Knowledge graphs are large networks of entities, their semantic types, properties, and relationships between entities"

Table 2.1: Selected definitions of KG (source: [74])

2.1.2 Historical Overview

As discussed in [7], the concept of comprehensive KGs containing real world's entities and their relationships goes back to pioneering works in AI such as constructing *universal knowledge bases* in the 1980s and 1990s, with *Cyc* [75] and *WordNet* [76], being two of the most notable examples. These and other early KGs have typically been created manually and were somehow limited in scope and scale. However, since the very beginning of the 2000s, the *Semantic Web* and the *knowledge harvesting* from the Web and other sources (like text documents) have become principal research avenues with a broad range of substantial practical impacts on the automatic construction of large-scale KGs enabling us to go beyond the earlier generation of KGs like the WordNet or Cyc.

The list of most notable projects for constructing large-scale KGs includes: *Wikidata*, *Freebase*, *BabelNet*, *DBpedia*, *KnowItAll*, *WikiTaxonomy*, *WebIsALOD*, *ConceptNet*, *Probase*, *XLore*, *KnowledgeVault*, *WebOfConcepts*, and *YAGO*. These KGs contain billions of entities and hundreds of thousands to hundreds of millions of their interrelations with new content being added continually. Most, if not all, of them are primarily rooted in academic research (or community) projects. However, the construction and use of KGs were not limited to academia.

The deployment of KGs has become more prevalent in big organizations and industries following the introduction of the abovementioned KGs. As a result, since early 2012, large-scale KGs have become a significant asset in a variety of commercial use cases and applications, including semantic search, data integration, text analytics, and recommendations. Major examples of these *enterprise* KGs include the deployment of the *Google Knowledge Graph* [10], *Microsoft Satori* [11], *IBM Watson* [12], *Amazon Product Graph* [13, 14] as well as many other domain-specific KGs in business, geography, biology, art, finance, life sciences, etc. for a variety of semantic and other applications. More details on the history of KGs can be found in [7, 16].

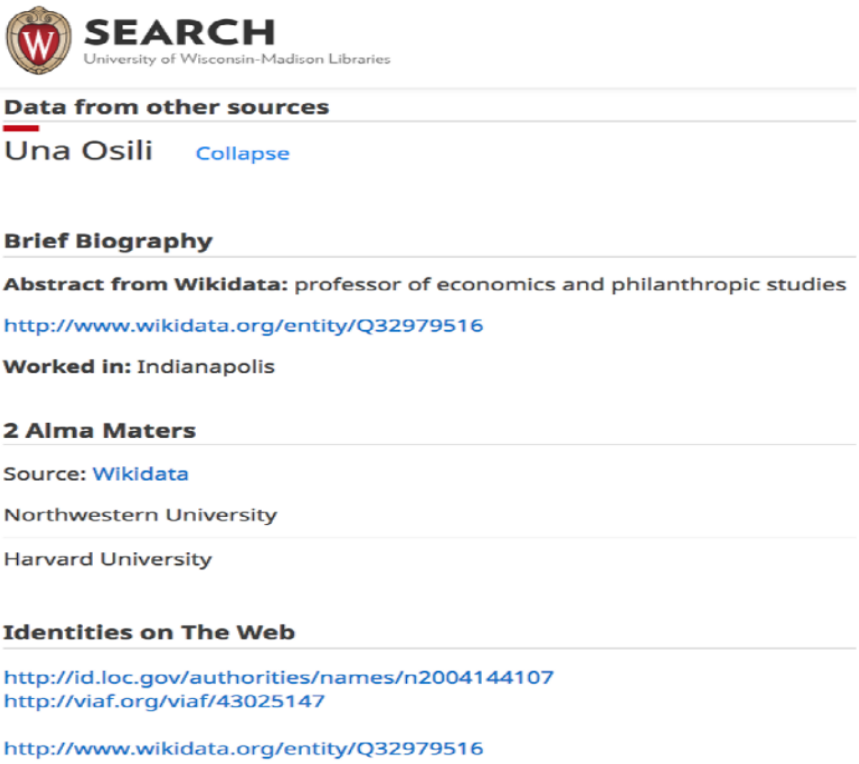


Figure 2.1: Una Osili's information in the University of Wisconsin–Madison KG (source: [77])

Having covered the brief history of KGs, we now provide some background information on some prominent KGs and their applications in the next section.

2.1.3 Knowledge Graphs in Practice

We have hinted at some of the application use cases of KGs in Section 1.2.3. In this section, we continue this discussion. KGs are utilized in a range of applications. Some of them are discussed below.

2.1.3.1 Galleries, Libraries, Archives, and Museums

Galleries, Libraries, Archives, and Museums (GLAMs) use open KGs like Wikidata for many different purposes like data integration and

semantic search. For instance, The University of Wisconsin–Madison Libraries deployed a KG called "BibCard" for representing authors and their publications as illustrated in [77]. To construct BibCard, identifiers of authors are extracted from other KGs such as Wikidata and DBpedia. BibCard is then used as a basis for semantic search. An example is shown in Fig. 2.1 (source: [77]) where Wikidata is used to expand the University of Wisconsin–Madison KG (author's biographical data). GLAMs have a high interest in using KGs for enhancing their catalogs as well as improving their archival and collection discovery.

2.1.3.2 Scholarly Knowledge

Constructing KGs for representing scientific publications, authors, and research organizations is another important use case. Prominent examples include CiteSeerX[1], Semantic Scholar[2], AMiner[3], and Scholia[4]. Using these KGs can assist in the semantic analysis of scholarly topics, authors' network detection, citation measurement, etc. These KGs can be viewed as open alternatives and value-added extensions to well-known services like Google Scholar and Microsoft Academic [7].

2.1.3.3 Entity Identification and Science Stories

KGs are widely in use for "*entity identification*" and "*cross-linkage*" between entities. In particular, Wikidata is taking up the central role for these purposes. Extracting entity identifiers like TwitterID, VIAF-ID, or GoogleScholarID can assist in interlinking entities of different datasets. An interesting use case is a recent project at Yale University called "Science Stories" [78]. In this project, a KG is constructed by extracting data from Yale University Library, Wikidata, and some other sources to identify Yale alumni who are women in

[1] https://citeseerx.ist.psu.edu/
[2] https://www.semanticscholar.org/
[3] https://www.aminer.org/
[4] https://scholia.toolforge.org/

the sciences. Based on the constructed KG, this project hopes to publish a "science story" for each prominent alumna.

2.1.3.4 Biomedicine and Health

Domains like biomedicine and health are specific domains in which KGs have the potential to play the role of a data hub [79]. KGs can facilitate the representation of information about diseases, drugs, proteins, etc. for scientists. KGs' rich coverage of identifiers can also help scientists to integrate different datasets (like biomedical datasets) in a user-friendly manner [7].

2.1.3.5 Checking Credibility

Over the past few decades, the number of online platforms for sharing information has increased. This led to an information explosion, especially in terms of volume and velocity. A large proportion of this information can potentially be misinformation (false) or disinformation (intentionally false). One of the major research directions toward fact-checking and false news detection can be assessing the credibility of information sources using KGs that are constructed by extracting information from polarized news sources or discussion forums. Such a research effort is still in its early stages. However, the major roles that KGs can play in the area of fact-checking are already recognized [7].

Having covered some application use cases of KGs in practice. In the next section, we provide some background knowledge on key factors that enable us to better leverage KGs. Throughout, we present concrete examples in the context of several hypothetical KGs.

2.1.4 Key Requirements to Unlock the Full Potential of KGs

The key requirements to unlock the full potential of KGs in response to the growing deployment include:

- **RDF Data model.** It has widely been accepted for representing the content of KGs.
- **SPARQL Query language.** It is widely used for querying the RDF data model.
- **RDF Data Management Systems.** These are largely in use for storing RDF data (i.e., the KG content modeled in RDF) and executing SPARQL queries over them.

As mentioned, RDF as a directed and labeled graph-like structure is typically used for *representing* the content of a KG using a large set of triples of the form <subject predicate object>. The subject is the entity from which the edge emanated, the predicate is the label of the edge, and the object is the name of the second entity [51]. RDF has been widely accepted as a standard for representing the content of KGs. However, while this is necessary, it is not sufficient since users need to query and use the data in different ways. To meet this need, SPARQL as an RDF query language was proposed. It is an SQL-like language in which queries are based on triple patterns. Triple patterns are similar to RDF triples except that each subject, predicate, and object may be a variable [31]. A number of data platforms have accordingly been developed over the last few years for RDF data management purposes. These platforms, also referred to as RDF-stores, enable us to store RDF data and execute SPARQL queries over them. We provide a preliminary background on the RDF data model, SPARQL query language, and RDF-stores, as well as the state-of-the-art research in these areas, in the next sections.

2.2 RDF Data Model

As mentioned previously, a KG can represent *knowledge* of the world using a set of nodes and edges where each node represents an entity of interest in a specific domain and the graph edges are to represent interrelations between these entities [16]. Using RDF, as an agreed-upon and common data model, enables us to seamlessly share and exchange the KG content across different applications. RDF is generic

and simple enough to express almost any body of information (any arbitrary data) to provide a canonical representation irrespective of the knowledge domain of interest [16]. In addition, it is structured enough for a machine to understand and process its content using generic and off-the-shelf software packages and technologies [80, 81, 82]. RDF's foundational elements are discussed below:

2.2.1 Foundational Elements

We provide background information on the foundational elements of the RDF data model below:

- **Characters (Unicode):** Almost all data models require some standard for encoding and manipulating textual information. For this, the RDF data model largely relies on the well-known and standard *Unicode character-set*.

- **Entities:** This is probably the most fundamental element in RDF. Weikum et. al. [7] defined an entity as "*any abstract or concrete object of fiction or reality*". This definition easily includes almost *anything* in the world such as people (living or dead), locations, organizations, books, universities, products, flights, events, poems, songs, athletes, fictional characters (e.g., *Apollo*, *Coeus*, *Chacha Chaudhary and Sabu*, etc.) as well as "abstract" concepts such as empathy and kindness.

- **Identifiers (URI/IRI):** To unambiguously refer to entities, we need to use a unique name for each entity. This name has to refer to only a single entity. For instance, we use identifiers such as ISBNs for books or DOIs for publications to distinguish different entities. In RDF, using the Uniform Resource Identifier (URI) specification could be a *natural* choice since it is already used on the Web to identify documents. However, a generalization of URIs (by using globally agreed-upon identifiers) supporting the broader Unicode standard has been adopted in practice by RDF that is called Internationalized Resource Identifier (IRI). An IRI is generally a long

```
<http://ex.com/s1> <http://ex.com/p1> <http://ex.com/o1> .
```

Figure 2.2: Absolute IRIs are used to represent an example triple.

```
PREFIX pre: <http://ex.com/>
pre:s1 pre:p1 pre:o1 .
```

Figure 2.3: Using prefixed names to represent the same triple in Fig. 2.2.

string of characters to uniquely denote only one entity. In practice, IRIs are less likely to be human-readable or even in a form that is easily interpretable by a human. For example, *Shakuntala* (wife of Dushyanta and mother of Emperor Bharata) as the protagonist of the famous play *The Sign of Shakuntala*, has an unwieldy (but globally unique) identifier like *https://www.wikidata.org/wiki/Q955123* (in the *Wikidata* KG). We may prefer to refer to her more easily by labels like "Shakuntala", but "Shakuntala" alone could also refer to the play itself. However, referring to entities using IRIs omits these ambiguities.

- **Prefixes:** A prefixed name is a prefix label and a local part, separated by a colon ":".[5] The '@prefix' or 'PREFIX' directive associates a prefix label with an IRI. For instance, Fig. 2.2 shows an example triple with all absolute IRIs while Fig. 2.3 shows the same triple represented using prefixed names.

- **Serialization (XML/Turtle (N-Triples)/JSON-LD):** To automatically parse RDF content, it is required to use common and agreed-upon syntaxes with clearly defined grammars to serialize RDF files (and store them on the disk, etc.). In practice, RDF serialization syntaxes may be divided into two main categories: generic syntaxes and custom syntaxes.

[5] https://www.w3.org/TR/turtle/#prefixed-name

```
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:Organization
dbr:Tesla_Inc dbp:name ''Tesla, Incorporation''
dbr:Tesla_Inc dbp:date ''2003-07-01''
```

Figure 2.4: A subset of RDF triples from DBpedia describing Elon Musk.

  - **Generic syntaxes.** Using well-known and generic syntaxes such as the XML and JavaScript Object Notation (JSON) (which have been dominant in the typical web contents) is not uncommon to serialize RDF files. For instance, *JSON-LD* is a JSON-based syntax for representing data in RDF. JSON is largely used as a serialization format by many Web-based applications. Based on this, using the JSON-LD syntax allows Web developers to parse RDF graphs similar to Javascript objects using the legacy JSON parsing mechanisms available in the scripting languages of their choice.
  - **Custom syntaxes.** Custom syntaxes have also been developed like Terse RDF Triple Language (Turtle) and its well-known subset *N-triples*. These custom grammars (especially *N-triples*) tend to be more human-readable (or at least interpretable by a human) than generic syntaxes like XML [81, 82]. For instance, Fig. 2.4 shows a subset of RDF triples from DBpedia describing Elon Musk represented in the Turtle format. This example is also used in [83].

- **Schema (sometimes referred to as *Ontologies*):** A KG can *potentially* be enhanced with the representation of a schema.

```
1  PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
2  PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
3  PREFIX dbr: <http://dbpedia.org/resource/>.
4  PREFIX dbo: <http://dbpedia.org/ontology/>.
5  PREFIX dbp: <http://dbpedia.org/property/>.
6  dbr:Elon_Musk rdf:type dbo:Person
7  dbr:Tesla_Inc rdf:type dbo:Organization
8  dbo:Person rdf:type rdfs:Class
9  dbo:Person rdfs:label ‘‘Person’’
10 dbo:Organization rdf:type rdfs:Class
11 dbo:Organization rdfs:label ‘‘Organization’’
12 dbp:birthDate rdf:type rdf:Property
13 dbp:birthDate rdfs:label ‘‘birth date’’
14 dbp:birthDate rdfs:domain dbo:Person
15 dbp:birthDate rdfs:range rdfs:Literal
16 dbp:birthName rdf:type rdf:Property
17 dbp:birthName rdfs:label ‘‘birth name’’
18 dbp:birthName rdfs:domain dbo:Person
19 dbp:birthName rdfs:range rdfs:Literal
20 dbp:name rdf:type rdf:Property
21 dbp:name rdfs:label ‘‘name’’
22 dbp:name rdfs:domain dbo:Organization
23 dbp:name rdfs:range rdfs:Literal
24 dbp:date rdf:type rdf:Property
25 dbp:date rdfs:domain dbo:Organization
26 dbp:date rdfs:range rdfs:Literal
```

Figure 2.5: The RDF schema information corresponding to the RDF triples in Fig. 2.4.

This *optional* schema may be embedded in RDF, or layered above it using prominent standards like RDF Schema (RDFS). A schema typically defines *classes* and *properties* used in the KG. Using a semantic schema potentially enables us to explicitly define the meaning of high-level terms (sometimes referred to as *vocabulary* or *terminology*) used in a KG. For instance, Fig. 2.5 shows the RDF schema information corresponding to the RDF triples in Fig. 2.4 using the Turtle format. This example is also used in [83]. In short, using a semantic schema can enable us to explicitly define the meaning of high-level terms used in the KG. However, as mentioned above, using a semantic schema is *optional*. The most important benefit of modelling KGs using the RDF data model (versus, for exam-

ple, the relational model) is perhaps the option to totally forgo (or postpone) the definition of a rigid *semantic schema* [16].

- **Querying:** We ultimately need to process KG content by executing queries over its RDF representation. This enables us to retrieve desired information by explicitly specifying *conjunctive conditions* and *query patterns*. The well-known and common standard for querying RDF data is SPARQL[6] which provides a "*mature*" and "*feature-rich*" language for querying KG content represented in conformity with the RDF data model.

We have provided background information on the foundational elements of the RDF data model above. We present a walkthrough of the main design principles and features of RDF in the next section.

2.2.2 Design Principles

Although RDF was initially proposed by the W3C as a standard for modeling *Web entities* on the Semantic Web, its use is now wider than the Semantic Web. It can be used for representing almost any body of information [26, 84]. In this section, we provide a walkthrough of the design principles and the features of RDF. We focus on core concepts that are important for further reading of this research.

2.2.2.1 Triples

In RDF, each entity can be simply described in terms of its associated predicates (also referred to as properties, attributes, or relations) and values. Based on this, each RDF dataset can be viewed as a collection of statements about entities and their interrelations, called *triples*, of the form $<s,p,o>$ where s is a subject, p is a predicate, and o is an object [16, 17, 18, 19, 20, 21, 22]. In RDF triples, subjects and objects denote the entities and relationships between them are represented as predicates. In each triple, the subject is the identifier of an entity, the object is the value for the predicate (the predicate itself

[6] https://www.w3.org/TR/sparql11-query/

is an identifier to be distinguished unambiguously) of the described entity. The object of a triple can be also the subject of another triple and vice versa. In this case, an identifier is typically used to refer to the object as well. Each RDF dataset can have an isomorphic representation in the form of a graph, where different entities are nodes in the graph, and relationships between them are represented as labeled edges. There is one triple corresponding to each edge in the RDF graph. A single RDF triple is generally the minimum (and atomic) unit of information in the RDF data model [23].

2.2.2.2 Basic Terms

In principle, each element of an RDF triple (i.e., subject, predicate, or object) belongs to one of the following *RDF terms*:

- **IRIs:** We have already mentioned in Section 2.2.1 that IRIs serve as global identifiers (preferably Web-scope or at least KG-scope) to refer to entities unambiguously. For example, *https://dbpedia.org/page/Jupiter* is used to identify *Jupiter*[7] (the fifth planet from the Sun and the largest in the Solar System) in DBpedia, i.e., an online KG extracted from Wikipedia content. If we use N-Triples to serialize RDF triples, IRIs should be enclosed in *angle-brackets* "<" and ">" similar to the following example: *<https://dbpedia.org/page/Jupiter>*. Using the N-triples syntax, we need to repeat writing this IRI n times if we need to refer to Jupiter n times. If we use the N-Triples serialization format to store a KG dataset in a file, each individual line of the file should contain all necessary information to parse the triple on that line independent of the rest of the document. This is perhaps the reason behind the popularity of N-triples for a broad range of applications, including RDF stream processing and fault-tolerant line-at-a-time processing [81, 82].

- **Literals:** Literals are a set of lexical values such as strings, dates, and numbers. In principle, *anything* can be represented

[7] https://dbpedia.org/page/Jupiter

by a literal [80]. Literals can either be *plain literal* or *typed literal*.

  - **Plain literal:** It is typically a plain string with an optionally additional language tag such as "*Ciao bella*"[8], potentially with an associated language tag such as "*Ciao bella*"@it.
  - **Typed literal:** It comprises a lexical string and a datatype, such as "16"^^xsd:int. More specifically, it is a string combined with a datatype URI. For example, to refer to a specific date the following typed literal can be used: "2021-01-01"^^xsd:date in which "*xsd:date*" is a datatype URI. Typically, simple datatypes are used in RDF triples such as numerics, date-time, and booleans. These datatypes also determine which plain strings are valid for that datatype. For example, "pi"^^xsd:int is invalid since "*pi*" is not an integer. In the RDF serialization syntaxes such as Turtle (or N-Triples), numbers and boolean values can be used without using quotes as delimiters. Plain literals without using the optional language tags can be mapped to their identical typed literals by combining them with the string datatype URI (i.e., "xsd:string").

- **Blank nodes (BNodes):** These are defined as variables to refer to the existence of some entities without using any specific IRIs or literals. Blank nodes are sometimes referred to as anonymous entities [80]. In practice, blank nodes serve as locally-scoped identifiers for anonymous entities. In other words, blank nodes cannot be referenced outside of their originating KG. Blank nodes are thus only significant within a local scope. A blank node can be explicitly denoted using an underscore prefix like "_:" in many RDF serialization formats such as Turtle and N-Triples.

[8] An informal Italian expression literally meaning: "*goodbye/hello beautiful*"

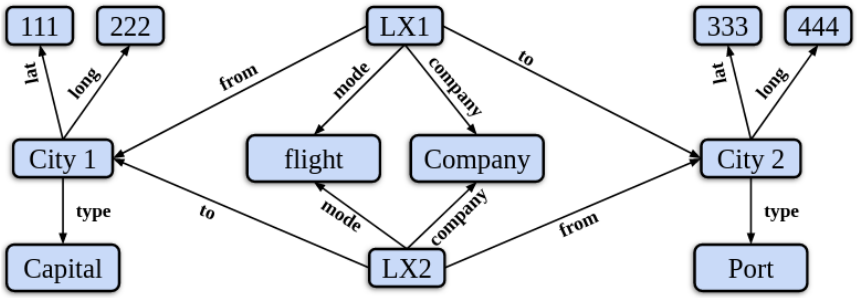


Figure 2.6: An example KG informally modeled in RDF to represent companies offering flights between two cities (based on an example from [16])

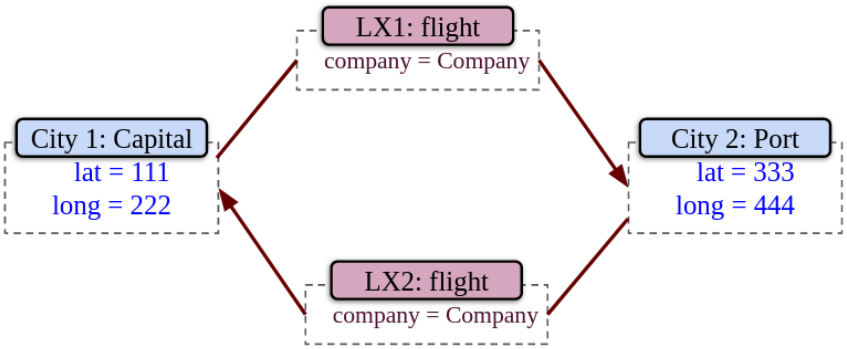


Figure 2.7: The KG in Fig. 2.6 is modeled in a property graph (based on an example from [16])

- **Named graphs:** Each "named graph" can be viewed as a subset of a KG. More specifically, each KG may hold multiple subgraphs and "name" each graph so as to allow an application to execute a query either over the entire KG as a whole or over specific subgraphs as subsets of it. Each named graph is typically identified by an IRI.

We have already mentioned that it is not uncommon to conceptualize RDF datasets as directed labeled graphs, where each subject and object is drawn as a labeled node and predicates are drawn as directed, labeled edges. RDF is a standardized data model based on this directed edge-labeled graph-like structure. The use of other *graph-structured* data models like *property graphs* is also feasible to

represent KG content whereby additional information can be assigned to the graph edges or nodes in the form of a set of "*property-value pairs*", also known as "*attributes*" [15]. For instance, Fig. 2.6 shows an example KG informally modeled in RDF to represent companies offering flights between "City1" and "City2" [16]. The same KG can be modeled in a property graph as shown in Fig. 2.7. Using alternative data models like *property graphs* can provide additional flexibility as compared to the RDF data model. However, RDF has been argued to offer a more "*minimal*" data model [15]. As a result, it has widely been accepted as a standard for representing the content of KGs.

Thus far, we have covered the core features of RDF. At the outset of this chapter, we also provided background information on KGs and highlighted the foundational elements and major design principles of the RDF data model to represent KGs content. We provide background information on the concept of structuredness in the next section.

2.2.3 RDF Structuredness

We have provided background knowledge on the concept of structuredness in Section 1.3.1. The way in which we can compute the structuredness of an RDF data is presented in [50] as follows: the first step for computing the structuredness of an RDF dataset is the determination of the type system of a dataset. Conceivably, one might infer the type system T of a dataset D intensionally, through an RDFS specification associated with D. In practice, however, many datasets do not come with such specifications. As discussed in [46, 80], RDF triples in the "*wild*" tend not to conform to ontologies. To address the fact that often there is no schema (sometimes referred to as ontology information) available in datasets to apply an intensional approach, we need to determine the type system T of D extensionally through the dataset itself, and thus we do not need any schema-level information. Specifically, we can scan D looking

for triples whose predicate is <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>, and for these triples, we need to extract the type T that appears as the object of the triple. We then determine the predicates of a type T through the union of all the predicates that the instances of type T have. We need to count the number of entities for which predicate p has its value set in the instances. The structuredness of T is affected by the sparsity (presence) of its predicates across instances. The structuredness of each type is computed based on the instances of the type that set a value for all its predicates (see Section 1.3.1). The structuredness of the whole RDF dataset is finally computed using a weighted average of the structuredness of all types.

Any RDF dataset's level of structuredness can be quantified by a normalized value in the [0, 1] interval, with values close to 0 corresponding to low structuredness, and 1 corresponding to high structuredness. In an RDF dataset with high structuredness, we expect that for any two instances of the same type, the instances have exactly the same predicates.

Having covered background information on KGs as well as the core features of the RDF data model. We provide background information on SPARQL (SPARQL Protocol and RDF Query Language) which is a semantic query language for retrieving and manipulating data from RDF in the next section.

2.3 SPARQL Query Language

SPARQL is designed specifically for RDF data. In 2008, its original specification became a W3C recommendation. An extension of the original SPARQL became a W3C recommendation in 2013. In this section, we focus on the core features of this standard.

2.3.1 Syntax

SPARQL is a SQL-like (i.e., the Structured Query Language used for querying relational databases) query language using very similar

keywords for processing RDF data but the SPARQL syntax is closely tied with the RDF-specific syntax. On a high level, each SPARQL query can consist of up to five main parts as follows: *Prefix Declarations*, *Dataset Clause*, *Result Clause*, *Query Clause*, and *Solution Modifiers*. These parts are explained below:

- **Prefix Declarations:** To define URI prefixes (similar to Turtle's `@prefix` directive) in order to use shortcuts later in the query. More information can be found in Section 2.2.1.

- **Dataset Clause:** To specify the targeted part of the RDF dataset over which the query is going to be executed.

- **Result Clause:** To specify what type of SPARQL query (i.e., SELECT, ASK, CONSTRUCT, or DESCRIBE) is being executed, and (if applicable) what results will be returned after the execution. Please note that we only focus on the SELECT query type in this research. This query type is to extract matched (RDF) graph patterns specified by the input SPARQL query.

- **Query Clause:** To specify the query patterns, conjunctions, disjunctions, and optional patterns that are matched against the data for generating variable bindings. More details can be found in Section 2.3.2.

- **Solution Modifiers:** To specify any modification of a query's results. More specifically, using solution modifiers allows us to apply classical operators such as ORDER BY (ordering the result), LIMIT (defining the desired maximum number for results), DISTINCT (removing all duplicates in the result if any), REDUCED (eliminating some duplicate results), OFFSET (skipping the position in the overall sequence of results), PROJECT (choosing desired variables to be part of the result set).

```
#Prefix Declarations
    PREFIX dbp: <http://dbpedia.org/property/>
    PREFIX dbr: <http://dbpedia.org/resource/>
    PREFIX foaf: <http://xmlns.com/foaf/0.1/>
    PREFIX dbo: <http://dbpedia.org/ontology/>
#Result Clause
    SELECT ?fullName  ?birthDate
#Dataset Clause
    FROM <http://dbpedia.org>
#Query Clause
    WHERE {
        ?person foaf:name       ?fullName       .
        ?person dbp:birthPlace  dbr:Wollongong  .
        ?person rdf:type        dbo:Swimmer     .
        ?person dbo:birthDate   ?birthDate      .
    }
#Solution Modifiers
    LIMIT 2
```

Figure 2.8: An example SPARQL query

Fig. 2.8 presents an example of a SPARQL query containing each of the above five parts. Comment lines are prefixed with the character "#". The shortcuts of IRIs' prefixes are defined by using the "*PREFIX*" directive at the beginning of the query to be used later in the *Query Clause* part. The *Dataset Clause* selects the targeted parts of the KG over which the query should be run. In this example, the targeted part is defined using the *Dataset Clause*. As specified in the *Result Clause*, this query looks for *swimmers* whose birthplace is Wollongong. The *Result Clause* explicitly states what data items should be returned for the query (i.e., only full name and birth dates). The *Query Clause* is to define the desired patterns that the query should match against. The desired place of birth (i.e., Wollongong) is stated in this part of the query. As specified by the *Solution Modifier* (LIMIT 2), the number of returned results is limited to 2. Thus, if the matching patterns for the *Query Clause* are found from the *DBpedia* KG, 2 matching patterns will be returned as the result of the query. For instance, the expected result would be:

```
  fullName                  | bithDate
----------------------------|-------------
‘‘Emma McKeon’’@en          | 1994-05-24
‘‘Beverley Whitfield’’@en   | 1954-06-15
```

where the header indicates the variables (i.e., *fullName* and *bithDate*) for which the respective results are returned, based on the *Result Clause* of the query (or more precisely based on the given *SELECT clause*).

We have provided an overview of the different parts of a query in the foregoing. In the next section, we focus on the SPARQL query clauses which will be used in later chapters of this thesis.

2.3.2 Query Clause

The SPARQL query clause is indicated by using the keyword "*WHERE*" surrounded by braces like "{" and "}" (i.e., the opening and closing braces). The SPARQL query clause would be simply referred to as the *WHERE clause* as well. This clause specifies the query patterns and other criteria that query variables must match to be returned. In its typical form, a *WHERE clause* contains one or multiple triple patterns. Any given triple pattern should contain three elements as the subject, predicate, and object of the pattern. Each of these elements might be a variable. For instance, the *WHERE clause* of the example SPARQL query shown in Fig. 2.8 contains four triple patterns in which "`?person dbo:birthDate ?birthDate`" is a triple pattern whose subject and object are variables. Triple patterns need to be executed against the underlying KG to retrieve the result. A basic graph pattern can comprise multiple triple patterns, considered as conjunction. The term Basic Graph Pattern (BGP) is used to refer to each conjunctive set of triple patterns. Based on this, the query shown in Fig. 2.8 contains only one BGP. A BGP query is very similar to a "*SQL inner-join query*" [85].

In a given SPARQL query, a BGP is generally identified by a conjunctive set of triple patterns surrounded by the opening and closing braces "{" and "}". BGPs can play the role of building blocks

to formulate more complex queries in various ways. There are other features that can be used along with BGPs to create complex query patterns such as *GRAPH*, *UNION*, *OPTIONAL*, and *FILTER*. These are explained below:

- **GRAPH:** As already mentioned (see Section 2.2.2.2), each KG may hold multiple subgraphs and name each graph so as to allow an application to execute a query either over the entire KG as a whole or over specific subgraphs as subsets of it. Each named graph is typically identified by an IRI. When used with an IRI, the keyword "*GRAPH*" specifies the subset of KG (i.e., the *named graph* from the KG) against which a BGP should be evaluated (matched). When a BGP is surrounded by the *GRAPH clause*, it will not be matched against the whole KG. It will be matched only against the specified *named graphs*.

- **UNION:** It specifies a disjunction of query patterns that the query should match to return the result, i.e., matching on one of several alternative graph patterns. The result of the query is the disjunction (the union) of all the matchings generated for each query pattern.

- **OPTIONAL:** It specifies optional patterns that a query should try to match. As mentioned above, if we consider that a BGP query is very similar to an "*SQL inner-join*" query, we can consider that a query containing the *OPTIONAL* pattern is very similar to an "*SQL left-outer-join*" query [85].

- **FILTER:** It can be used to specify a broad range of constraints and conditions that a query result should satisfy to be returned. Conditions can be combined by BGPs to create far more complex query patterns. The result of a query can be viewed as an input that some conditions might be used to filter this input (based on the specified conditions) to return the final result to the user. Conditions can comprise various operators and functions as explained below:

- **Operators.** A filter expression can contain several operators, including *equality* and *inequality* operators. Less than or greater than operators can be used for range filtering in queries.
- **Built-in Functions.** These functions can be used to check whether an RDF element is an IRI, blank node, or literal (or even unbound). *Regular expression* functions can also be used as well as text parsing functions for literals, e.g., to test the additional language tags of a literal. Datatype URIs can also be checked using the built-in functions.
- **Casting Operations.** When any conversion between different types of datatype literals is needed, we use casting operators. Converting URIs to a string is also allowed.
- **Boolean Connectives.** Filter expressions can be combined using Boolean connectives, including binary and unary connectives such as conjunction ("&&"), disjunction ("||"), and negation ("!").
- **User-defined Functions.** It refers to custom built-in functions that might be defined by RDF data management systems (if existing SPARQL filtering functions do not fully satisfy their requirements).

We have provided an overview of the different parts of a query in this section. We have also presented the core specifications of the SPARQL query clauses. The results of a specific query can then be further modified through *result modifiers* before being returned. In the next section, we discuss the result modifiers such as sorting.

2.3.3 Result Modifiers

To post-process results generated from the query clause, we can use the *result modifiers*. The following modifiers can be used in SPARQL:

- **ORDER BY.** No default ordering is specified for returning results in SPARQL query processing. In other words, the default ordering is non-deterministic. However, if we use the *ORDER BY* clause, the ascending sorting will be performed by default but the optional keywords: "*ASC*" and "*DESC*" can be used to specify whether the sorting should be in ascending or descending order.

- **LIMIT.** The LIMIT clause can be used to define the desired maximum number of results. This clause allows us to specify a non-negative integer n, where n defines the maximum number of results to be returned.

- **OFFSET.** The OFFSET clause can be used for skipping the positions in the overall sequence of results. By defining a non-negative integer n for this clause, we can skip over the first n results. As mentioned above, no order is specified by default for returning results in SPARQL query processing. Thus, default ordering can be non-deterministic. As a result, using the OFFSET clause is only useful in combination with the ORDER BY clause. In brief, the combination of OFFSET, LIMIT and ORDER BY clauses allows for a form of "*pagination*" of results [81, 82].

Thus far, we have provided background information on core features of SPARQL including the above-mentioned modifiers (that can be applied directly to any SELECT query for post-processing of the results). Having discussed these topics, we proceed to present background information about the KG query types in the next section.

2.4 Knowledge Graph Query Types

In this section, we present background information about the KG query types. As already mentioned, we can use an RDF graph to represent the content of a KG about the Opera House (see Fig. 1.1). We use this KG as our running example to present KG query types in this section.

When we have a KG represented in RDF at hand, we can use SPARQL to execute different queries over it. Similar to an RDF graph, each SPARQL query can also be viewed as a directed graph where nodes are formed by the subjects and objects of the query's triple patterns and edges are the predicates of these patterns [80]. Variables of a query can also be viewed as nodes or edges. Based on this graph representation, each SPARQL query can further be classified into shape-specific categories. In this research, we use the label "*KG query types*" to refer to the common shape-specific categories of SPARQL queries. In practice, there are five *KG query types* as follows:

- Single Triple Pattern
- Star-shaped queries (aka, subject-subject joins)
- Chain-like queries (aka, subject-object, path, or linear joins)
- Tree-like queries (aka, snowflake-shaped queries)
- Optional joins (aka, left-outer-join or OPT clauses).

Please note that a widely accepted typology of KG queries is yet to emerge. At this stage query types such as subject-subject, subject-object, tree-like, and optional queries have been analyzed in previous research like [19, 26]. The importance of optional queries has been highlighted in [85]. In the next sections, we provide information about the above-mentioned query types using the *Opera House* KG. Please note that we express queries informally in pseudocode.

2.4.1 Single Triple Pattern

We start with a simple query containing only one triple pattern. An example of this type of query is given below. Please note that we informally expressed queries of this chapter in pseudocode to increase their readability.
This query asks for the subject "OperaHouse's" architectural style name. "?styleName" is a variable to return the associated value as the

```
SELECT ?styleName
WHERE {
OperaHouse style ?styleName .
}
```

result (i.e., "expressionist"). This query has only one triple pattern but KG queries typically contain more than one (see Section 2.3.2). In this case, the result of each triple pattern needs to be *joined* with the results of other triple patterns to return the final resultset. Different types of joins are explained in the following.

2.4.2 Star-shaped Queries (Subject-subject Joins)

A subject-subject join is performed when a KG query has at least two triple patterns such that the predicate and object of each triple pattern is a given value (or a variable), but the subjects of both triple patterns are replaced by the *same* variable [19, 26]. A star-shaped query is given below:

```
SELECT ?x
WHERE {
?x  style       "expressionist" .
?x  located_in  "Sydney"        .
}
```

This query looks for all subjects of the KG in Fig. 1.1 that are located in "Sydney" and their style is "expressionist". Its result will be "OperaHouse".

2.4.3 Chain-like Queries (Subject-object Joins)

A subject-object join is performed when a KG query has at least two triple patterns such that the subject of one of the triple patterns and the object of the other triple pattern are replaced by the same variable [19, 26]. A chain-like query is given below:

```
SELECT ?y
WHERE {
?x  located_in  "Australia"  .
?y  located_in  ?x           .
}
```

The above query looks for all subjects that are located within Australian cities. This query will return the following result: "OperaHouse".

2.4.4 Tree-like Queries

Queries belonging to this type consist of a *combination* of subject-subject and subject-object joins [19]. An example is given below:

```
SELECT ?y
WHERE {
?x  opening_date     ?y           .
?x  located_in       ?z           .
?z  instance_of      "capital"    .
?z  instance_of      "metropolis" .
}
```

This query requires a tree-like join to look for the opening date of "OperaHouse" (the result will be "20 Oct. 1973"). In this example, two subject-subject joins and one subject-object join are combined.

2.4.5 Optional Joins

Queries return resultsets only when the entire query pattern matches the content of the KG. However, optional joins allow KG queries to return a resultset even if the optional part of the query is not matched since completeness and adherence of KGs' content to their formal ontology specification is not enforced [85]. For example, the following query uses optional join (in addition to a subject-subject join) to return "OperaHouse" as one of Sydney's tourist attractions.

```
SELECT ?x
WHERE {
?x  instance_of  "tourist attraction" .
?x  located_in   "Sydney"             .
OPTIONAL {?x  instance_of  "zoo"      .}
}
```

Recall that a query containing the *OPTIONAL* pattern is very similar to an "*SQL left-outer-join*" query [85].

Thus far, we have covered the KG query types. At the outset of this chapter, we also provided background information on KGs and highlighted the foundational elements and major design principles of the RDF data model to represent KGs content. We have also provided background information on SPARQL which is a semantic query language for retrieving and manipulating data from RDF. In the next section, we provide an overview of major RDF data management systems and highlight the main approaches as well as the recent research efforts in this area.

2.5 RDF Data Management Systems

2.5.1 Overview

The ever-growing use of KGs calls for the efficient processing of queries over them. A number of data platforms have accordingly been developed over the last few decades for RDF data management purposes. These platforms that are also known as RDF-stores can store RDF data and execute SPARQL queries over them.

In this section, we provide an overview of major state-of-the-art RDF-stores. Previous studies have categorized RDF-stores in different ways. In this section, we include the common categories and review the common approaches of each category. It would highlight the main characteristics of the different approaches that belonged to each category.

2.5.2 Classification of Centralized RDF-stores

We start with studies like [18, 86] that categorized RDF-stores into: "*non-native RDF-stores*" and "*native RDF-stores*". These categorizes are defined as follows:

- **Non-native RDF-stores.** These are defined as solutions that make use of existing database systems like the relational database systems for storing RDF datasets and executing queries over them.
- **Native RDF-stores.** In contrast, these are defined as approaches that are not using existing database systems to store RDF data. These approaches implement their own storage strategies specific to the RDF data model and extensively use different indexing techniques to efficiently execute queries.

Non-native RDF-stores are built on top of existing systems (relational, document-store, etc.) for storing RDF data. In contrast, native RDF-stores are customized systems for the storage and retrieval of triples and are built from scratch. Previous research such as [80, 87] has argued that the structure of RDF data is the most important point to be considered for *effectively* storing RDF data and optimizing SPARQL query processing. Based on this, they have suggested the following two classes for categorizing RDF-stores: "*structure-aware RDF-stores*" and "*non-structure-aware RDF-stores*". These classes are defined as follows:

- **Structure-aware RDF-stores.** These are defined as solutions that leverage "*structure information*" derived from an input RDF data (like the set of correlated predicates) to store it and then use this information for the efficiency of their SPARQL query optimization.
- **Non-structure-aware RDF-stores.** In contrast, these are defined as approaches that do not exploit any structure information on the input RDF datasets for storing the data or executing queries against it.

As mentioned above, [80, 87] advocated that understanding the structure of RDF data plays a crucial role in classifying existing RDF-stores. Others have argued (such as [23, 88]) that RDF-stores are better to be classified based on their choice of physical design. Accordingly, they categorized existing RDF-stores into the following classes: "*systems with workload-aware physical layout*" and "*systems with workload-oblivious physical layout*". These classes are defined as follows [23, 88]:

- **Workload-aware RDF-stores.** These are defined as solutions that utilize "*workload-driven techniques*" for computing their physical layouts based on information in users' queries (workload). In this class, RDF-stores' physical designs are not "*fixed*" and would be updated (or adapted) as the incoming workload changes.

- **Workload-oblivious RDF-stores.** In contrast, workload-oblivious RDF-stores rely on designs with fixed physical representation without dynamically updating their physical representation or switching to a better one at runtime as the workload changes.

Table 2.2 outlines various classifications of RDF-stores (along with their respective definitions) that were discussed in this section. Having covered this, we now review state-of-the-art RDF-stores in the next sections. In our review, we follow [18, 86] and divide major existing RDF-stores into two classes: centralized non-native and native RDF-stores. We also highlight some common distributed approaches in the context of RDF data management systems.

Source	Classes & Definitions
[18, 86]	• **Non-native RDF-stores.** Solutions that make use of existing database systems like the relational database systems for storing RDF datasets and executing queries over them. • **Native RDF-stores.** In contrast, these approaches implement their own storage strategies specific to the RDF data model and extensively use different indexing techniques to efficiently execute queries.
[80, 87]	• **Structure-aware RDF-stores.** Solutions that leverage "*structure information*" derived from an input RDF data (like the set of correlated predicates) to store it and then use this information for the efficiency of their SPARQL query optimization. • **Non-structure-aware RDF-stores.** In contrast, these are approaches that do not exploit any structure information on the input RDF datasets for storing the data or executing queries against it.
[23, 88]	• **Workload-aware RDF-stores.** Solutions that utilize "*workload-driven techniques*" for computing their physical layouts based on information in users' queries (workload). It is mentioned that RDF-stores whose physical designs are not "*fix*" and would be updated (or adapted) as the incoming workload change, are belong to this class. • **Workload-oblivious RDF-stores.** In contrast, these rely on designs with fixed physical representation without dynamically updating their physical representation or switching to a better one at runtime as the workload changes.

Table 2.2: Various classifications of RDF-stores

2.5.3 Major Non-native RDF-stores

2.5.3.1 Overview

A non-native RDF-store is typically built on top of an existing Data Management Systems (DMS) (e.g., relational, document-store, etc.) for storing RDF data like [46, 48, 89, 90]. In practice, most of these systems make use of relational systems to store RDF data. This allows them to widely apply techniques of storing and indexing data originally developed for relational systems to the RDF data model (with some potential specialization if needed). It is very common for relational-based non-native RDF-store to translate SPARQL queries to SQL and then execute them. Leveraging similarities between SPARQL and SQL makes this translation feasible in most cases. In practice, most, if not all, major publicly-accessible relational-based non-native RDF-stores employ the mapping of RDF datasets into relational tables following one of the following ways:

- **Statement Table.** This approach is a "*straightforward*" way to store RDF triples by maintaining the input RDF data as a "*linearized list of triples*" [91]. This approach stores RDF triples as ternary tuples. In other words, all triples are stored into a triple table with three attributes as follows: subject, predicate, and object. These attributes correspond to the three elements of each RDF triple. This approach is referred to as a "*generic approach*" in [35] and as the "*triple stores*" approach in [81, 92]. It is not even uncommon to refer to it as the *single giant table* approach. An additional attribute column can also be added to store the *named graph* (see Section 2.2.2.2) that contains triples.

- **Property Tables.** In general, using a single giant statement table to store RDF triples can introduce a number of disadvantages, especially when it comes to the query evaluation process. For instance, the need for a large number of self-joins is inevitable to evaluating many queries. This can negatively affect queries' execution times. Employing the *property tables*

approach is proposed to alleviate this problem [39, 91, 93]. In particular, the core idea behind this approach is to group triples by predicate names and store all triples with the same predicate name in a separate table. This approach can be implemented within a column-store as discussed in [90]. This approach is referred to as "*vertical partitioning*" in [81, 92], as "*group-by-predicates*" in [88], and as "*schema-specific*" solutions based on a number of property tables in [94].

- **Cluster-property Tables.** As discussed in [95], the property tables approach still can suffer from an excessive number of joins between tables in many cases. To address this, *cluster-property tables* is proposed [96]. This approach instead of employing a large number of tables for storing triples based on their predicate names, groups triples into classes based on the occurrence of sets of predicates in the dataset. The classes are typically defined by clustering algorithms (and sometimes application experts).

Relational-based non-native RDF-stores attempt to derive structural information from input RDF datasets to store them (and then execute queries over them). To this end, RDF-stores typically follow one of the above-mentioned approaches.

Given the success of *NoSQL* systems in many domains (like cache management in large-scale web-based applications) in recent years, a limited number of academics publications has attempted to develop NoSQL-based RDF-stores such as [97, 98, 99, 100, 101]. These are mainly designed for cloud-based distributed use cases [102]. To the best of our knowledge, no major centralized NoSQL-based RDF-store is widely in use at this time. Some academic prototypes such as [103] have shown the efficacy of document-stores in similar contexts. However, we focus primarily on some of the major non-native RDF-stores that are widely in use.

Literals

Hash	Model
int64	Full Text

Models

Hash	Model
int64	Full Text

URIs

Hash	Model
int64	Full Text

Model	S	P	O	Inferences	Literals
int64	int64	int64	int64	boolean	boolean

Figure 2.9: The logical database design of 3store [38]

2.5.3.2 Statement Table-based

3store [38] is an example of a non-native RDF-store. This follows the statement table approach and stores the RDF triples into a single relational table. The logical database design of 3store is shown in Fig. 2.9. This shows that 3store uses three extra tables to store hash values. It also added 3 columns to store flags to indicate whether the object of a triple is a literal or a URI. 3store stores a hash of the elements of each triple (i.e., URIs and literal values) into the statement table to minimize the storage space needed. This also allows it to ensure that all rows in the table have the same size on the disk. Since 3store used the same hashing function for both URIs and literals, the statement table needs to contain flags to indicate whether the object of a triple is a literal or a URI.

4store [104] has proposed another relational-based system in which RDF is stored as quads (i.e., subject, predicate, object, and model). Employing the additional attribute "*model*" allows 4store to support *named graphs* (see Section 2.2.2.2) while querying datasets. If a query needs to be executed over the whole RDF dataset, 4store can ignore this attribute. The use of the "model" attribute is not limited to representing the named graphs. In general, the semantic information from RDF data can be exploited so that additional data can be annotated per triple and stored as a fourth element (i.e., the model attribute) for each input triple.

Virtuoso [105] is another relational-based system that uses a single table and stores RDF data as quads (i.e., subject, predi-

cate, object, and graph). Similar to 4store, employing the additional attribute "*graph*" allows Virtuoso to support *named graphs* (see Section 2.2.2.2). Although Virtuoso is a relational-based system, it is sometimes referred to as "*Virtuoso native RDF-store*" [80]. More specifically, Virtuoso can be viewed as a traditional relational database with enhanced RDF support [91] since its RDF support is indeed implemented and stored entirely within the *Virtuoso's SQL database system* [80].

RDFMATCH [106] has been proposed by *Oracle* as an "*SQL-based table function*" to execute queries over RDF data. In this work, RDF triples are typically stored in a single table using the Oracle conventional relational DMS. Traditional SQL queries can be executed over them. In addition, RDFMATCH is a function that can be integrated with each SQL query to query RDF triples. This function supports a SPARQL-like syntax.

2.5.3.3 Property Tables-based

Jena [107] is an example of a non-native RDF-store that follows this approach to store RDF data [91]. More specifically, Jena has two implementations: Jena SDB and Jena TDB. Jena SDB[9] uses conventional relational databases like MySQL and PostgreSQL for the storage and execution of queries. Jena SDB can be categorized as a non-native RDF-store. As of June 2013, Jena SDB has not been developed. However, it is now in the "*maintenance only*" status which means its developers intend to continue releasing Jena SDB but it is not actively developed.[10] As discussed in [80], Jena can support the property table approach in which it groups triples by their types. All triples of the same type are stored in a separate table. However, the database schema of each property table generally needs to be defined by the application. In addition, multi-valued predicates need to be specified by the database developer before storing them separately in new tables [80].

[9] https://jena.apache.org/documentation/sdb/
[10] https://jena.apache.org/documentation/sdb/sdb_index.html

Sesame [39] is designed as an open-source framework for storing RDF data. Similar to Jena, it allows the use of different storage engines (including relational databases like PostgreSQL) as its backends. As mentioned in [80], since May 2016, Sesame has officially forked into an Eclipse project (and now referred to as RDF4J). Sesame can support the use of property table storage. However, similar to Jena, the actual schema and physical design are needed to be tuned by the application experts. Sesame can derive the table definitions automatically if the ontology of a dataset is available.

RStar [42] is originally implemented to store ontology information along with the RDF data (i.e., instance-level data) using multiple relational tables. In particular, RStar employs five tables to store ontology information, namely, class, subclass, property, sub-property, and property-class. An "*InstanceOfClass*" table is then employed to store RDF triples (i.e., instances of the classes). In RStar, a single triples table (with three attributes: SubID, PreID, and ObjID) is also employed to store all instance triples. RStar also maintains links between ontology and instance data.

DLDB [43], DBOWL [108], and RDFSuite [36] are non-native RDF-stores that use the ontology class structure for storing RDF datasets in relational systems. These systems are very similar to the above-mentioned RStar. More specifically, DLDB uses the definition of classes and properties in ontology information to create tables. This can be viewed as the "*hybrid*" of the property table and the vertical partitioning approaches [80]. DBOWL uses *axioms* in the ontology information to create relational tables. RDFSuite uses RDFS (see Section 2.2.1) to create relational tables for storing RDF data. In addition to the storage and query processing, it has provided a suite of tools for RDF validation. Using ontology information can assist in designing schema for relational-based RDF data management. However, this information tends to be partially available in reality (if not totally absent). For instance, only a small percentage (i.e., approximately 30%) of ontology class properties are available for datasets of the LOD project [80].

SW-Store has been proposed by Abadi et al. [90] to store RDF

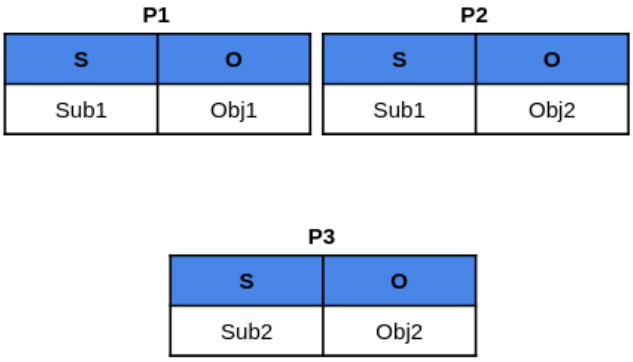


Figure 2.10: An example illustration of vertical partitioning [90]

datasets following the vertical partitioning approach. In this approach, a fully decomposed storage model is employed to store triples. More specifically, SW-Store first creates n tables where n is the number of unique predicates in the RDF dataset. Each table has two columns to store subjects and objects that are associated with each predicate. Fig. 2.10 shows an example illustration of the vertical partitioning proposed by SW-Store [90, 91] where for each existing predicate one subject-object table is created. As can be seen, triples are decomposed into multiple binary tables with two columns (S, O) and each binary table is corresponding to a unique property (e.g., "*P1*"). This approach was implemented based on a column-oriented DMS (i.e., C-Store [109]). Similar to Virtuoso, SW-Store translates SPARQL queries to their equivalent SQL and then uses the C-Store query processing engine to execute them. An advantage of employing the vertical partitioning approach is that creating the binary tables does not need any a priori schema design [91]. However, the high number of resulting joins imposes restrictions on the performance of systems following the vertical partitioning approach [95].

2.5.3.4 Cluster-Property Tables

Fig. 2.11 shows an example illustration of the clustered-property tables in which frequently co-accessed attributes are stored together [91].

Figure 2.11: An example illustration of clustered-property tables [91]

Sintek and Kiesel [96] followed the cluster-property tables approach and proposed RDFBroker. The suggestion of this work is to exploit structure information from RDF datasets like predicate-object pairs of each subject to automatically group subjects and create a table for each group. However, this strategy can lead to the creation of many tables for diverse datasets. Each of these tables typically contains only a small number of rows. To address this, the merging of small tables into larger ones can be considered (by using association rule mining methods to find all predicates that frequently occur with the same subject). The core idea behind using clustered-property tables is to group commonly accessed subjects together in a single table to avoid expensive joins on the data.

2.5.3.5 Summary

The high number of resulting joins imposes restrictions on the performance of relational-based RDF-stores [95]. Besides, a major drawback of many relational-based approaches (like property and cluster-property approaches) is that these approaches do not support queries with unbounded properties [81, 92].

Table 2.3 has summarized the storage layouts and supported features (e.g., schema inference and update support) of centralized non-native RDF-stores that we have reviewed. Having covered the non-native RDF-stores, we now provide background information on some of the native approaches, in the next section.

RDF-store	Storage Layout	Schema Inference	Update Support
Jena	Property tables	✓	✓
Sesame	Property tables	✓	✓
3store	Statement Table	✓	
4store	Statement Table	✓	
Virtuoso	Statement Table	✓	✓
SW-Store	Property tables (vertical partitioning)		
RDFBroker	Auto-detected Cluster-property tables		
RStore	Ontology-based Property tables		✓
DLDB	Ontology-based Property tables		✓
DBOWL	Ontology-based Property tables		✓
RDFSuit	Ontology-based Property tables	✓	✓

Table 2.3: Centralized non-native RDF-stores

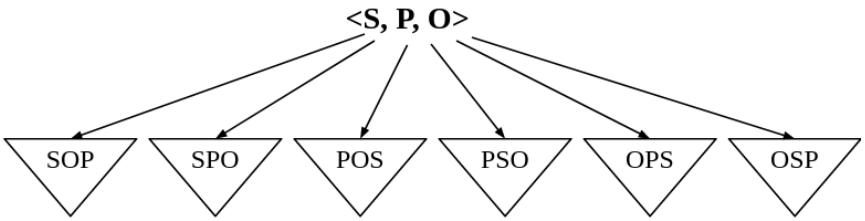


Figure 2.12: Indexing RDF triples in all six possible orders

2.5.4 Major Native RDF-stores

2.5.4.1 Overview

RDF can represent diverse KG content with different levels of *structuredness* (see Section 2.2.3) without any difficulty. Thanks to this inherent flexibility, many data management system designers have eschewed mapping RDF data into generic systems (like relational DMSs) and focused on designing customized systems for the storage and retrieval of RDF data. This is indeed consistent with the principle and rationale advocated in [59] that customized data management systems can typically outperform generic ones. In the context of RDF data management, these customized systems are typically referred to as native RDF-stores that are built from scratch specific to the schema-relaxability feature of the RDF data model. We review major existing native RDF-stores in this section. We divide them based on their primary storage locations into two groups: disk-based and memory-based (e.g., all the data and indexes are stored on disk in the disk-based systems).

2.5.4.2 Aggressive Indexing Strategy

Native RDF-stores are typically designed based on a "*workload-independent*" or "*index-everything*" (sometimes referred to as "*aggressive indexing strategy*") storage scheme specific to the schema-relaxability feature of RDF data model. Over time, a range of native RDF-stores for storing, indexing, and querying RDF have been developed. These approaches typically maintain a set of six indexes (aka, exhaustive indexing) covering all possible triple orders. As shown in Fig. 2.12,

six possible indexing orders are PSO, POS, SPO, SOP, OPS, and OSP where P stands for the predicate, O for the object, and S for the subject (referring to three elements of each RDF triple). This exhaustive indexing may result in high space consumption. However, it is quite practical in the case of RDF data [44, 47, 60, 91, 110].

2.5.4.3 Dictionary Encoding

In addition to following the exhaustive indexing strategy, employing dictionary techniques to map literal values and IRIs of an RDF dataset to unique numeric object identifiers ("*ids*") is a typical design choice of most, if not all, major RDF-stores (especially in native systems). This dictionary-based encoding can lead to a significant reduction in storage since RDF datasets typically contain many frequently repeated IRIs and literal values. Two common approaches to generate "*ids*" are as follows: hash-based approaches (i.e., using a hash-function) or counter-based approaches (i.e., maintaining a counter and increasing the counter for each new RDF element). Dictionaries can be implemented using different data structures but utilizing B-trees and sorted vectors are perhaps more prevalent [44, 80]. It is not uncommon to apply various compression and optimization techniques to mapping dictionaries such as the storage separation of common namespace prefixes and IRIs. More information about various dictionary encoding techniques in the context of RDF can be found in [111]. Having covered exhaustive indexing and mapping dictionaries as two typical design choices of almost all native RDF-stores, we review major existing native RDF-stores in the following.

2.5.4.4 In-memory Approaches

As mentioned in the previous section, native RDF-stores typically follow exhaustive indexing strategies to store RDF data and efficiently execute queries over them. One of the systems that followed this approach is Yars [112]. This system has adopted information retrieval techniques (e.g., inverted indexes) and combined them with database techniques to create two sets of indexes, namely, *lexicon*

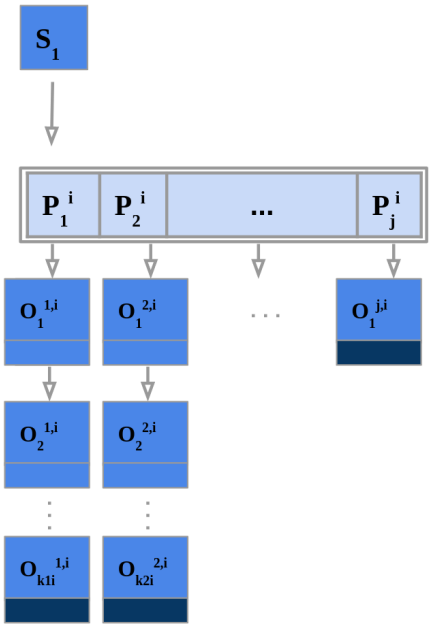


Figure 2.13: An example illustration of SPO index in Hexastore [60]

and *quad indexes* to store RDF data. Specifically, the *lexicon* index is to cover the string representations of RDF data. This is an inverted index enabling Yars to perform full-text searches. Its quad indexes are created to efficiently store RDF data as quads of <S, P, O, C> where S, P, and O are to represent RDF triples and C can be used to refer to any application-specific metadata like the origin of each RDF triple. These quad indexes are implemented using the B+tree data structure. These indexes are created over the following orders of the quad: SPOC, POC, OCS, CSP, CP, and OS. Yars2 [113] has been developed as an extension to Yars [112] in which B+tree indexes are replaced by alternative indexing data structures like hash tables to decrease the I/O cost. More specifically, Yars2 typically consumes more storage space by maintaining 16 hash tables for exhaustive indexes over quads but its search operation tends to be faster [80].

Similar to Yars, Hexastore [60] has also followed the exhaustive indexing approach. In Hexastore, RDF data is indexed in all six possible orders. This system has utilized a combination of two sorted

vectors and a list to store each index order. For instance, Fig. 2.13 shows the SPO index order created by Hexastore. In the SPO ordering index, every distinct subject is associated with the vector of predicates that occur in one triple with that subject, and every predicate, in turn, is appended with the list of objects. To reduce the storage consumption, Hexastore also employs a dictionary technique to map URIs to ids. Hexastore can support single triple pattern lookups and merge-joins of any pair of triple patterns. Query performance was the first design priority of Hexastore. Thus, insert operations tend to be slower. Hexastore was originally proposed as an in-memory system. More recently, an on-disk version of it was also introduced in [114]. Experimental evaluations show that employing the sorted vector-based scheme can lead to faster query execution times (in most cases) compared to B+trees.

Similar to Hexastore, BitMat [115] is an in-memory native RDF-store utilizing a "*bit-matrix*" structure for representing RDF triples. Reducing space consumption was one of the main design priorities of BitMat. To achieve this, BitMat considered each RDF triple as a 3-dimensional entity that can be horizontally partitioned into multiple fragments based on the usage requirements. Conceptually, BitMat is a 3-dimensional "*bit-cube*" in which each cell is a bit to represent the absence or presence of a unique triple. To store this bit-cube in memory, it has been flattened into a 2-dimensional bit matrix. To mitigate the potential excessive space consumption, BitMat only maintains bits representing the presence of triples. To this end, it utilized an array of bit-rows, where each row is a collection of all the triples having the same subject. This enhancement led to a compact in-memory representation of RDF data [86]. Queries can then be processed using bitwise operations (like "*AND*" and "*OR*") on the BitMat rows. No update mechanism has been involved in the BitMat design. Thus, this is mainly a read-only RDF triple storage system with no dynamic insertion or deletion (of RDF triples) support at present.

Three-way Triple Tree (TripleT) has been proposed in [116]. The main design goal of this system was to decrease the negative effects

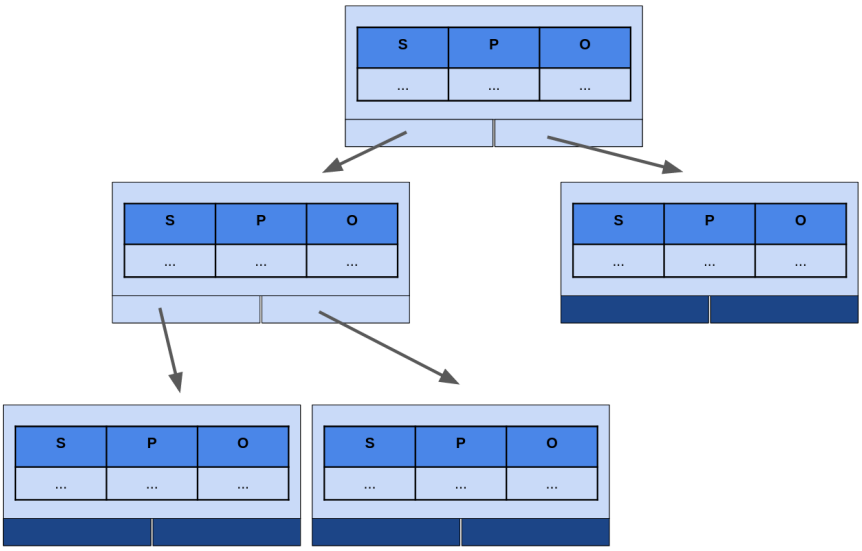


Figure 2.14: An example illustration of the Kowari system's AVLtree-based indexing [117]

of weak data locality on the storage of RDF data. The weak data locality in this context refers to the design of a multi-index approach where a piece of data can appear in multiple memory locations and several different data structures. In other words, they reported that creating multiple indexes and storing each of them separately can potentially lead to weaker data locality. TripleT still follows the exhaustive indexing strategy but in contrast to common approaches, TripleT creates a single B-tree index over all the "*atoms*" occurring in an RDF dataset instead of building indexes over triples. More specifically, TripleT first extracts elements of RDF triples and stores them separately using three "*buckets*", namely, S-bucket, P-bucket, and O-bucket, referring to storage locations of subjects, predicates, and objects, respectively. Each key atom in the B-tree then points to the actual data stored in the buckets with regard to the role of the atom in the RDF triples (i.e., an atom can be either a subject, a predicate, or an object). This B-tree finally is used as the access path when it comes to query processing. As discussed in [80], the advantages of the TripleT design (as compared to conventional RDF indexes like SOP, PSO, and OSP) are somehow unclear.

2.5.4.5 Disk-based Approaches

The Kowari system [117] (also referred to as Mulgara) is another native system that followed the typical approach of exhaustive indexing to store RDF data. This system creates indexes over the following six different orders SPOM, POSM, OSPM, MSPO, MPOS, MOSP where S, P, and O are to represent RDF triples and M can be used to refer to any application-specific metadata like named graphs. The Kowari system has implemented these indexes using multi-version blocked AVLtrees. In cases where there is no need to store any metadata, Kowari ignores the M and creates an index over the following three orders: SPO, POS, OSP (see Fig. 2.14 for an example illustration).

RDF-3X [44] as a native RDF-store has eliminated the need for physical database design by creating exhaustive indexes over all permutations of RDF triples. RDF-3X uses a giant single table to store triples. This is not based on a relational system since RDF-3X implemented its own storage engine. In fact, all processing is index-only in RDF-3X. Thus, the triples table exists merely virtually. More specifically, RDF-3X has utilized a compressed clustered B+tree to store each index order in which RDF triples are sorted lexicographically. A mapping dictionary technique is also employed to replace long string literals in the triples with ids. It supports updates following a staged strategy.

iStore [118] has introduced the notion of "*structure index*" which can be used for storing RDF data and executing queries over it. This can lead to clustering of those subjects of the RDF data that are similar in structure. The similarity here refers to the "neighborhood" of the graph vertexes and the "structure" refers to the set of incoming and outgoing connections of each vertex. iStore uses this "structure index" as an access path to perform join processing.

BlazeGraph[11] (formerly BigData) has been proposed as a native RDF-store supporting RDF/SPARQL Sesame APIs, the Apache TinkerPop stack, and graph mining API. Blazegraph's data modeling is based on B+trees to store RDF triples in the form of ordered

[11] https://blazegraph.com/

data. Blazegraph typically creates the following indexes for triples modes: SPO, POS, and OSP. For normal use cases, these indexes are laid out on variable-sized pages. These index pages are read from the backing store and loaded into the main memory on demand (i.e., into the Java heap). However, Blazegraph takes advantage of a variety of data structures to execute queries when stored RDF data is loaded in the main memory. For example, the underlying data model (i.e., B+trees) is retained by a mixture of a ring buffer (hard reference queue), weak references, and hard references on the stack during the use along with a native memory cache for buffering writes to reduce write application effects. Blazegraph also implements some advanced query optimization techniques such as runtime query optimization and vectorized query engine. It is alleged that Blazegraph was acquihired by Amazon and the Amazon Neptune is based on Blazegraph.

2.5.4.6 Summary

We have provided background information on the main approaches that are followed by native RDF-stores in this section. We also reviewed major existing systems. The evolution of native systems is shown in Fig. 2.15 (based on [86]), in which the edges indicate influences. Table 2.4 has summarized the storage layouts and supported features (e.g., data structure and update support) of centralized native RDF-stores. In the next section, we describe how modern RDF-stores perform SPARQL query processing. We concentrate on query processing in the context of native RDF-stores that follow the index-everything strategy.

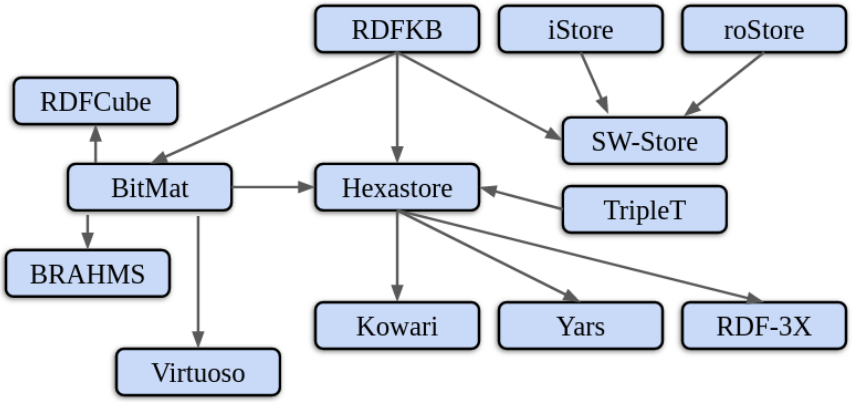


Figure 2.15: Evolution of native RDF-stores [86]. Edges indicate influences.

RDF-store	Storage Support	Index Data Structure	Update Support	Primary Storage
Yars	Quad	B+trees	✓	In-memory
Yars2	Quad	Hash Tables	✓	Hybrid
Hexastore	Triple	Sorted Vectors and Lists		In-memory
BitMat	Triple	Bit Arrays		In-memory
TripleT	Triple	Btree		In-memory
Kowari	Quad	AVLtrees		Disk-based
RDF-3X	Triple	Clustered B+trees	✓	Disk-based
iStore	Triple	N/A (is not presented in [118])		Disk-based
Blazegraph	Quad	B+trees	✓	Disk-based

Table 2.4: Centralized native RDF-stores

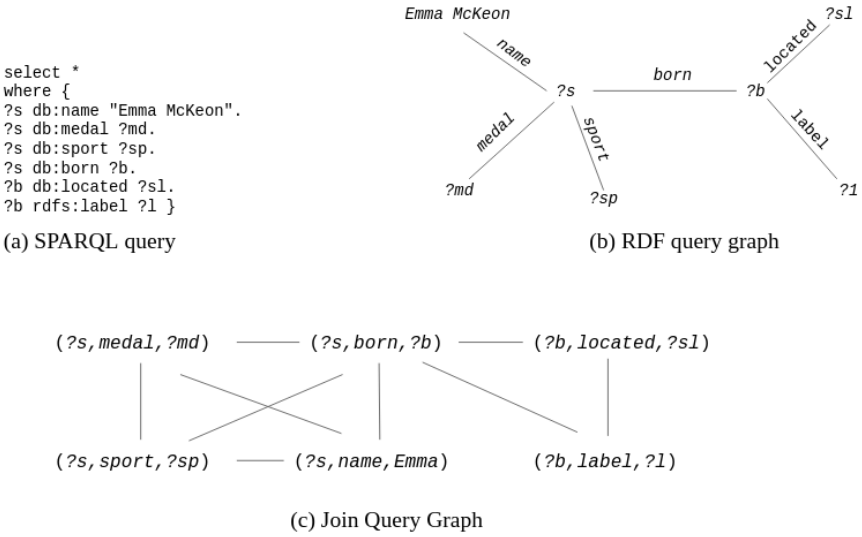
2.6 SPARQL Query Optimization

2.6.1 Overview

We proceed by describing different steps of SPARQL query optimization and processing in this section. When an RDF dataset is loaded, major RDF-stores typically perform the following steps to execute a given query. In general, the RDF-store parses a query according to the query language grammar (i.e., SPARQL) and generates the corresponding syntax tree for the query. Transforming the syntax tree into a logical operator graph is the next phase. This is followed by logical operator graph optimization. For example, the execution order of different parts of a query such as filtering or sorting should be planned. The next phase is to choose the best implementations for each logical operator referred to as physical optimization. The output of this phase is the physical operator graph which should be executed and the resultset will be returned [52].

2.6.2 Join Query Graph Construction

Given a SPARQL query, the query engine parses it according to the query language grammar and generates the corresponding syntax tree for the query. If the query has only one triple pattern, it will be executed through an index lookup (depending on the query pattern, a proper index order will be used) and the result will be returned. However, SPARQL queries typically contain more than one triple pattern. In this case, the query engine constructs a representation of the query called the "*join query graph*". This is constructed based on the decomposition of the given query to its triple patterns. Every triple pattern of the given query consists of literals and variables. Literals typically need to be mapped to their ids if dictionary encoding techniques are used for triple representation. After this, the triple patterns are represented as the nodes of the join query graph. In this graph, triple patterns that share (at least) one variable are connected with an edge. Edges in the query graph correspond to the



(a) SPARQL query

(b) RDF query graph

(c) Join Query Graph

Figure 2.16: Query graphs of an example SPARQL query (based on [81, 92])

join possibilities within this query. Each query can also be represented with a dual graph structure called an "*RDF subgraph*". This subgraph describes the pattern that has to be matched against the underlying RDF dataset. The RDF subgraph will also be used as a basis for further query optimization like join ordering, etc.

Fig. 2.16 (based on [81, 92]) shows a typical SPARQL query with several *star-shaped* subqueries (see Section 2.4.2) connected via chains (see Section 2.4.3). Star-shaped queries correspond to entities whose predicates were specified by the predicates on the edges that form the star. For instance, star-shaped subqueries are formed around the variables *?s* and *?b* in Fig. 2.16 to describe a person and a city, respectively. When the query graph is constructed, RDF-stores can use it as a basis to perform query optimization. To achieve this, RDF-stores construct an algebraic representation (containing join operators) for the query graph called the "*join tree*". The join tree can be viewed as a binary tree whose leaf nodes are the query's triple patterns and the inner nodes are joins. This tree can help the query optimizer to construct the query's execution plan in which indexes and types of joins are determined. An execution plan con-

sists of all essential steps to clearly determine how the query will be executed. For instance, an example of an execution plan (perhaps un-optimized) is given below:

1. Perform an index scan for every triple pattern (use literals and their positions to determine the range of the scan and the appropriate index order).
2. Look at the query graph and add a join for each edge
   - If multiple edges are associated with two nodes, turn them into selections.
   - If the query graph is disconnected, no join is needed. Instead, add cross products between them (create a single join tree).
3. Check whether any FILTER clause exists in the query.
4. Check whether the DISTINCT modifier is used in the query.
5. Add the id-to-string mapping on top of the plan (in case the dictionary encoding is implemented in the system).

An execution plan can further be optimized by applying techniques like splitting the FILTER condition into the conjunction of conditions, pushing filters as deep as possible towards the leaves in the join tree, etc. However, the most challenging part of optimizing an execution plan is to order joins [92]. An optimized join ordering can lead to minimizing the amount of intermediate results transferred from operator to operator while executing a query. In the next section, we describe the join ordering process.

2.6.3 Join Ordering and Plan Enumerations

As mentioned above, queries typically contain sets of triple patterns requiring pattern matching. For any given query with multiple triple patterns, there exist multiple join trees. A query optimizer has to select the *best* one with regard to a certain cost function. Typically,

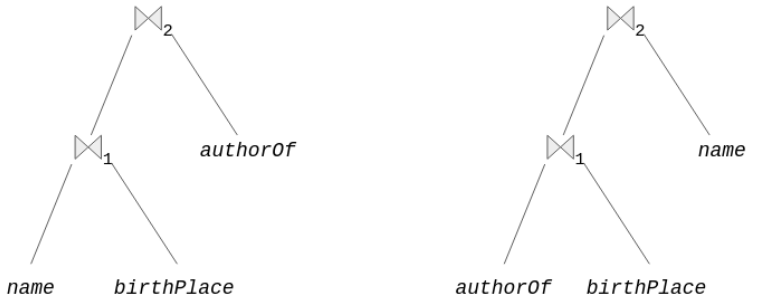
the query optimizer estimates the execution cost by evaluating the join tree in terms of the cardinality of triple patterns and cardinality of intermediate results. The cardinality of a triple pattern is defined as the number of triples that match the pattern. It is not very hard to predict that the higher the number of joins, the higher the number of possible orderings which can make the index choice and plan enumerations difficult. The number and complexity of query features would make the search space larger in order to find an efficient plan. For instance, one of the difficulties in executing queries with conjunctions or disjunctions such as union and optional patterns is that query engines need to generate plans with all possible combinations. However, the core problem of SPARQL query optimization is still finding the least-cost join ordering. As discussed in [44, 47, 81, 92], this requires the query optimizer to take the three following common cases into consideration:

- Assigning higher priority to the fast generation of optimal plans for star-shaped subqueries since this is a very common query type.
- In principle, there are two important classes of join trees: (i) left-deep trees (sometimes referred to as linear trees) and (ii) bushy trees. Building bushy trees as join plans for the star-shaped subqueries that are connected with long chains can lead to more optimal plan generation.
- Selecting appropriate index orders is important for exhibiting better performance.

The performance of RDF-stores relies on finding the least-cost execution plan where estimated cardinalities have a significant impact on the optimal plan generation. For instance, an example of a star-shaped query is shown in Fig. 2.17. Inappropriate join ordering can lead to the plan that is depicted in Fig. 2.18 (a). This is a suboptimal plan. The optimal plan is shown in Fig. 2.18 (b). This example is used to show how easily a query optimizer may make mistakes and generate suboptimal plans.

```
1    PREFIX dbpr: <http://dbpedia.org/property/>
2    PREFIX foaf: <http://xmlns.com/foaf/0.1/>
3    SELECT ?author ?book
4    WHERE {
5        ?author foaf:name       ?fullName .
6        ?author dbpr:birthPlace ?Bp .
7        ?author dbpr:authorOf   ?book
8    }
```

Figure 2.17: An example SPARQL query (based on [81, 92])



(a) Suboptimal Plan: $|\bowtie_1|$ = 200Mln (b) Optimal Plan: $|\bowtie_1|$ = 1.7Mln

Figure 2.18: Optimal and suboptimal query plans for the star-shaped query in Fig. 2.17 (based on [81, 92])

In theory, sampling-based query execution may help in these cases, but it is not very practical to follow a sampling-based plan enumeration (or a randomized selection) to find the optimal plan since for many queries, the sample would have to be large to be useful, which is obviously very expensive. As a result, it is not very easy for RDF-stores to generate optimal plans for queries with multiple joins as well as optional patterns and unions.

Having covered the key features of query optimization and processing in modern RDF-stores, we now provide background information on cardinality estimation in the next section.

2.6.4 Cardinality Estimation

As mentioned in the previous section, the good performance of RDF-stores relies on finding the least-cost execution plan where estimated

cardinalities have a significant impact on the optimal plan generation. However, it is already recognized in the database community that obtaining accurate statistics (or even reasonable estimations) is a standard yet extremely complicated problem.

Building histograms to compute frequencies of RDF elements is a common approach to estimating cardinalities. In fact, cardinality estimation of single triples using these histograms is not very complicated in most cases. However, common SPARQL queries contain more than a single triple pattern. This requires estimating the join sizes of two and more triple patterns (depending on the number of triple patterns in a given query) to select the most efficient join ordering. This clearly calls for estimating correlations between them which is not an easy task to accomplish (in most cases) due to the schema-relaxability of the RDF data model. For instance, computing cardinalities for datasets with lower levels of structuredness is challenging since triples are more heterogeneous (i.e., datasets with a larger diversity of predicate names) and the common data structures are not compact enough to group all correlated triples. In other words, the schema-relaxability of RDF fundamentally complicates the query optimizers' efforts for deriving reliable estimates [44, 81, 92].

2.7 Virtual Integration of RDF-stores

2.7.1 Overview

Executing queries over multiple RDF-stores is not uncommon especially when several RDF datasets are strongly related to a domain of interest but physically are stored in different systems. When RDF datasets are separately stored, one can integrate (materialize) them all into a single dataset and employ a single RDF-store to store it and execute queries over it. There is an alternative option as well. This is to "*virtually*" integrate them in which the RDF datasets are physically separated but any given query will be executed against all of them and the results will be materialized before being returned to the user. In practice, there are two common ways that preserve the data sources' autonomy but at the same time allow for evaluating queries based on the data of multiple sources, namely, mediator-based systems and federated systems [94]. These systems are explained in the next sections.

2.7.2 Mediator-based Systems

These systems provide a service to virtually integrate RDF datasets from a selection of independent RDF-stores. In a mediator-based approach, the RDF-stores are often unaware that they are participating in an integration system but there is a mediator that provides a common interface for the user to receive queries and return the results. The mediator receives a query and sends it to all of the underlying RDF-stores and when the results are returned, the mediator materializes them before sending them to the user. The mediator performs this by using a "global catalog" (statistics about the RDF-stores and data available at the sources). The mediator typically provides some additional services like rewriting and optimizing the query. To do this, the mediator needs to determine which RDF-stores are relevant with respect to a given query and create subqueries for the relevant RDF-store. In case the RDF-stores manage their local RDF dataset using any data models or query languages different from what the

mediator uses (e.g., SQL to XQuery), rewriting the subqueries in a way that makes them "understandable" for the source RDF-stores is needed. This is sometimes referred to as "*wrapping*" [94]. Transforming the returned results back into the mediator's model then needs to be performed.

2.7.3 Federated Systems

Similar to the mediator-based systems, federated systems also provide a service to virtually integrate RDF datasets from a selection of independent RDF-stores. Federated systems can execute SPARQL queries over multiple RDF-stores. A typical example is linked data (like the LOD project), where different RDF datasets are interconnected.

The typical approach for federated SPARQL query processing is first to *precompute metadata* for every single RDF-store that is a member of the federation. This metadata can assist in decomposing any given SPARQL query into several subqueries. Each subquery is then sent to its relevant SPARQL endpoints (the storage location). The results of all subqueries are finally merged together as the answer of the given query [26].

As the second variant of virtual integration, a federation is a consolidation of multiple RDF-stores offering a common interface and thus similar to the mediator-based systems. However, the main distinction between them is as follows: in the federated systems, RDF-stores are aware that they are part of the federation and they actively support the data model and the query language that the federation agreed upon. Ideally, there should be no difference between these two architectures from a user's point of view as long as both offer transparent access to the data.

Having described the two common approaches for virtually integrating RDF-stores, we now provide background information on multi-database solutions for query processing in the next section.

2.8 Polygloty: A Multi-database Solution for Query Processing

Several influential writers in the practitioner community, in particular, have converged on an interpretation of polygloty that suggests using different DMSs and physical data stores depending on the type of application. For instance, Fowler [119] has suggested the use of traditional RDBMS for financial data, document stores (e.g., MongoDB) for product catalog data, Key/Value stores (e.g., Cassandra) for use activity logs, Property graph stores (e.g., Neo4j) for recommendation systems, and in-memory Key/Value stores (e.g., Redis) for user sessions data processing, etc. This approach violates the basic principle of integrated data that is central to the database approach since the early 1970s, despite the exigencies of immediate commercial imperatives. It is not also difficult to see that the lack of integration across the entire data will lead to balkanized data islands.

Supporting multiple data models against a single, integrated backend can potentially address this problem [120]. As a result, over the last few years, there has been growing interest in employing multiple DMSs for query processing. This interest has manifested in the research and development of some open-source platforms such as Apache Beam[12] and Drill[13] as well as some academic prototypes [121]. In general, these proposals utilize a model consisting of multiple DMSs but require input from expert users to decide which specific DMS meets the requirement for a given application or query set. For example, [121] presents two commands, namely `scope` and `cast` which provide a user with information to select the most appropriate DMS for the query being analyzed. Recent works [122, 123, 124] present parallel cross-platform data processing systems to decouple application interaction from underlying platforms. These systems follow a process that splits each given query into subqueries, executing them on multiple platforms simultaneously to minimize the overall runtime. Although providing a speedup, it is unclear,

[12] Available Online: https://beam.apache.org
[13] Available Online: https://drill.apache.org

however, how much of the performance gain comes from minimizing inter-platform communication overheads by taking advantage of data locality for sub-query processing. Various proposals take alternative approaches, presenting cross-platform *stream processing* [125] or building dynamic workload management through adaptable architecture design [126, 127]. There are some academic prototypes as well to improve the performance differences between relational and multi-language user-defined functions (UDFs), e.g., [128].

These, as well as other similar examples, show that research and experimentation have been carried out by utilizing multiple DMSs for query processing, however, current solutions are heavily focused on applications such as *ETL*, *machine learning*, *stream processing*, *OLAP*, etc. To the best of our knowledge, there have been few scientific or empirical investigations into employing multiple DMSs for high-performance query processing over RDF datasets.

2.9 Benchmarking Efforts

In addition to the design of the DMSs (see Section 2.5), analysis of available DMSs using benchmark datasets has been a core topic of Semantic Web data management research. The growing number of applications that use RDF data provides the motivation for large-scale benchmarking efforts [83]. For example, some studies such as [45, 54, 129] presented new benchmark datasets. In particular, [54] proposed the Berlin SPARQL Benchmark dataset for comparing SPARQL engines with relational systems. Similarly, [45] proposed a benchmark dataset based on the e-commerce use case scenario called The Waterloo SPARQL Diversity TEST Suite (WatDiv) in order to analyze the correlation between DMS performance against varying query structures and complexities. There are studies such as [22] which comprehensively surveyed and analyzed available datasets.

Some other studies such as [102] did not propose any new dataset but attempted to use available benchmarks and distributed NoSQL DMSs for reporting key advantages and drawbacks of them. Some

other authors have performed similar experimental comparisons across database types. For instance, Abreu et al. [130] compared RDF-3X [44] as a native RDF-store with a number of graph databases, showing the supremacy of RDF-3X for graph pattern matching [131]. Hernandez et al. [131] compared the performance of four different DMSs, namely, Virtuoso, Blazegraph, Neo4J, and PostgreSQL. The focus of their experiments was to reveal the strengths and weaknesses of the tested DMSs in the context of Wikidata KG. Note that this and similar major studies (e.g., [45]) were published over five years ago. Our comparative study in Chapter 4 complements such studies.

2.10 Conclusion

This chapter has presented an overview of the technical background along with a review of the related literature. It has introduced KGs in Section 2.1 in which various definitions, a brief historical overview, and major application use cases have been discussed. Key requirements to unlock the full potential KGs such as the RDF data model and its foundational elements have been presented in Section 2.2. This has been followed by background information on SPARQL query language in Section 2.3. KG query types were also covered in Section 2.4. An in-depth discussion of KG data management systems and a review of major state-of-the-art approaches have been provided in Section 2.5 along with an overview of query optimization in Section 2.6. Having covered the background and literature review in this chapter, we present the research approach and methods in the next chapter.

Chapter 3

Research Approach and Methods

The main advantage of RDF is precisely that it can be used to represent KG content across the full spectrum of structuredness, that is to say, from completely unstructured to fully structured. This inherent flexibility is perhaps the main reason behind RDF's widespread acceptance. Blurring the structuredness lines, however, poses performance challenges for RDF data management since no assumptions can be made a priori about the KG content that it is going to be stored [50]. Although designing Data Management Systems (DMSs) with customized physical storage layouts for RDF datasets is well-studied and related best practices are widely available, it is not very clear if the variability in the data and the applications' requirements can be matched by any available sophisticated DMSs. A systematic approach would identify this and determine how efficiently major existing DMSs perform to store diverse RDF datasets and execute diverse queries over them. The systematic approach adopted for this study is presented in this chapter.

3.1 Introduction

By leveraging the RDF data model along with other standards and technologies, an increasing number of KGs has been published over

the past few years covering diverse domains ranging from commonsense and encyclopedic knowledge to geographic information, general life sciences, and evolutionary biology data. This proliferation of KGs has provided important opportunities for various application use cases. It also heightened our awareness of the need for employing RDF-stores to *efficiently* store and process the content of RDF datasets for semantic and other applications to unlock the full potential of using KGs. While the significance of this need has already been recognized, our understanding of how efficiently major existing DMSs perform to store diverse RDF datasets and execute queries over them is still somewhat limited.

In this chapter, we discuss our approach to evaluating the performance of DMSs for storing large-scale KGs with different levels of structuredness and executing a large and diverse number of queries over them. We begin by defining *diversity-tolerant* RDF-stores in Section 3.2. On this foundation, we will go on to present our approach to evaluating the performance of RDF-stores. In Section 3.3, we develop clear guidelines on how to detect potential inefficiencies of RDF-stores and discuss the design of a controlled experiment to perform a comparative analysis of RDF-stores. Finally, we provide detailed information about the basis of our evaluation.

3.2 Definition of Diversity-tolerance

When designing RDF-stores, three properties are commonly desired:

- **Support for a diverse range of query features.** This includes required and optional graph patterns, aggregation, subqueries, negation, along with their conjunctions and disjunctions as well as creating values by expressions and extensible filtering (see Section 2.3.1).

- **Support for widely varying RDF datasets in terms of structuredness and size.** This refers to the structuredness and size of RDF data which is introduced and defined in Section 2.2.3. In brief, the structuredness of T is affected by the

sparsity or absence of its predicates across instances. In a KG dataset with high structuredness, we expect that for any two instances of the same type, the instances have exactly the same number of predicates with the exact same names.

- **Exhibiting good performance.** In the context of KG query processing, the performance is typically measured by query execution times. This is an end-to-end time counted starting from a query submission time to the time the final (and correct) result is returned.

A diversity-tolerant RDF-store is an RDF data management system that can achieve all the above-mentioned desiderata simultaneously. The diversity tolerance of an RDF-store can be evaluated by exploring interactions between (i) its support for the full range of query features (**Q**), (ii) its support for widely varying RDF datasets (in terms of structuredness and size) (**S**), and (iii) time the RDF-store takes for processing SPARQL queries over the RDF data before returning the correct result efficiently (**P**).

Theoretically, RDF-stores can be diversity-tolerant and efficiently execute diverse queries by constructing an optimal plan for each query (background information can be found in Section 2.6) based on accurate cardinality estimations (and the cost model) and effective indexing schemes. However, S and Q have significant impacts on P by affecting the cardinality estimation accuracy (see Section 2.6.4), the effectiveness of indexes (see Section 2.5.4.2), and plan enumerations (see Section 2.6.3). Our approach seeks to evaluate these effects. This will lead to a generalized approach to evaluating any RDF-store's diversity tolerance.

3.2.1 Remarks on the Impacts of Structuredness (S) on Performance (P)

Obtaining accurate statistics for efficient join processing is already recognized in the database community as a standard yet difficult problem [92]. This is because estimating the join sizes of two and

more triple patterns clearly calls for estimating correlations between triple patterns which is not an easy task to accomplish especially for datasets with lower levels of structuredness where higher levels of the sparsity of the dataset complicate the query optimizers' efforts at deriving reliable estimates (see Section 2.6.4).

While lower values for structuredness typically affect the accuracy of cardinality estimations, higher values can decrease the effectiveness of the indexes. More specifically, higher values of structuredness typically indicate the presence of identical predicates across triples which can impact the density of indexes negatively since density usually shows the ratio of unique values. Thus, the higher the density, the lower the selectivity, and consequently, the lower P for highly structured datasets.

3.2.2 Remarks on the Impacts of Query features (Q) on Performance (P)

Generating optimal execution plans can lead to faster query execution times (see Section 2.6.3). However, the number of triple patterns in a query and the complexity of query features would enlarge the search space in order to find an efficient plan. The search space can consist of $n!k^n$ plans where n denotes the number of required joins and k the number of available algorithms for join processing that are implemented by the RDF-store (like merge join and hash join) [132]. One of the difficulties in executing queries with conjunctions or disjunctions such as Union and Optional patterns is that RDF-stores need to generate plans with all the possible combinations [81, 92]. In theory, sampling-based query execution may help in these cases, but it is not very practical to follow a sampling-based plan enumeration (or a randomized selection) to find the optimal plan since the sample would have to be large to be useful, which is obviously very expensive. As a result, it is not an easy task for RDF-stores to generate optimal plans for queries with multiple joins as well as optional patterns and unions.

3.2.3 Remarks on the of Impacts of Volume on Performance

The performance of RDF-stores relies on their cost models. These, in turn, are highly dependent on the accuracy of cardinality estimations. The size of stored data can affect the statistics. For larger datasets, the likelihood of inaccuracies increases leading to estimated cardinalities being less likely to be reasonable reflections of the actual content of KGs (more information can be found in Section 2.6.4). In addition, the execution of queries tends to get slower when the size of the underlying data grows given that a larger portion of the data will typically need to be scanned to return the result. Scanning more data means more intermediate results that need to be managed. This puts more pressure on the computer hardware and software resources like the relevant components of the operating system (e.g., the I/O subsystem), the storage media, etc. For faster query processing, RDF-stores need a good awareness of the physical locations of the data and indexes (on the primary storage media, in-memory, etc.) to decrease the CPU and I/O costs. However, this is more difficult to achieve when very large KGs are processed.

3.3 Evaluation Approach: Controlled Experiments

3.3.1 Overview

In this research, we use experimental evaluation to measure the efficiency of KG query processing. One of the main reasons behind our experiments and measurements is to test propositions related to the behavior and efficiency of major RDF-sores. We seek to quantify RDF-stores' bottlenecks in KG query processing with the goal of proposing improved design choices that can contribute to improved performance.

Evaluating the performance of an RDF-store can help us to detect its potential efficiency issues (if any). We make the performance evaluation more insightful by repeating the same experimentation on

several major RDF-stores and using their query execution times as quantitative evidence to compare them with each other. The comparison is fair and reliable if we set up a control experiment in which everything is constant except for RDF-stores that are changed each time. To achieve conformity with accepted rules and standards of an experimental evaluation, we use appropriate benchmark datasets and queries. This helps to make the experiments highly persuasive and replicable. The basis of our evaluation is presented in the following.

3.3.2 Basis of Evaluation

A good choice of measure is crucial to practical system evaluation and to further insightful analyses. As part of the experimental evaluation, we measured the query execution time. This is an end-to-end time computed from the time of query submission to the time when the result is outputted. After the execution of each query, we carefully check to ensure that the output results are correct and exactly the same across different employed RDF-stores.

3.3.3 Robustness of Measurements

In general, for performance evaluation based on a series of runs, query execution times will be average times. For fairness and to ensure that our experimental design is robust and persuasive, the query times reported for each data management system are averaged over five successive runs (with no delay in between) to account for any randomness and noise. In this research, we used the Geometric mean – the nth root of the product of n numbers – instead of the arithmetic mean since it typically provides a more accurate reflection of the total speedup factor [89].

3.4 Conclusion

This chapter has discussed that KG query performance (P) is attributable to the characteristics of the RDF datasets (S) and the

SPARQL query features (Q). This chapter also presented the research approach and methods of this thesis. Our study adopted an experimental evaluation method. We presented that query execution times (as quantitative evidence) are measured in our experimental evaluation, that is to say, end-to-end times computed from the time of query submission to the time when the result is outputted. The robustness of measurements is also discussed.

Chapter 4

High-performance Knowledge Graph Query Processing: A Comparative Analysis

We have previously discussed the performance challenges posed to various Data Management Systems (DMSs) by RDF data models aiming to represent diverse KG content in both *structured* and *unstructured* environments with diverse query types. Although the significance of these factors has been recognized, the comparative performance of major DMSs executing various query types is still somewhat limited. In this chapter, we draw on experimental approaches presented in Chapter 3 to evaluate the performance of existing RDF-stores to store diverse datasets while executing archetypal queries.

4.1 Introduction

In this chapter, we carry out analyses of the performance of SPARQL query processing across diverse DMS and query types. Our analyses cover a variety of query types to measure their respective performance across a range of RDF-stores, with the aim to provide

a fine-grained comparative analysis of major native and non-native RDF-store types. Prior to this, we clarify exactly what is meant by KG query processing and what are the major steps involved in a query's execution process in the following.

In general, KG query processing proceeds as follows. First, the KGs are loaded into a DMS, triggering the creation of related indexes which are crucial to performance (see Section 2.5.4.2 of Chapter 2). Users are then able to submit their queries which are parsed according to the SPARQL grammar. During this phase, the DMS also validates that the user has appropriate permissions to run the query. After this, a physical optimization takes place. To achieve this, existing cardinality estimations are utilized and index orders will be selected to ensure the best performance. This then triggers a complex process whose outcome is to generate a low-cost *physical operator graph* to perform the query. Finally, the execution phase is performed, wherein the desired data items are retrieved and the answer to the query will be returned [52, 133].

The remainder of the chapter is organized as follows. We begin with presenting a background in Section 4.2, highlighting which major factors contribute to query performance. Section 4.3 presents the experimental configuration including benchmark datasets, queries, configurations and the KGs' bulk loading process that is included. Section 4.4 presents the results of the experiments, followed by remarks of the lessons learnt in Section 4.5. Section 4.6 concludes the chapter.

4.2 Remarks on Factors Influencing Performance

SPARQL enables users to formulate their queries by specifying "*what*" is desired as an answer without specifying (in any way) "*how*" the answer is to be retrieved. This is often referred to as the "*non-procedurality*" feature of SPARQL. On this basis, the user is not required to be involved in specifying the step-by-step access plan to execute the query since it is generally feasible to leave this to the

employed RDF-store. For many queries, however, it is a challenge to generate an optimized plan following a step-by-step access strategy that minimizes run time and resource use. Typical challenges include how to efficiently utilize CPU, RAM, and I/O to perform intermediate steps of query processing, such as sorting and the order of joins. These challenges are explained below.

Resource use. Three major resources are used for query processing; CPU, storage devices (typically referred to as I/O), and RAM. Among these, the maximum RAM capacity usable by an RDF-store is specified at system initialization time, prior to loading KGs, whereas CPU and I/O resources are under the control of the RDF store to be used during query processing. RDF-stores typically attempt to generate execution plans that can minimize the utilization of CPU and I/O, achieved through estimations based on the query. A major challenge, however, is to derive associated CPU and I/O usages before the execution of the query as it depends on factors which include things such as the complexity of the query and the accuracy of cardinality estimations. Although a number of sophisticated techniques are implemented, estimating the utilization of resources still poses inherent challenges for RDF data management systems.

Joins. The number of algorithms that can be used by an RDF-store for joining triple patterns is an important factor. Typical algorithms are *nested loop join*, *hash join*, and *merge join*. Each of these algorithms shows performance advantages in situations that can arise in performing a join between two triple patterns. Given a query with a number of triple patterns, a join of two triple patterns in an RDF-store can be performed as follows. It begins with an index lookup to retrieve all triples that match the first triple pattern of the query. During this step, matched triples are usually read from the disk and cached in memory; often referred to as the *left* side, or the outer side, of the join. A similar lookup is to be performed to retrieve triples of the *right*, or the inner, side of the join. When both sides (i.e., left and right) are retrieved, the RDF-store uses a join algorithm to combine them to answer the query. The choice of join algorithm can significantly affect the efficiency of the join processing.

For instance, the nested loop join is typically efficient when only a "*small*" number of triples qualify from the outer side of the join, or when the inner side of the join is "*small enough*" that all index and data become resident in memory during the join processing. The merge join (sometimes referred to as *merge scan join* or *sort merge*) is more likely to be efficient when both sides are already sorted [133]. Although the importance of using an efficient join algorithm is recognized, choosing the most optimal join algorithm to process a given query is still a major challenge.

It is now well established that the above-mentioned factors play major roles in efficient KG query processing. In the next section, we provide a fine-grained comparative analysis to understand how efficiently RDF-stores, both native and non-native, perform under diverse RDF datasets executing archetypal query types.

4.3 Experimental Context

In this section, we report our experimental context and details of the KG benchmark datasets used for the experiments presented in this chapter. This also includes DMSs' configuration, indexing, and the data loading process.

4.3.1 Benchmark Datasets

Five well-known benchmarks used for our experiments are presented in this chapter. All or some of these have also been used in previous studies such as [22, 45, 54, 57, 85]. These benchmarks are: Berlin SPARQL Benchmark[1] (BSBM) [54], Waterloo SPARQL Diversity Test Suite[2] (WatDiv) [45], FishMark [55], BowlognaBench [56], and BioBench-Allie[3] [57]. Of these, WatDiv and BSBM follow specific rules that allow us to scale the datasets to arbitrary sizes using their scale factors but other datasets are fix-sized. Table 4.1 shows the

[1] http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/

[2] https://dsg.uwaterloo.ca/watdiv/

[3] http://allie.dbcls.jp/

statistical information related to these benchmarks. The RDF representations of these benchmarks are available in different formats such as N-Triples, Turtle, and XML. We used the RDF/N-Triples format. To load them into a document-store like MongoDB, we had to convert them to the JSON-LD[4] format. We performed the conversion using a parser designed and developed as part of this project[5].

4.3.2 Knowledge Graph Queries

We ran benchmark queries against the corresponding datasets using the four DMSs. From the above-mentioned benchmark suites, twelve queries were selected[6], representative of the four major query types, or the archetypal KG query types, that are discussed in Section 2.4. These queries were selected as they provide varying degrees of selectivity and complexity with the ability to highlight positive characteristics of the systems under test. The selected queries are the following:

i) *Subject-Subject* Join Queries:

- **FishMark-Q5** — Query 5 from FishMark
- **BowlognaBench-Q7** — Query 7 from BowlognaBench
- **WatDiv-Q7** — Query 7 from WatDiv

ii) *Subject-Object* Join Queries:

- **BioBench-Allie-Q2** — Query 2 from BioBench-Allie
- **WatDiv-Q21** — Query 21 from WatDiv
- **WatDiv-Q22** — Query 22 from WatDiv

iii) *Tree-like* Join Queries:

- **BioBench-Allie-Q1** — Query 1 from BioBench-Allie
- **FishMark-Q19** — Query 19 from FishMark
- **BowlognaBench-Q14** — Query 14 from BowlognaBench

[4] https://www.w3.org/2018/jsonld-cg-reports/json-ld/
[5] The source code is available through https://github.com/m-salehpour/cmp
[6] Queries available at https://github.com/m-salehpour/cmp

iv) *Optional* Join Queries:

- **BSBM-Q2** — Query 2 from BSBM
- **BSBM-Q4** — Query 4 from BSBM
- **FishMark-Q2** — Query 2 from FishMark

Benchmark	Scale (nominal)	#Subjects	#Predicates	#Objects	#Triples
BSBM	10M	934,324	40	1,919,901	10,190,687
	100M	9,197,305	40	15,207,734	100,652,457
	1000M	91,647,129	40	140,996,171	1,004,406,629
WatDiv	10M	521,585	86	1,005,832	10,916,457
	100M	5,212,385	86	9,753,266	108,997,714
	1000M	52,120,385	86	92,220,397	1,092,155,948
BioBench-Allie	100M	19,227,252	26	20,287,231	94,420,988
FishMark	10M	395,491	878	1,148,159	10,002,178

Table 4.1: Statistics of the Benchmark datasets

4.3.3 Computational Environment

Our benchmark system is a Virtual Machine (VM) instance with a 2.3GHz AMD Processor, running Ubuntu Linux (kernel version: 4.4.0-170-generic), with 48GB of main memory, 16 vcores, 512K L2 cache, 5TB instance storage capacity. The VM cache read is roughly 2799.45MB/sec and the buffer read is roughly 35.85MB/sec (i.e., the output of the "hdparm -Tt" Linux command). The operating system is set with no "soft/hard" limit on the file size, CPU time, virtual memory, locked-in-memory size, open files, processes/threads, and memory size using Linux "ulimit" settings.

4.3.4 Experimental Platforms

We chose four different DMSs: (1) Row-store Virtuoso (Open Source Edition, version 06.01.3127), (2) Column-store Virtuoso (Open Source Edition, Version 07.20.3230–commit 4a668a5), (3) Blazegraph[7] (Open Source Edition, version 2.1.5–commit 3122706), and (4) MongoDB (community edition, version: 4.0.9).

For our experiment, Virtuoso was selected since it is already employed as the DMS of choice for a broad range of projects, e.g., the Linked Data for the Life Sciences project[8]. Blazegraph was selected since it is the DMS behind Wikidata[9] (i.e., a KG constructed from the content of Wikimedia sister projects including Wikipedia, Wikivoyage, Wiktionary, and Wikisource). MongoDB was selected as a representative document-store since it is considered to be the leader in this class of tools [103]. MongoDB's efficacy for executing queries over KGs has not been researched extensively but some academic prototypes such as [103, 134, 135] have already shown its efficacy in similar contexts.

[7] Previously known as Bigdata DB. It is alleged that Blazegraph was acquihired by Amazon and the Amazon Neptune is based on Blazegraph.

[8] https://bio2rdf.org/sparql

[9] https://query.wikidata.org/

4.3.4.1 Configurations

Configuration of row- and column-store Virtuoso. We configured both of them based on the vendor's official recommendations.[10] For example, we configured the Virtuoso process to use the main memory and the storage disk effectively by setting "*NumberOfBuffers*" to 4,000,000, "*MaxDirtyBuffers*" to 3,000,000, and "*MaxCheckpointRemap*" to *a quarter* of the database size as recommended. We also used the following version of GNU packages that are necessary to build column-store Virtuoso: *GNU gpref 3.0.4*, *libtool 2.4.6*, *flex 2.6.0*, *Bison 3.0.4*, and *Awk 4.1.3*.

Configuration of Blazegraph. We configured it based on the vendor's official recommendations[11] as well. For example, we turned off all *inference*, *truth maintenance*, *statement identifiers*, and the *free text index* in our experiment since *reasoning efficiency* was not part of our research focus in this research.

Configuration of MongoDB. We used its default settings. We set its level of *profiling to 2* to log the data for all query-related operations for precise and detailed query execution time extraction.

4.3.4.2 Indexes

Indexes have a major impact on the overall performance of a system. Each system was configured with default values or recommendations from the vendor.

Virtuoso. The default indexing scheme of both column- and row-store Virtuoso was used, as highlighted on the official website: "alternate indexing schemes are possible but will not be generally needed"[12]. More specifically, Virtuoso's data modeling is based on a relational table with three columns[13] for S, P, and O (i.e., S: Subject, P: Predicate, and O: Object) and carrying multiple indexes over that table to

[10] http://vos.openlinksw.com/owiki/wiki/VOS/VirtRDFPerformanceTuning

[11] https://wiki.blazegraph.com/wiki/index.php/PerformanceOptimization

[12] http://docs.openlinksw.com/virtuoso/rdfperfrdfscheme

[13] In the case of loading named graphs, it adds another column for the context, called C.

provide a number of different access paths. More recently, column-store Virtuoso added columnar projections to minimize the on-disk footprint associated with RDF data storage. Virtuoso (both row- and column-store) creates the following compound indexes by default for the loaded KG: PSO, PO, SP, and OP.

Blazegraph. As recommended on its official website[14] we did not change its default indexing schema. The indexes are based on the "B+tree" data structure. Blazegraph typically uses the following three indexes for the triples mode: SPO, POS, and OSP. For normal use cases, these indexes are laid out on variable-sized pages. These index pages are read from the backing store and loaded into main memory on demand (i.e., into the Java heap).

MongoDB. We created indexes on the name/value pairs of the JSON-LD that were representatives of subjects and predicates. The default storage engine of MongoDB was used to store JSON documents, i.e., the `WiredTiger` key-value store. MongoDB uses the binary equivalent, BSON, for each JSON document in which the structure remains unchanged. Each JSON document is assigned a unique, arbitrary identifier as the key, and the document itself is considered the value. Therefore, as MongoDB uses B-trees, it creates an index on the name/value pairs of each JSON document.

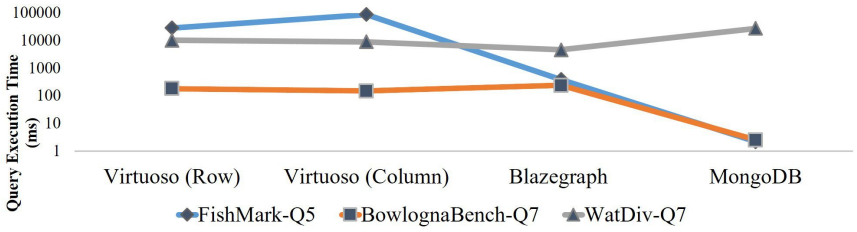
4.3.4.3 Remarks on Bulk Loading and Measurement

To load the benchmark datasets into their respective databases, we utilized native tools. Virtuoso (both column- and row-store) used the native `ld_dir` bulk-loader function. In Blazegraph, we used the native `DataLoader` utility[15]. Finally, in MongoDB, the native `mongoimport` utility was called. Once the KG data was loaded into the database, the benchmark queries were executed.

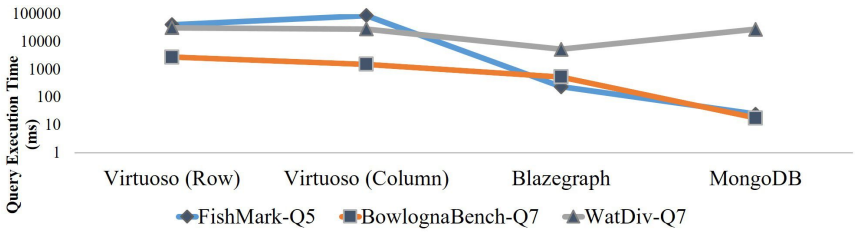
Query execution times were used to form the basis of our evaluation. We define query execution time computed as the time of

[14] https://wiki.blazegraph.com/wiki/index.php/PerformanceOptimization

[15] Available online: https://wiki.blazegraph.com/wiki/index.php/Bulk_Data_Load.



(a) Subject-subject join (warm)



(b) Subject-subject join (cold)

Figure 4.1: Impacts of subject-subject join queries on the DMSs (cold- and warm-run). X axes show DMSs and Y axes show the execution time of each query in milliseconds (using log scale).

query submission to the time the result is output, representing the end-to-end time of the requested query. After the execution of each query, we verified the correctness of the output across multiple DMSs. Query times for both cold (empty cache) and warm (warm cache) are reported. For fairness, warm-run query times reported for each DMS are averaged using the geometric mean over 5 successive runs (with no delay between runs).

4.4 Exploratory Analysis

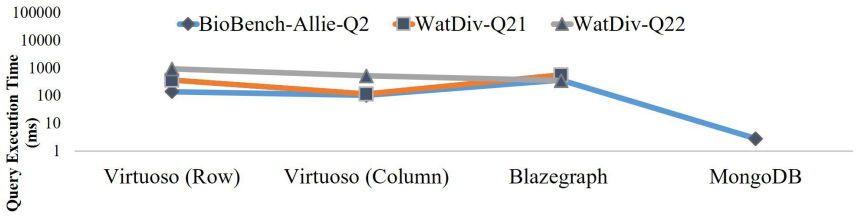
In this section, we present our experimental results. We also conduct a set of exploratory analyses to illustrate how efficiently major existing (both native and non-native) RDF-stores perform to store diverse RDF datasets and execute archetypal query types over them.

4.4.1 Results

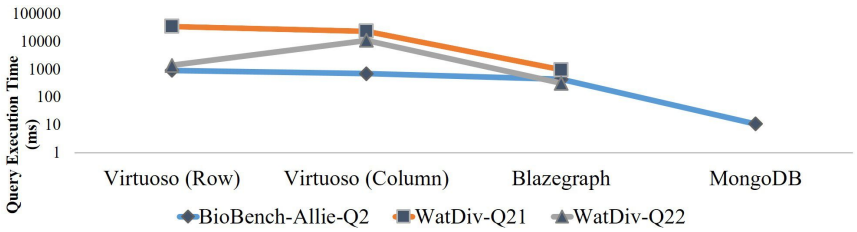
The experimental results highlight various query types and their performance across the selective DMSs.

Subject-subject Joins. The query execution times are shown in Fig. 4.1 in which the X axis represents the DMSs and the Y axis shows the execution times in milliseconds (log-scale) of queries, namely, **FishMark-Q5**, **BowlognaBench-Q7**, and **WatDiv-Q7**. Fig. 4.1 highlights that MongoDB is able to execute highly-selective queries of this type over one order of magnitude faster than other DMSs. For example in the warm run, MongoDB executed **FishMark-Q5** in 2.19ms whereas BlazeGraph, column-store Virtuoso, and row-store Virtuoso executed the same query in 394.24ms, 89543.81ms, and 29045.86ms respectively. However, our results indicate that Blazegraph performs at least 2x faster than other DMSs when subject-subject join queries are low-selective as used in **WatDiv-Q7**. In Fig. 4.1, the differences between cold- and warm-run show that Virtuoso (row- and column-store) can take advantage of caching techniques more than other DMSs. For example, Virtuoso (row and column) executes **BowlognaBench-Q7** in over 1500ms (cold-run) while its execution time is around 150ms in a warm-run.

Subject-object Joins. The query execution times for the selected subject-object join queries are shown in Fig. 4.2 in which the X axis represents the DMSs and the Y axis depicts the execution time of queries in milliseconds (log-scale), namely, **BioBench-Allie-Q2**, **WatDiv-Q21**, and **WatDiv-Q22**. Although MongoDB executed **BioBench-Aliie2-Q2**, a high-selective query, over 2 orders of magnitude faster than other DMSs, it could not finish the execution of **WatDiv-Q21** and **WatDiv-Q22** within the given time-out period of 50,000 milliseconds. The complexity and non-selectivity of these two queries may have contributed to the unsuccessful execution over MongoDB. However, Fig. 4.2 also highlights that other DMSs experienced comparable performance. For instance, **WatDiv-Q21** executed over Blazegraph in around 570 milliseconds (warm-run), where this execution time is equal to 118.38 and 374.6 milliseconds



(a) Subject-object join (warm)



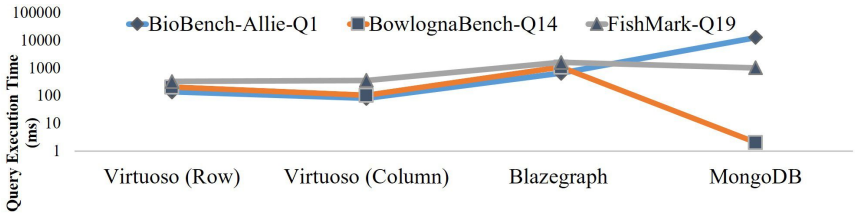
(b) Subject-object join (cold)

Figure 4.2: Impacts of subject-object join queries on the DMSs (cold- and warm-run).

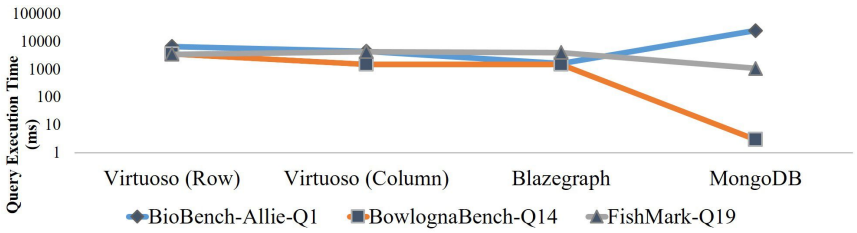
for column-store Virtuoso and row-store Virtuoso, respectively.

Tree-like Joins The query execution time for the selected tree-like join queries is shown in Fig. 4.3. This figure reveals that row- and column-store Virtuoso performed similarly for warm-run execution of **BioBench-Allie-Q1** and **FishMark-Q19** while Blazegraph is around 5x slower. MongoDB appeared to be the slowest for warm-run execution of **BioBench-Allie-Q1** while its performance is comparable with Blazegraph for **FishMark-Q19**. **BowlognaBench-Q14** executed around 2 orders of magnitude faster using MongoDB, as it contains a high-selective tree-like join query. The comparison between cold- and warm-run execution of **FishMark-Q19** can also give rise to the importance of the role that caching techniques play in query performance where MongoDB is the fastest in cold-run, but in warm-run, it is almost the slowest (after Blazegraph).

Optional Joins. The query execution time for the selected optional join queries is shown in Fig. 4.4 in which MongoDB outperformed other DMSs. The high-selectivity of the selected queries may have been an important factor yielding MongoDB's performance ad-



(a) Tree-like join (warm)



(b) Tree-like join (cold)

Figure 4.3: Impacts of tree-like join queries on the DMSs (cold- and warm-run).

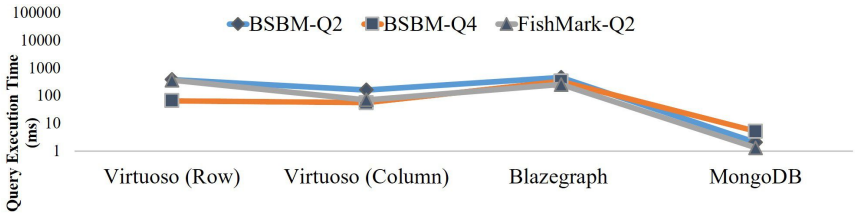
vantage. Row-store Virtuoso was the slowest across others while column-store Virtuoso performed over 3x faster than Blazegraph to run these queries (warm-run). However, during the cold-run experimentation, other than MongoDB's performance advantage, Blazegraph performed slightly better than others when executing the **BSBM-Q2**.

4.4.2 Analysis

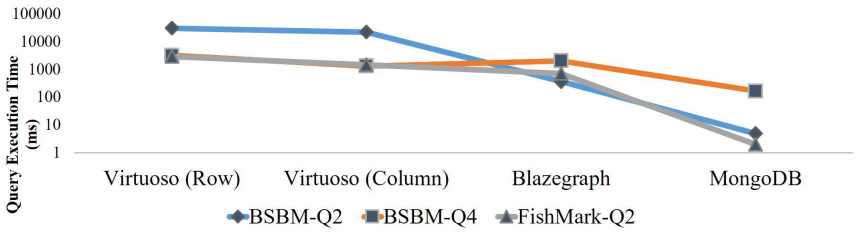
The experimental results yield insight into the performance of the selected DMSs when executing a range of query types. The results highlight, however, that no single DMS proves superior in all benchmark scenarios, suggesting that a DMS should be selected and tailored to the query types being executed.

4.4.2.1 Mapping Query Types and DMS types

MongoDB was shown to outperform other DMSs when executing subject-subject joins, which is a product of storing all triples with the same subject within the same document, effectively leading to



(a) Optional join (warm)



(b) Optional join (cold)

Figure 4.4: Impacts of optional join queries on the DMSs (cold- and warm-run).

only a single index lookup for this query type. Others, however, keep intermediate results in memory (shown in Section 4.2) and perform joins, typically *hash joins* as shown in both variants of Virtuoso and Blazegraph, which lead to hindered performance.

Blazegraph offered a significant performance improvement with subject-object joins, especially with cold-run queries. The literature highlights the benefit of merge-join algorithms in these scenarios [52], however, to the best of our knowledge there is no DMS that implemented such a join integrated with their query processing engine. As a result, the DMSs use an *index nested loop join* to support queries of this type (see Section 4.2). The performance exhibited by Blazegraph may stem from the use of a *B^+-tree-based index nested loop join* that is more read-optimal compared to the *bitmap index-based* used by Virtuoso. Similarly, the performance may also be a factor in the accuracy of Blazegraph's cardinality estimation, as it affects the accuracy and selection of an efficient join-ordering.

Virtuoso (more specifically column-store) displayed increased performance using tree-like join queries, which technically exhibit both

subject-subject and subject-object joins. The performance relies on the complexity and efficiency of the query optimizer. In the experimental analysis, Virtuoso (column- and row-store) displayed better performance under queries with low-selectivity. We speculate that Virtuoso's vectorized query execution model, along with well-engineered query optimization engines, may explain the high performance when compared to others. In addition, the column-store variant of Virtuoso stores indexes more compactly, contributing to less I/O and positive performance indications in tree-like joins.

4.4.2.2 Optional Joins and DMSs

There is no evidence to show that any of the four DMSs implement specialized optimizations for optional joins. As a result, although MongoDB showed better performance for running high-selective queries, there was no significant difference in query performance observed across all DMSs for low-selective queries. We note that both row- and column-store Virtuoso integrate a compression strategy for storing KG datasets. Furthermore, bitmap indexing provides row- and column-store Virtuoso with better space utilization as compared to the B^+-tree of Blazegraph (or MongoDB). In this regard, we speculate Virtuoso is likely to aggressively prune intermediate results and perform faster than others for optional join query processing, especially for low-selective optional join queries.

4.4.2.3 The effects of scale

FishMark, BioBench-Allie, and BowlognaBench comprise fixed-size datasets that cannot be scaled. In contrast, WatDiv and BSBM are scalable and allow higher triples to be tested to show the effect of scale. In this chapter, we reported the corresponding query time of these two datasets with 100M triples. However, corresponding results for datasets with 10M and 100M are computed and available online[16]. Our results indicate that the trends and performance differences between DMSs remain almost unchanged at scale.

[16] Available at https://github.com/m-salehpour/cmp.

4.5 Lessons Learned

Locality. Column-store Virtuoso and MongoDB are designed to increase data locality while storing KGs' content more than others. In the column-store Virtuoso storage model, each column of a table or index is stored contiguously to provide physical adjacency. Therefore, when queries (e.g., tree-like joins) need to access a subset of columns from one table, only those columns actually being accessed need to be read from the disk which can be culminated in better use of I/O throughput and memory. This locality has the potential to reduce the traffic between CPU cache and main memory and provide better CPU utilization. Similarly, MongoDB takes advantage of data locality since all the triples related to one resource (i.e., a subject in the JSON-LD) are physically located together. We speculate that such locality leads to denser data layout, more RAM locality, and more CPU cache locality contributing to the increased overall performance on high-selective KG queries.

Cache Efficiency. DMSs usually utilize their internal and underlying file system cache memory for performance. When enough free memory is available and allocated to DMSs, efficient utilization of this memory for caching purposes can typically contribute to faster warm-run query execution. When comparing the results of different queries across the DMSs in cold and warm environments, it suggests that column-store Virtuoso provides more effective cache management. In applications with ad-hoc queries, the cache management may not impact the performance significantly, however, for cases in which a number of queries are repeated periodically, employing suitable cache techniques can positively contribute to query performance.

Intermediary Result. We note that the performance of different query types tends to be negatively affected by the size of the query's output, more specifically the intermediary results which pose a challenge to DMSs. When a query type contains more than a single triple pattern, DMSs usually have to scan large parts of indexes for each triple pattern and join the result of these scans, which produce

large intermediary results. We observed that even when the query itself is very selective with small output, the size of the intermediary results is still large.

4.5.1 Limitations

Our results indicate that no single DMS displays superior query performance across different query types. These results are likely to be generalizable. In our experiments, we had four archetypal query types, however, there may be other query types as well that are to be considered in the future.

Currently, the maximum size of each JSON document in MongoDB is 16MB. It rejects JSON documents when their size exceeds this value. Technically, the maximum document size in document-stores helps ensure that a single document cannot use an excessive amount of memory, but the JSON-LD representation of KGs might be affected negatively by this. In our experiments, there were no cases in which the document size exceeded the maximum value. However, in principle, the size of JSON documents may exceed the maximum document size depending on the KG content.

In our experiments, DMSs are implemented using different programming languages. For example, Blazegraph is written in Java, or Virtuoso is developed using C. We did not explore the effects of each DMS's underlying programming languages on its performance although it is already recognized that some programming languages like C can deliver high performance via low-level access to RAM and flexible control over allocation and freeing of RAM [136]. We speculate that Virtuoso was more benefited from these features since it is written in C. In contrast, DMSs that are written in high-level languages like Java (e.g., Blazegraph) can take advantage of type- and memory-safety and convenient abstractions such as threads as well as garbage collection (it may also consume more CPU time and can cause delays) to further reduce memory bugs [136]. It would be very expensive to re-write existing DMSs in the same programming language (with the same level of optimality) to compare the effects

of their architecture and design choices on query execution times in a very controlled way. However, it is worth exploring different ways in which the language helped the development of high-performance DMSs for KG query execution and situations in which it was less helpful.

4.6 Conclusion

In this chapter, we have explored the efficiency of major existing RDF-stores and their performance when storing diverse RDF datasets with archetypal query types being executed. In summary, we note four key takeaways that provide insight into the performance of DMSs in relation to query types and datasets:

- There are significant factors that relate to different DMSs and query types, hinting that a DMS must be carefully selected and tailored based on the data and queries in the deployment environments.
- Performance advantages from underlying storage layout, such as increased data locality and caching techniques, are evident in tree-like join queries as shown by column-store Virtuoso. This indicates that a DMS can benefit from caching and locality techniques for performance in queries that require large reads.
- Evident in Blazegraph, the observed results highlighted the effectiveness of suitable cardinality estimation and efficient query optimization on cold-run executions in subject-object join queries.
- Taking advantage of data locality and employing efficient data structures such as B-trees for implementing indexes in MongoDB can contribute to over one order of magnitude better performance for executing subject-subject join queries, especially for queries with higher selectivity.

Chapter 5

High-performance Knowledge Graph Query Processing: A Polyglot Model-based Approach

The growing size and variety of KG datasets have triggered the research and development of a range of Data Management Systems (DMSs). The workload of queries executed across this system has also experienced a transformation to a diverse set of query types, to the extent that the performance of representative DMSs tends to vary significantly across diverse query types. We have seen in the previous chapter (Chapter 4) that no single DMS shows dominant performance across diverse KG query types. In this chapter, we address the shortcomings by architecting a polyglot model of KG query processing.

5.1 Introduction

The growing diversity and size of KG content pose unique challenges to DMSs, not only to effectively store but efficiently query KG datasets. The diversity also extends to the queries executed for a range of applications running on these systems. Although there have been proposals to address these challenges through the development of a range of DMS types, the consensus seems to be that a single one-size-fits-all DMS is unlikely to emerge for efficient KG query processing [19, 45].

Our proposed solution builds upon the foundations inspired by Ashby's First Law of Cybernetics [58] and Stonebraker et. al. [59] which can be paraphrased in this context to state that the variety in solution architecture should be greater than or at least equal to that of the variety displayed by the data and the queries. This can be achieved through an architecture based on the *polyglot* model that depicts query processing and a range of access languages supported by a system design that analyzes the query being run and matches to the likely best-performing database engine to execute the query. From a conceptual standpoint, this model implies a requirement to integrate multiple DMSs in a single system, supporting friction-free translation across the different query and access languages to optimize query execution performance.

We begin this chapter by presenting our extensible prototype, SymphonyDB, a multi-database system providing a unified access layer that can analyze and translate individual queries Just-In-Time (JIT). By utilizing experimental results discussed in Chapter 4, showing that certain DMSs exhibit higher performance for specific query types, we have mapped different types of KG queries to the best-performing DMS. SymphonyDB draws on this to match any given query to the best-performing DMS among Virtuoso, Blazegraph, RDF-3X, and MongoDB at this time. We then provide details of the experimental setup in Section 5.3, in which Section 5.4 presents the results of query processing and related analyses performed during the experimental executions which include comparative performance

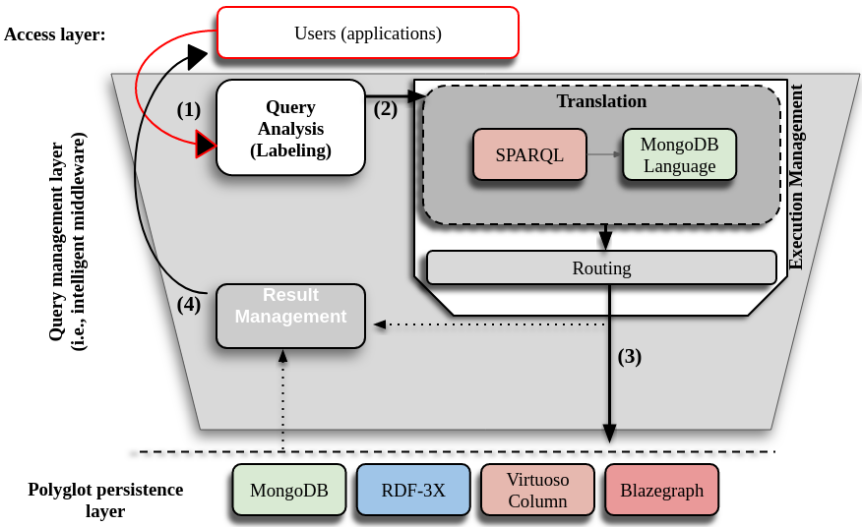


Figure 5.1: A schematic view of SymphonyDB's architecture

analysis of SymphonyDB against representative single DMSs with different KG query types. Section 5.5 concludes this chapter and presents closing remarks.

5.2 SymphonyDB: A Polyglot Model for Query Processing

In this section, we present *SymphonyDB*, a prototype that provides polyglot support for KG query processing. A schematic view of SymphonyDB's architecture is presented in Fig. 5.1. The proposed approach consists of three layers: access layer, query management, and polyglot persistence. Users interact with the proposed approach like they interact with any conventional single DMS. For example, a user may send their workload issuing SPARQL queries. The proposed architecture contains multiple DMSs internally where the query management layer has the responsibility of selecting one or more of the employed DMSs that can best serve requests made by each user. The query management layer directly uses the execution and storage engines of the underlying DMSs in the polyglot persistence layer. This enables the proposed approach to have full control of what

gets executed and how. Currently, the polyglot persistence layer includes *Virtuoso*, *Blazegraph*, *RDF-3X*, and *MongoDB* as representative DMS types. We give the details of the proposed approach's internals below. An abstract overview of the interactions across DMSs and the access layer is also expressed using the pseudocode in Algorithm 1.

The polyglot access management layer is the entry point for query processing over stored data, providing a unified query interface for accessing the underlying KGs. This layer receives the incoming SPARQL queries (shown by (1) in Fig. 5.1) and labels each query based on its characteristics (shown by (2) in Fig. 5.1) into three target categories, namely: subject-subject, subject-object, and tree-like (line 1 in Algorithm 1). The labels are used for routing (shown by (3) in Fig. 5.1) each query to one (or more than one) of the employed DMSs. After execution, query results are returned to the user (shown by (4) in Fig. 5.1). We defer details of the labeling approach to Section 5.2.2. After determining which DMS should be used for each query (line 2 in Algorithm 1), a suitable JIT query translation may be needed for further query execution (line 3-5 in Algorithm 1). For example, the incoming query is written in SPARQL, but it is labeled (and selected) to be run using MongoDB which cannot support SPARQL directly as an input query language. In this case, the incoming query needs to be translated JIT into an equivalent JavaScript-like query, MQL, to be executed over MongoDB. Handling this JIT translation process is one of the responsibilities of the access layer. The following sections describe the detail of each step.

5.2.1 Polyglot Database Management System Layer: Storing KGs

Selection of the likely best-performing DMSs is important in maximizing the power of a multi-database system. Currently, column-store Virtuoso, Blazegraph, RDF-3X, and MongoDB are included in SymphonyDB. Recall that we experimented with Blazegraph, MongoDB, column-store Virtuoso, and row-store Virtuoso in Chapter 4.

Algorithm 1: SymphonyDB KG query processing

Input: SPARQL queries (applications' workload)
Initialization: Let q be an incoming SPARQL query
1 $qLabel \longleftarrow$ **Query_labeling**(q)
2 $DMS \longleftarrow$ **DMS_select**$(qLabel)$
3 **if** $DMS == MongoDB$ **then**
4 $\quad q \longleftarrow$ **Translate_MQL**(q)
5 **end**
6 $qResult \longleftarrow$ **Route_execute**(q)
7 **Return_result**$(qResult)$

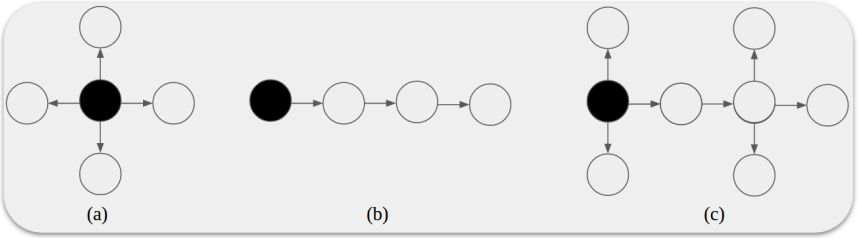


Figure 5.2: (a) An example of a subject-subject query pattern, (b) an example of a subject-object query pattern, and (c) an example of a tree-like query pattern

In our experiments in Chapter 4, column-store Virtuoso could load datasets easier and achieve better performance than row-store Virtuoso in many cases. Therefore, we did not include row-store Virtuoso in our prototype. Instead, RDF-3X was replaced to be included in SymphonyDB. RDF-3X is selected since it is widely used in a range of studies such as [45]. RDF-3X also has a number of interesting features. For instance, it creates exhaustive indexes on all permutations of triples along with their binary and unary projections. Its query processor is also designed to aggressively leverage cache-aware hash and merge joins. Our other selected DMSs are also broadly used in many major projects.

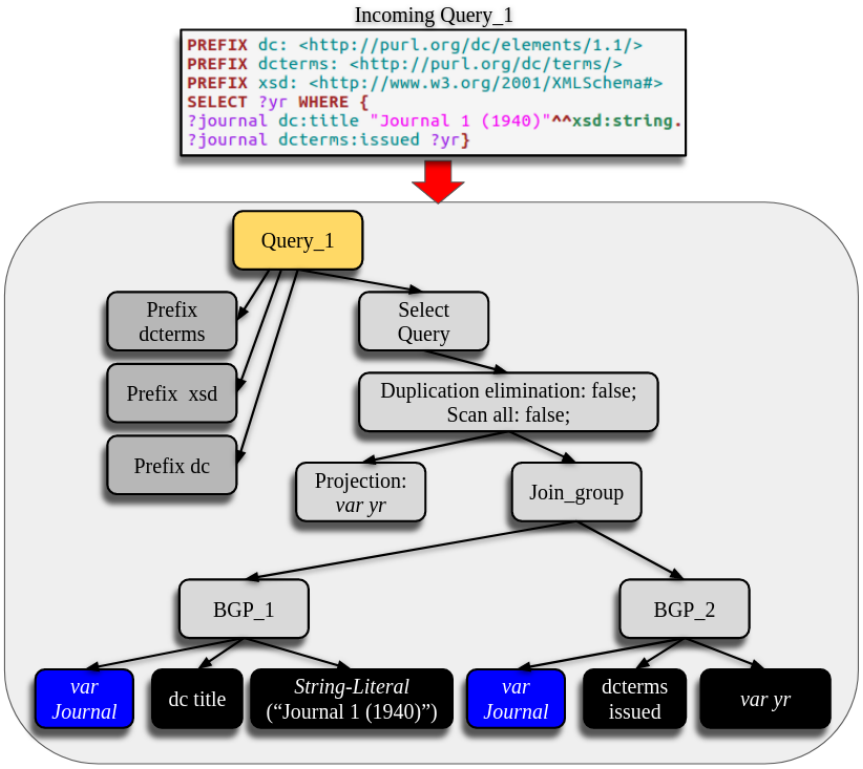


Figure 5.3: A simplified example in which an incoming SPARQL query and its corresponding abstract syntax tree is depicted.

5.2.2 Polyglot Access Management: Query Labeling and Execution

As mentioned in Section 2.4 of Chapter 2, each SPARQL query can be classified into shape-specific categories. At this stage, we confine our focus to the following query types: subject-subject, subject-object, and tree-like queries. Examples of these types are also shown in Fig. 5.2 where each node represents a variable. In general, each query has a source variable as shown in Fig. 5.2 using nodes represented with solid black. Query patterns are typically recognizable by the position of the source variable and the way that it is connected to the other variables in a query (see Section 2.4 of Chapter 2 for more details).

Inspired by [137], we utilize a heuristic-based approach to exploit

the syntactic and the structural variations of patterns in a given SPARQL query in order to label it. On this foundation, SymphonyDB begins by finding the source variable of a given query, looking for all immediate neighbor nodes with one edge distance away. From there, it then iteratively visits nodes further away until all nodes are visited, using a queue data structure to store visited nodes at each stage. Upon finishing the traversal, the query will then be labeled according to the characteristics of its variables and graph. A subject-subject label is applied if all nodes are only immediate neighbors of the source variable with one edge distance away like Fig. 5.2 (a). Fig. 5.3 shows a simplified example in which an incoming SPARQL query is parsed and its corresponding abstract syntax tree is constructed. This query has two triple patterns represented under `BGP_1` and `BGP_2`. It also has two variables (i.e., `journal` and `yr`. Of these, `journal` is the source variable and it is shared between the two triple patterns (represented with two blue boxes) indicating that this is a subject-subject join query. A subject-object label, however, follows a pattern where there is just one outgoing edge from a node at each stage (starting from the source variable) where there is a final node with no outgoing edge as depicted in Fig. 5.2 (b). Finally, queries that contain a combination of both patterns are labeled as *tree-like* (Fig. 5.2 (c)). In addition to the query patterns, SymphonyDB checks the existence of modifiers as well, where modifiers are keywords such as `LIMIT` that are recognizable by the query parser and lexical analyzer implemented in SymphonyDB.

Drawing on our findings in Chapter 4 whereby we identified the most appropriate DMS for each query type, the access management layer employs the following heuristics to select one or more of the integrated DMSs. Queries are routed to both MongoDB and RDF-3X if the following characteristics are present: (1) it contains only a single triple pattern, (2) it is a query with subject-subject joins, (3) it contains no modifiers in the query, and (4) it contains no optional patterns. Alternatively, if a query contains subject-object joins, it is routed to Blazegraph, and finally, all other queries (tree-like queries with or without optional patterns) are routed to Virtuoso. As stated

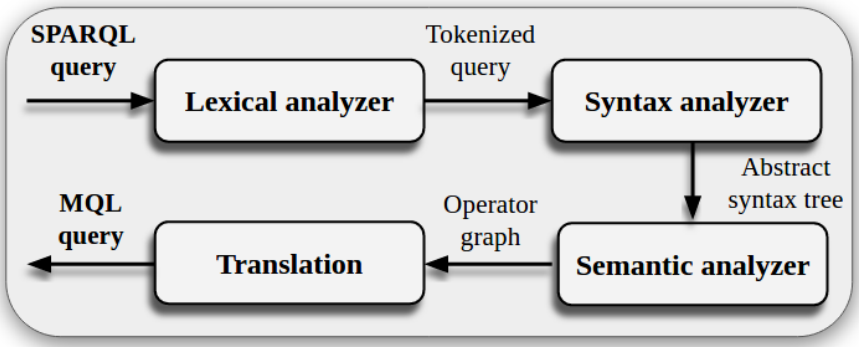


Figure 5.4: The query translation logic flow

previously, as an intermediate step, any queries being routed to MongoDB must run through the JIT query translation to be translated from SPARQL to MQL.

5.2.3 Polyglot Access Management: Query Translation

Query translation provides the extensibility for SymphonyDB to access various DMSs with differing query languages. Currently, the only integrated DMS that requires this functionality is MongoDB as it provides its own query language MQL, which is a Javascript-like, object-oriented imperative language. This contrasts with SPARQL, a domain-specific declarative language [135], which is not supported natively by MongoDB, however, it is feasible to map SPARQL to MQL in most cases. Fig. 5.4 depicts the logic flow of SymphonyDB's JIT query translation, showing that each query is analyzed lexically and tokenized based on the SPARQL query syntax. The lexical analyzer dissects the SPARQL query into logical units of one or more characters that have a shared meaning, often referred to as *tokens*. For instance, "`WHERE`" is a token representing a keyword, whereas "`.`" is an identifier and "`=`" is a sign. In parallel, it parses each query with regard to the grammatical description of the SPARQL language to generate the corresponding syntax tree. The semantic analyzer then produces an operator graph containing information about projection variables, join patterns, conditions, and modifiers. Finally, once the

SPARQL	MQL
Exists (<e1>)	<e1>:{$exists:true}
Not Exists (<e1>)	<e1>:{$exists:false}
(<e1> && <e2>)	{$and:[{<e1>},{<e2>}]
(<e1> || <e2>)	{$or:[{<e1>},{<e2>}]
!(<e1>)	{$not:{<e1>}
(<e1> = <e2>)	{$eq:[{<e1>},{<e2>}]
(<e1> != <e2>)	{$ne:[{<e1>},{<e2>}]
(<e1> > <e2>)	{$gt:[{<e1>},{<e2>}]
(<e1> >= <e2>)	{$gte:[{<e1>},{<e2>}]
(<e1> < <e2>)	{$lt:[{<e1>},{<e2>}]
(<e1> <= <e2>)	{$lte:[{<e1>},{<e2>}]

Table 5.1: SPARQL expressions representation and their equivalent MQL expressions

semantic analysis has been completed, heuristic techniques are used to map the operator graph to MQL and translate the query.

Similar to [134, 135, 138], SymphonyDB maps each SPARQL to MQL using a collection of rules. Table 5.1 shows SPARQL expressions and their equivalent MQL query string, along with an additional set of rules to map SPARQL query patterns to MQL illustrated in Table 5.2. For example, to translate subject-subject join queries, SymphonyDB uses the `$match` aggregation pipeline operator of MongoDB to filter documents and pass a subset of the documents that match the specified condition(s) to the next pipeline stage. It also uses `$lookup` aggregation pipeline operator of MongoDB to translate joins.

Query / Pattern	SPARQL (triple patterns)	MQL (aggregate pipeline)
Single Triple Pattern	"subject" "predicate" "object" ?subject "predicate" "object" "subject" "predicate" ?object	{$match:{ subject_id:"subject", "predicate":"object"}} {$match:{ subject_id:{$exists:true}, "predicate":"object"}} {$match:{ subject_id:"subject", "predicate":{$exists:true}}}
Subject-subject join	{?subject "predicate1" ?object1" . ?subject "predicate2" "object2" .}	{$match:{ subject_id:{$exists:true}, "predicate1":"object1","predicate2":"object2"}}
	{?subject "predicate1" ?object1 . ?subject "predicate2" "object2" .}	{$match:{ subject_id:{$exists:true}, "predicate1":{$exists:true},"predicate2":"object2"}}
	{"subject" "predicate1" "object1" . "subject" "predicate2" "object2" .}	{$match:{ subject_id:"subject", "predicate1":{$exists:true},"predicate2":"object2"}}
Subject-object join	{?subject "predicate1" ?object1 . ?object1 "predicate2" "object2" .}	{{$match:{ subject_id:{$exists:true}}}, {$lookup:{ from: "cole_name", localField: "predicate1", foreignField: "subject_id", as: "join_field"}}, {$match:{ "join_field.predicate2":"object2"}}}
	{?subject "predicate1" ?object1 . ?object1 "predicate2" ?object2 .}	{{$match:{ subject_id:{$exists:true}}}, {$lookup:{ from: "cole_name", localField: "predicate1", foreignField: "subject_id", as: "join_field"}}, {$match:{ "join_field.predicate2":{$exists:true}}}}
	{"subject" "predicate1" ?object1 . ?object1 "predicate2" ?object2 .}	{{$match:{ subject_id:"subject"}}, {$lookup:{ from: "cole_name", localField: "predicate1", foreignField: "subject_id", as: "join_field"}}, {$match:{ "join_field.predicate2":{$exists:true}}}}
Tree-like join	{?subject "predicate1" ?object1 . ?subject "predicate2" ?object2 . ?object2 "predicate3" "object3" .}	{{$match:{ subject_id:{$exists:true}, "predicate1":{$exists:true}, "predicate2":{$exists:true}}}, {$lookup:{ from: "cole_name", localField: "predicate1", foreignField: "subject_id", as: "join_field"}}, {$match:{ "join_field.predicate2":"object2", "join_field.predicate3":"object3"}}}
	{"subject" "predicate1" "object1" . "subject" "predicate2" ?object2 . ?object2 "predicate3" ?object3 .}	{{$match:{ subject_id:"subject", "predicate1":"object1", "predicate2":{$exists:true} }}, {$lookup:{ from: "cole_name", localField: "predicate2", foreignField: "subject_id", as: "join_field"}}, {$match:{ "join_field.predicate3":{$exists:true}}}}
	{"subject" "predicate1" ?object1 . ?object1 "predicate2" ?object2 . ?object1 "predicate3" "object3" .}	{{$match:{ subject_id:"subject", "predicate1":{$exists:true} }}, {$lookup:{ from: "cole_name", localField: "predicate1", foreignField: "subject_id", as: "join_field"}}, {$match:{ "join_field.predicate2":{$exists:true}, "join_field.predicate3":"object3"}}}

Table 5.2: Sample rules used to translate SPARQL queries to MQL

KG	Statistics			
	Sub. (#)	Pre. (#)	Obj. (#)	Triples (#)
Allie	19,227,252	26	20,280,252	94,404,806
Cellcycle	21,745	18	142,812	322,751
DrugBank	19,693	119	276,142	517,023
LinkedSPL	59,776	104	719,446	2,174,579

Table 5.3: Characteristics of the KGs that were used to run the experiments

5.3 Experimental Context and Platform

In this section, we report the experimental setup and details of the KG benchmark datasets that are used in the experimental evaluation in this chapter. This includes detailed information about DMSs' configuration, indexing, data loading process as well as our computational platform. The query performance of SymphonyDB and a range of DMSs are evaluated and presented below.

5.3.1 Evaluation Datasets and Queries

We select four well-known KG datasets with a collection of relevant queries that are publicly available and used in previous studies as well (e.g., [57, 139]). The datasets are as follows. **Allie**[1] is a KG surrounding life sciences, containing abbreviations and long forms utilized within the field. **Cellcycle**[2] contains orthology relations for proteins consisting of ten sub-graphs constituting the cell cycle. In our experiments, however, we integrated all ten sub-graphs into a single KG dataset without modifying any content. **DrugBank**[3] contains bioinformatics and chemoinformatics resources which include detailed drug (chemical, pharmacological, pharmaceutical, etc.) and

[1] Available from: http://allie.dbcls.jp/
[2] Available from: ftp://ftp.dbcls.jp/togordf/bmtoyama/cellcycle/
[3] Available from https://download.bio2rdf.org/files/current/drugbank/drugbank.html

comprehensive drug targets (sequence, structure and pathway information) in the dataset. **LinkedSPL**[4] includes all sections of FDA-approved prescriptions and over-the-counter drug package inserts from DailyMed. Table 5.3 depicts the statistical information related to the above KGs.

We selected 17 representative queries[5]. Table 5.4 shows the classification of the 17 queries (see details of query types in Section 2.4). A range of these queries have also been used in previous studies (e.g., [22, 57, 139]).

5.3.2 Evaluation Platform

Computational Environment. Our benchmark system was a physical machine with a 3.4GHz Core i7-3770 Intel processor, running Ubuntu Linux (kernel version: 4.15.0-91-generic), with 16GB of main memory, 8 cores, 256K L2 cache, 1TB instance storage capacity.

Data Management Systems (DMSs). Our DMSs: (1) column-store Virtuoso (version 07.20.3230), (2) Blazegraph (version 2.1.6), RDF-3X (version 0.3.8), and MongoDB (version 4.2.3). All or some of these DMSs have also been used in previous studies such as [22, 45, 54, 57, 85, 102]. We configured these DMSs based on their vendors' official recommendations. We did not change the default indexing scheme of the DMSs. For MongoDB, we created indexes on those name/value pairs of the JSON representations that were representatives of subjects and predicates.

[4] https://download.bio2rdf.org/files/current/linkedspl/linkedspl.html

[5] All queries are available through https://github.com/m-salehpour/SymphonyDB/tree/master/queries

Benchmark \ Types	Query	SS^{a*}	SO^{b*}	Co^{c*}	OPT^{d*}	Selective	Fil^{e*}	ORD^{f*}	Lim^{g*}	OFF^{h*}	STP^{i*}
Allie	Q1										✓
	Q2						✓				✓
	Q3	✓									
	Q4		✓						✓		
	Q5	✓						✓	✓		
Cellcycle	Q1			✓							
	Q2			✓	✓						
	Q3			✓		✓					
	Q4			✓							
	Q5	✓			✓						
DrugBank	Q1	✓			✓				✓		
	Q2	✓			✓			✓	✓	✓	
	Q3			✓							
	Q4			✓							
	Q5			✓					✓		
LinkedSPL	Q1	✓							✓		
	Q2			✓				✓	✓	✓	

Table 5.4: Types of the queries. SS^{a*}: Subject-subject join, SO^{b*}: Subject-object join, Co^{c*}: combination of SS and SO, OPT^{d*}: Optional pattern, Fil^{e*}: Filter, ORD^{f*}: Order by, Lim^{g*}: Limit, OFF^{h*}: Offset, STP^{i*}: Single triple pattern (no join)

Measurement. The query times for cold-cache are reported in the next section. We dropped the cache before the execution of each query. The output of each query was verified to ensure that output results were correct and consistent across the different DMSs.

5.4 Results

The query execution times over the KGs are presented in Fig. 5.5 where the X axis shows the different queries and the Y axis shows the execution times in milliseconds (**log scale**). These results suggest that RDF-3X offers several orders of magnitude performance advantages over others for queries with a single triple pattern (i.e., no join) and less complex triple patterns (e.g., no optional or complex filtering patterns) such as Allie-Q1 and Allie-Q3-Allie-Q5. However, this DMS could not execute Allie-Q2 as fast as others. Note that RDF-3X could not execute queries with complex triple patterns or offset modifiers (e.g., Fig. 5.5b). In these cases, no value is shown. Virtuoso exhibits around one order of magnitude better performance to run complex queries containing a combination of subject-subject and subject-object joins. As compared to Virtuoso, Blazegraph showed relatively better performance to execute subject-object join queries like Allie-Q4. MongoDB as a document-store could execute all the queries. For subject-subject join queries like DrugBank-Q1, DrugBank-Q2, and LinkedSPL-Q1, its performance is comparable with others.

Our results indicate that the strength of SymphonyDB is that it performs *consistently* across different datasets. Its performance is almost equal to the fastest DMSs in all cases. More specifically, SymphonyDB is consistently the second-best DMS (with a negligible difference as compared to the best DMS) for executing all the queries.

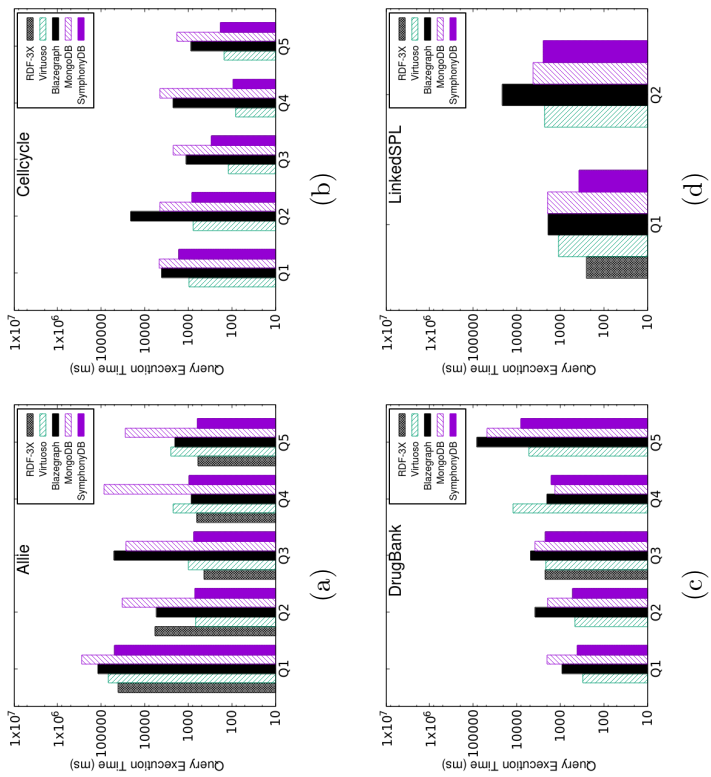


Figure 5.5: Execution time of each query in milliseconds (**log scale**).

5.4.1 Discussion

In this section, we provide a discussion on the experimental results, detailing further insight into the results, and providing key takeaways for SymphonyDB.

5.4.1.1 Analysis: Performance Consistency

Although the results indicate SymphonyDB's consistency across ranging query types, various factors contribute to the performance differences of SymphonyDB with other DMSs. SymphonyDB labels queries based on strict characteristics and heuristics, such as the number of triple patterns, modifiers, optional patterns, and a number of query join patterns. This classification forms the basis for which underlying DMS is selected to route the given query to. For instance, Allie KG queries were routed to RDF-3X and MongoDB (after translation) as two of these queries contained the *single triple pattern*, shown in Allie-Q1 and Allie-Q2, and others contained subject-subject patterns with no modifiers or optional patterns. This analysis is essential in routing queries to the best-performing DMS, even though it imposes minor overheads.

The overhead of query labeling increases with the need for query translation if routing to a DMS that requires this functionality. It is plausible that this overhead influenced the performance of SymphonyDB, justifying the performance difference between SymphonyDB and the best execution time for each query. However, the significance of these overheads can be overlooked due to the consistency in performance across a range of query types exhibited by SymphonyDB. Thus, the overheads observed as a result of labeling and translation are viewed as a small trade-off for added consistency in query execution performance.

5.4.1.2 Insight: SymphonyDB as Common Ground

We have seen in Chapter 4 that efficient query processing across diverse KG query types is moving beyond the limits of a single DMSs. We have seen in this chapter that SymphonyDB enables applications

```
1 SELECT *
2 WHERE {
3 ?subject ?style ?styleName .
4 ?subject type ?typeName .
5 }
```

Figure 5.6: An example SPARQL Query whose predicate is replaced by a variable.

and users to be free from being tied to a single DMS (sometimes referred to as being "locked in" [123]) to store their KG datasets and process diverse queries over them. SymphonyDB as a multi-database helps users to achieve consistent performance across the query types.

SymphonyDB is quite adaptive as new platforms are constantly appearing and applications may emerge with new query types and requirements. Therefore, SymphonyDB is designed to be extensible and dynamic enough to adapt to these constant changes with little effort. For instance, whenever a *new DMS* that displays *better performance* than existing ones in executing a query type becomes available, SymphonyDB can be extended to include the new one as well to ensure that the likely best-performing data platform is always selected for processing each query type. Similarly, SymphonyDB can *include* more DMSs as a response to the emergence of *new query types* or it can *exclude* its existing data platforms to respond to the elimination of some of the existing query types in an application.

The emergence of KG-based applications and their ensuing query types can influence the way query languages have to be designed. In contrast to many conventional DMSs, SymphonyDB decouples KG query processing from query languages. This was one of the driving principles when designing SymphonyDB. For instance, this separation allows users to express their subject-subject join queries in SPARQL and execute them over MongoDB, without being concerned about the query translation. This provides a unified abstraction for data storage and access for multiple DMSs.

5.4.1.3 Limitations

Although performance improvements were observed during experimental conditions, there are a number of hindrances experienced by multi-database environments that still pose challenges to increased performance. Limitations experienced by SymphonyDB include:

i) **Replication of KG datasets** — As multi-database systems employ multiple DMSs, it requires the datasets replicated on each system. The number of replications is determined by the number of DMSs utilized in the underlying layers, e.g. this number is equal to four for SymphonyDB. For write-heavy applications, this replication can lead to increased latency during write operations and therefore decreased write performance. However, most KG applications tend to be *read-mostly* if not *read-only* [44, 60]. Thus, write latency is not a concern in most use cases.

ii) **Efficiency of translations** — All SPARQL queries may not be translated to (efficient) MQL queries due to the *dissimilarity* between the expressiveness of SPARQL and MQL [134]. For instance, triple patterns whose predicates are replaced by variables could not be translated into an efficient query for being executed over MongoDB (in most cases). Fig. 5.6 shows a simple subject-subject join query expressed informally in pseudocode whose predicate is a variable (i.e., `?style`). Currently, SymphonyDB is not able to translate and allocate this query to the likely best-performing DMS. In this research, we did not have such queries and we carefully checked to ensure that our JIT query translation can produce correct and efficient MQL queries for the benchmark SPARQL queries. However, future work entails further optimization and improvements on the translation to ensure the optimality of the query.

5.5 Conclusion

The growing interest in the use of KGs as well as the increases in the size and variety of KG datasets have triggered the development of a range of DMS types, including document, columnar, and graph stores in addition to the relational. Over time, the query workloads executed on these DMS types have also become very diverse. In Chapter 4, we have provided experimental evidence to show that the performance of representative DMS types tends to vary significantly across diverse query types and no single platform dominates performance. From these foundations, we have mapped different types of KG queries to the best-performing DMS under those conditions Chapter 4.

In this chapter, we addressed some of the critical performance challenges in the context of KGs by presenting our extensible prototype, SymphonyDB, as an architecture that can achieve genuine polygloty at the level of access languages and data persistence to classify queries, analyze individual query types and match each to the best-performing platform among Virtuoso, Blazegraph, RDF-3X, and MongoDB as representative DMS types that are included in our prototype at this time. The results of our experiments with the prototype over well-known KG benchmark datasets pointed to the efficiency and consistency of its performance across different query types and datasets. Although a number of limitations are present, the overheads observed are overlooked in favor of increased consistency across a range of queries and become the focus for improvements and future work.

Chapter 6

High-performance Knowledge Graph Query Processing: A Persistent Memory-based Approach

In this chapter, we investigate the effects of emerging hardware devices on RDF storage and indexing strategies. A closer look at the existing RDF data management systems reveals that they trace their roots to architectures and hardware devices from the 1970s. For example, Virtuoso borrowed heavily from object-relational systems, RDF-3X and Blazegraph implemented different variations of the legacy B-tree data structure to create exhaustive indexes for KGs. In general, these systems were architected typically based on the following design choices (i) disk-oriented persistent model, (ii) disk-resident indexes, and (iii) in-memory buffer-pooling to reduce latency. There have been a number of extensions over the past years, ranging from supporting compression and columnar storage to bitmap indexes and vectored execution, to name a few. However, too little attention has been paid to the effects of emerging hardware devices on RDF storage and indexing strategies. This is our focus in this chapter.

6.1 Introduction

As we have discussed in Section 2.5.2, RDF-stores can generally be classified into non-native and native types [80]. Non-native RDF-stores are built on top of existing DBMSs (relational, document-store, etc.) for storing RDF data like [46, 48, 89, 90]. While utilizing non-native systems like the relational DBMSs for RDF data management is feasible in most cases, their performance is often inadequate as discussed in [44, 60, 95]. In contrast, native RDF-stores are customized systems for the storage and retrieval of triples and are built from scratch. These systems are typically designed based on a "*workload-independent*" or *index-everything* storage scheme specific to the *schema-relaxable* (sometimes referred to as *schema-free* or *schema-last*) feature of RDF data model. The performance of native RDF-stores is, in general, considerably better than non-native systems [44].

In the workload-independent storage scheme, indexes and their primary storage locations are critical to achieving good performance. In this scheme, indexes are built over different permutations of the three dimensions that constitute an RDF triple (see Section 2.5.4.2). The data structure that is used to implement these indexes along with the speed with which these indexes can be read from the primary storage device play pivotal roles in the performance of native RDF-stores.

Employing *read-optimized* data structures such as B-trees, sorted vectors, and hash maps have been the common choices for indexing since most RDF datasets are *read-intensive* if not totally *read-only* [44]. For the primary storage locations, disks (e.g., HDD or SSD) and main memory (DRAM) are the typical options. Disk-based devices only support bulk data transfers as blocks which tend to be slow [61, 62]. This means that even for reading a single byte of data stored on an SSD, a block of data (typically 4 KB) will be transferred. This adversely affects random access to the indexes. To decrease this access latency, an in-memory cache for blocks of triples is typically maintained along with maximizing the number of sequen-

tial reads and writes. In this case, special components are added to data management systems which inevitably impose additional overheads. In contrast, a single byte can be quickly read from DRAM but all data is lost in the event of a power failure. DRAM also consumes more energy since it requires periodic refreshing to preserve data even if it is not actively used. Its energy consumption can increase by up to 40% of the overall power consumed by a server [61, 140]. Based on this, DRAM cannot be solely used as the primary storage of indexes due to its *volatility*, *high cost*, and *limited capacity* [64].

Fortunately, emerging storage technologies such as Phase Change Memory are reducing the fundamental gaps between main memory and disk. Specifically, Intel's Optane DC Persistent Memory Modules (Optane DC PMM) offers an appealing blend of the best properties of DRAM and disk. This Persistent Memory (PM) is durable like disk and directly byte-addressable like DRAM [63, 141, 142, 143, 144, 145, 146]. PM enables us to directly and quickly persist any data structure without the overhead of a file system [147]. PM's *price*, *capacity*, and *latency* lie between DRAM and SSD [64, 65]. This allows us to allocate larger spaces for holding indexes without the need for reconstruction after a crash [66]. To the best of our knowledge, efficient indexing architectures for RDF data to better leverage PM technology are not available at this time. A number of PM-based indexes are currently available, with FAST&FAIR [148], bzTree [149], DPTree [150], RNTree [151], FPTree [152], NV-Tree [153], wB+-tree [154], and CDDS-Tree [155], being a few of the most recent proposals. However, their main focus is on reducing the write cost when updating the indexes [156]. This is not aligned with the rapid read performance requirement of RDF data indexes. Our goal is to propose an efficient indexing structure that is tailored to the characteristics of RDF data and PM as the primary storage hardware.

We introduce RDFix, a specialized architecture for indexing RDF data from the ground-up in which all data persist on PM. RDFix aims to provide a high-performance and flexible access path by combining old and new techniques such as employing a multi-layered design inspired by [60] and the implementation of complete binary

trees using static arrays. RDFix can be used to index the RDF data in all the six possible ways, one for each ordering of the three RDF elements. Given that most RDF datasets tend to be *read-mostly* if not *read-only* [44, 60], RDFix employs two layers of static arrays to store keys and a layer of lists to store associated values (see Section 6.2). For instance, in the *spo* order, RDFix stores the prefix"*sp*" as the key and "*o*" as the associated value. Utilizing a static array-based architecture enables RDFix to retrieve each key in constant time with minimum overhead. Typical of most access methods, RDFix can be deployed as a stand-alone indexing structure, or embedded in an RDF-store.

In this chapter, Section 6.2 presents the design and architecture of RDFix including operations that it supports such as lookup. This proceeds by providing experimental validation of RDFix where we demonstrate its advantages for indexing diverse RDF datasets compared to major read-optimized architectures such as B-trees, sorted vectors, and hash maps over PM. Our experimental analyses show that RDFix outperforms these indexing structures by at least a factor of 3 for performing lookup operations. These results also suggest that RDFix is the superior access method for performing scan operations as it has the lowest overhead and the highest throughput. A discussion of the space-time tradeoffs is also included. Needless to mention, this superior performance is achieved by trading off more memory (space). Finally, Section 6.5 concludes this chapter.

6.2 RDFix: Design and Architecture

We start with an overview of RDFix's architecture. More details of the operations that RDFix supports are included in Section 6.2.1. A detailed explanation and rationale for the design choices in RDFix are presented in Section 6.2.2.

RDFix is designed and developed for indexing RDF data in the six combinations referred to earlier, one for each possible ordering of the three RDF elements. We use "*spo*" indexing order as our running example for the rest of this chapter. For instance, in this order, the

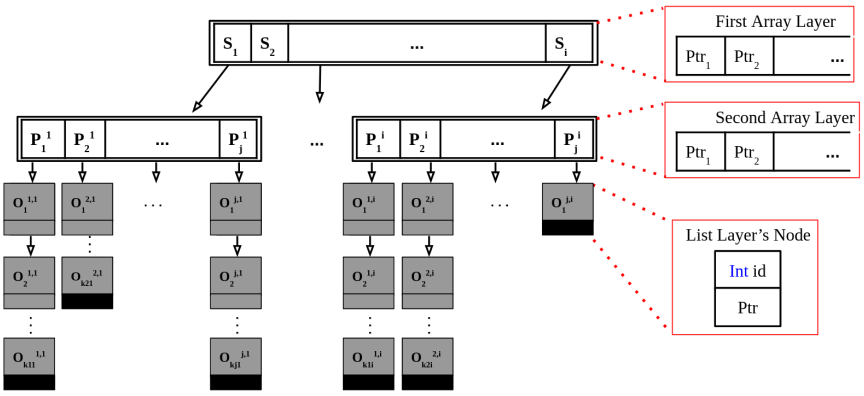


Figure 6.1: Overview of RDFix's structure for *spo* indexing order. Layouts of the arrays and each list node are also shown in the red boxes.

index structure centers around s (subject) playing the role of the header to access its associated elements p (predicates) followed by o (objects). Fig. 6.1 depicts the RDFix architecture for this order. This shows that a number of arrays and lists are used to store predicates and objects (respectively) that are associated with each subject in the first layer array.

We allocate memory on PM for the first and second layer arrays. More details about the space consumption of each layer are shown in Fig. 6.2 in which the first layer's array length is at least equal to the number of unique subjects in the RDF data (❶ in the figure). Each cell of this array points to an array in the second layer (i.e., ❷) whose length is equal to the number of unique predicates in the RDF data (see ❸). Based on this, the total number of allocated arrays in the second layer is equal to the number of unique subjects (❹). Each cell of an array in the second layer itself points to a list in which all the associated objects are stored (❺). Each list is a single linked-list in which each node stores an object along with an 8-byte *next* pointer to point to the successive node if multiple objects are associated with a subject-predicate pair. As a result, the number of nodes in each list is equal to the number of objects that are associated with each subject-predicate pair (❻). As shown in the list layer, the total number of nodes across all lists is equal to the total number

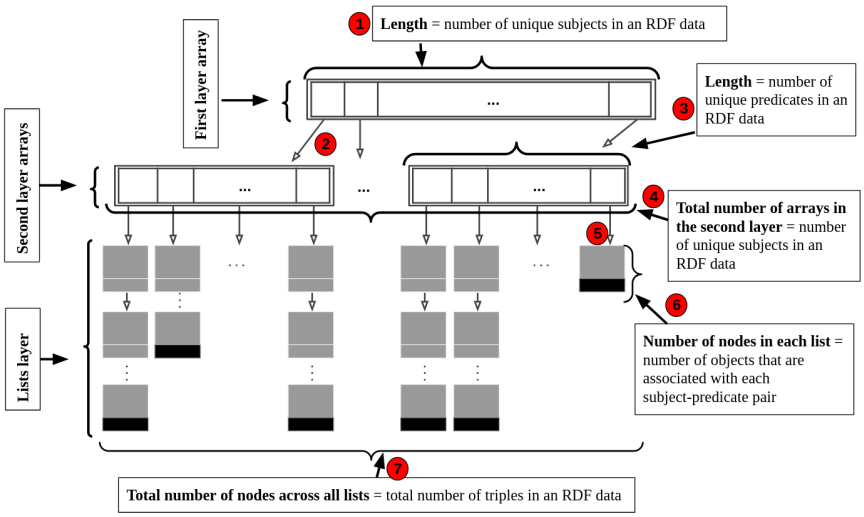


Figure 6.2: Space consumption of RDFix for *spo* indexing order.

of triples in the RDF data (see 7). Analogous structures can be materialized for each of the other five indexing structures.

To allocate memory on PM for the first and second layer arrays, we assume that the total number of unique subjects and predicates are known a priori (otherwise, upperbounds need to be estimated). After the allocation of memory for the arrays, the triples can be stored. To this end, we sort the RDF dataset and employ a *dictionary encoding technique* similar to that of [41, 44, 60, 157] for *mapping string* literals of RDF elements to sequential integer *ids*. The benefits of this mapping include: (1) compressing the index structure on PM and (2) providing a simplification for operations such as lookup since matching *ids* is typically faster than text matching. In addition, the associated *ids* are used to directly access the arrays' cells in constant time. For instance, if a subject is mapped to n (as integer *id*), then the n^{th} cell in the first layer array stores this subject. This cell then plays the role of the header to access its associated elements (e.g., predicates and objects in Fig. 6.1). The same technique is used to directly access the cells of the second layer arrays in which each array contains multiple pointers. These pointers point to a list node storing associated values (objects). After the memory allocation on

PM, inserting a node in the list layer involves just modifying the 8-byte next pointers. RDFix allocates the space for each list node through The Persistent Memory Development Kit (PMDK)[1] which is the de-facto standard toolkit to interact with and allocate memory on PM.

As the collation order in each of the six indexing orders is different (e.g., spo, sop, osp, etc.), we use a generic terminology for referring to the different RDF elements of each indexing order, for the rest of this chapter. For instance, we use v1, v2, and v3 to represent the first, second, and third elements (respectively) of each indexing order. For instance, v1, v2, and v3 will represent predicate, subject, and object values respectively in the *pso* order.

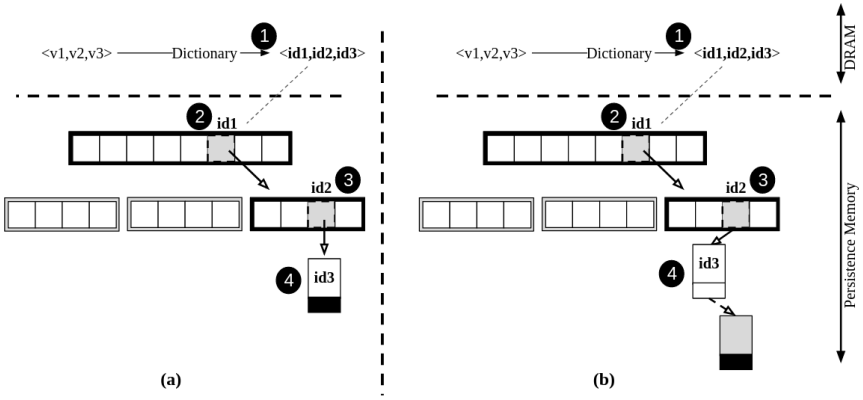


Figure 6.3: Process of *insert* operation.

6.2.1 Operations

We describe the operations, namely, *insert*, *lookup*, *range scan*, *delete*, and *update* in this section. These operations are sometimes referred to as "*interfaces*" or "*standard*" key-value operations [149].

6.2.1.1 Insert

This operation (sometimes referred to as *put*) stores a triple of the form <v1,v2,v3> where v1 and v2 can be viewed as a *key pair* and v3

[1] http://pmem.io

as the associated value. Fig. 6.3 (a) shows the steps to insert a new triple where the elements of the given triple of the form <v1,v2,v3> are first mapped to integer *ids* of the form <id1,id2,id3> (respectively) using the dictionary encoding technique (❶). After this, id1 is used to access the first layer array's cell (representing v1) (❷) and id2 to access the associated array's cell (representing v2) in the second layer (❸), pointing to the list layer. We create and insert a new node into the list layer to store id3 (❹). Fig. 6.3 (a) shows the case in which id3 is the first value that is stored and associated with the key pair <id1,id2>. Fig. 6.3 (b) shows the other case in which id3 is not the first value that is associated with the key pair <id1,id2>. In the second case, steps ❶ to ❸ are still the same as the previous case. In contrast, we place the newly created node as the first node in the existing list (❹). To achieve this, we set the *next* pointer of the newly created node to point to the first node of the list and update the pointer in the predecessor array's cell to point to the newly created list node (❸).

When a triple is inserted, it is not immediately written persistently to the PM device. Instead, it is buffered in the regular on-CPU cache. To store it persistently on PM, the triple should be flushed out from the CPU cache to the PM. In RDFix, two flushes are needed for each insertion operation. One is to persist the newly created list node (including the next pointer that links it to the potential next list node). The other one is to persist the pointer of the predecessor array's cell in the second layer (to make the newly created node accessible).

Given that most RDF datasets tend to be *read-mostly* if not *read-only* [44, 60], RDFix utilizes a concatenation of static arrays as its main data structure. This imposes minimum overhead on RDFix to insert the triples. In this context, the overhead (sometimes referred to as "*structural modification operations*") refers to inevitable operations internal to the indexing architecture to ensure that the invariants of the underlying data structure hold, or to improve performance. For instance, when the nodes in a B-tree overflow (e.g., during insertion) or underflow (e.g., during deletion), node splits

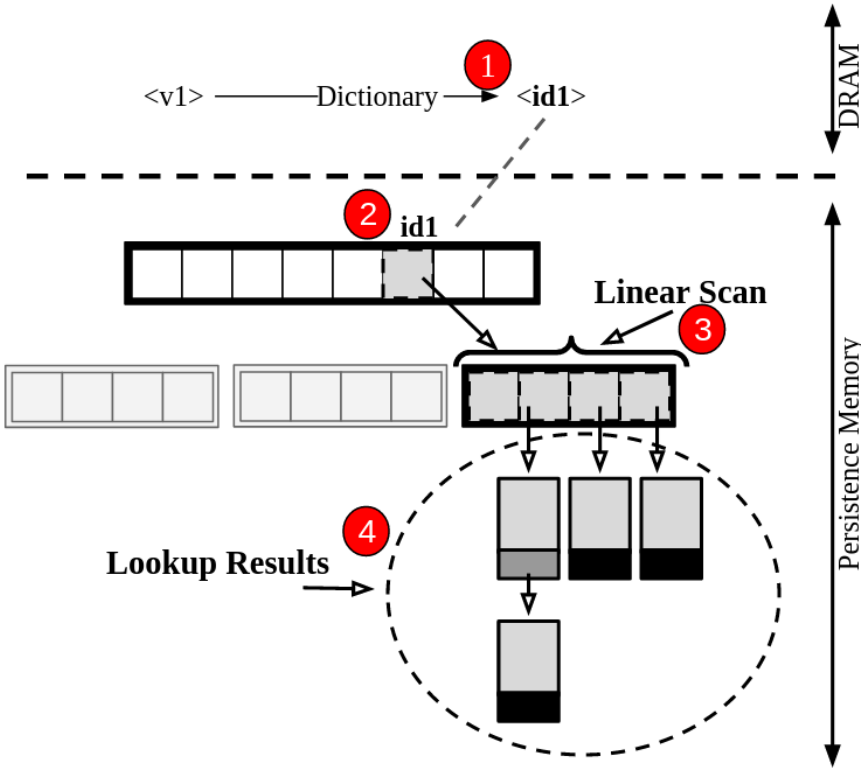


Figure 6.4: Process of *lookup* operation for a given single key <v1>.

or merges are inevitable internal operations to re-establish the invariants of a B-tree. In other data structures like hash maps, rehashing is the inevitable internal operation to keep the average cost per lookup constant as the data size grows. However, the concatenation of static arrays to store each key pair provides RDFix with minimal structural overhead when inserting triples.

6.2.1.2 Lookup

We consider two common scenarios for performing lookup operations. The first is to retrieve all associated values with a given *key pair* <v1,v2> while the second is to search for all associated values with a given single key <v1>.

In the first scenario, a key pair of the form <v1,v2> is given as input. To perform *lookup* operation, we map this pair to integer *ids* of

the form <id1,id2> using the dictionaries. We then find the related entry in the first layer array using id1 through which we can access the associated array in the second layer where the pointer to the target list can be fetched. If the key pair has already been inserted, the associated values will be returned by performing a sequential scan on the unsorted list space. Otherwise, *NULL* will be returned. In the second scenario, the given key, <v1> (as the input) is mapped to an id (e.g., id1) using the dictionary as shown in Fig. 6.4 (1). We find the related entry in the first layer array (2) through which we can search all of its associated values by traversing the array in the second layer (3) and following the pointers to the all target lists to return all the associated values (4).

6.2.1.3 Range Scan

This operation typically takes two key pairs $k1$ and $k2$ as inputs and returns all associated values if their keys are within the specified range (i.e., |k2 - k1|). There are similarities between performing a range scan and the lookup operations. A range scan is implemented using an iterator. We first perform a lookup operation to find the starting entry at $k1$ and then traverse the associated list layer to return all stored values. We continue probing in an incremental way to the next key pair in the sequence and return associated values until the given range is scanned, or the user terminates the iterator. RDFix supports both forward and backward range scans. In the case of backward range, the next key pair in the sequence is probed in a decremental way instead.

6.2.1.4 Delete

This operation removes the *value* associated with a specified key pair from the index (we assume that the deletion of the key pair itself is rarely needed). Fig. 6.5 (a) shows the required steps to perform this operation. Similar to the *lookup* operation, we first use id1 and id2 as the key pair to locate the the predecessor array cells (steps ❶ to ❸). We then delete the target list node, ❹ in Fig. 6.5 (a). After

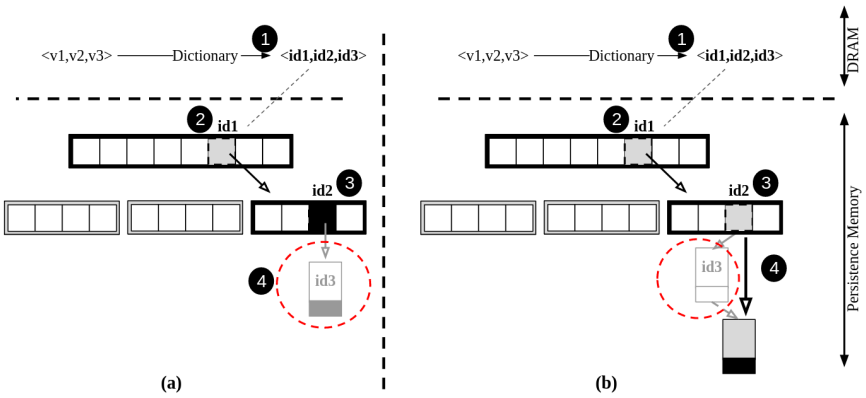


Figure 6.5: Process of *delete* operation.

this, we update the pointer in its predecessor cell (❸) to point to NULL. In the case that more than one value is associated with the key pair, we need to modify the *next* pointer of nodes in the list. For instance, if the targeted node is the first node of the list we modify the *next* pointer of its predecessor to the *next* node and then delete the targeted entry, as shown by ❹ in Fig. 6.5 (b).

6.2.1.5 Updates

Following [44], the four assumptions in designing the update operation for RDFix are: (1) when RDF datasets are bulk-loaded, queries are far more frequent than updates, (2) updates are mostly insertions of new values, (3) overwriting existing keys are rarely needed, and (4) updates can be staged, batched, and performed as incremental loading. As discussed in [44, 60], it is reasonable to make these assumptions. Based on this, RDFix can be viewed as a read-optimized index architecture with good support for efficient online updates of associated values with existing key pairs like adding new values, deleting, or modifying previously stored ones. If the given key exists in the indexing architecture, RDFix performs an update to that key. This is the common approach to adding new values in most indexing architectures and is sometimes referred to as "*upsert*" operation [149]. Nonetheless, depending on the user requirement, delete and insert operations can be combined to replace a value.

Due to the aggressive static array-based architecture that RDFix uses for fast querying, online updating of *key pairs* after the initial index creation (once the RDF dataset is bulk-loaded) will be part of our future work. Inspired by [44], we plan to support direct updates to key pairs by means of a staging architecture where these updates are deferred, and instead applied to compact differential indexes and later merged into the main indexes in a batched manner.

6.2.2 Rationale

Space utilization. RDFix places all indexes on PM and only the mapping dictionaries in DRAM. A copy of these dictionaries is also placed on PM for recovery purposes. As a result, it may consume extra DRAM space. However, the benefits of replacing all string literals of RDF elements by *ids* include: (1) it compresses the index structure on PM, and (2) it provides a simplification for operations such as lookup since matching *ids* is generally faster than text matching. Employing two mapping dictionary indexes is a small cost to bear. During operation translation, the literals occurring in the request are translated into their dictionary *ids*, which can be done with a standard ordered list from strings to *ids*. After processing the operation, the resulting *ids* have to be transformed back into literals as output to the user.

Performance. Apart from the above-mentioned benefits, the use of the dictionary mapping technique enables us to utilize static arrays and provides RDFix with the constant access time when looking up triples. The related benefit of using static arrays is high cache locality since the entries occupy contiguous memory locations. In addition, there is no need to shift the entries of the array layers on PM when performing insert/delete operation as it is often needed in *dynamic* data structures like B-tree (e.g., by merging or splitting tree nodes to keep it balanced).

In RDFix, the static arrays to store key pairs play the role of headers to access associated values in the list layer. The element-based storage of associated values using a linked-list provides RDFix

with a simple yet concise and efficient way of handling multi-valued RDF elements. The list nodes own their separate memory locations on PM and operations on their stored values can be performed by simply manipulating the *next* pointers.

Durability. RDFix persists all data items on PM (as well as the dictionaries) to preserve it across system failures. It does not need to employ a specific recovery algorithm. RDFix instead relies on the recovery algorithms of the persistent memory allocator (i.e., PMDK) to avoid persistent memory leaks and ensure recoverability.[2] In short, RDFix first requests the allocation of a memory chunk. The allocator updates the chunk's meta-data to indicate that it has been allocated and returns it (to RDFix). RDFix owns the memory only after the allocator has successfully persisted the address. This design is already employed by many existing PM-based access methods like [149].

6.3 Experiments

We have implemented RDFix in C and assessed its performance.[3] In this section, we describe the experimental setup (in Section 6.3.1). We proceed by evaluating the performance of RDFix in comparison with the widely-used indexing architectures for RDF data, namely, B-tree, hash map, and sorted vectors (Hexastore) in Section 6.3.2. A discussion of space-time tradeoffs between these indexing architectures is included in Section 6.3.3.

6.3.1 Setup

Hardware Platform. We ran our experiments on a machine with the *actual* PM (not prototyped or simulated), i.e., Intel Optane DC

[2]Please note that creating a safe and correct persistent memory allocator (as discussed in [147]) is outside of the scope of this thesis.

[3]The source code and data have been made available at https://github.com/m-salehpour/RDFix.

		Description
CPU	Type	2x Intel Xeon Gold 6230 (3.9 GHz)
	# of cores/threads	20/40
	Caches	L1: 1.3MB Icache, 1.3MB Dcache
		L2: 40MB, L3: 55MB
MEM	PM Capacity	1TB (8 modules, NMA1XBD128GQS, 2.6GHz)
	PM Read Latency	1399 ns (rnd/4K/1job/1 PM-Module/lat-avg)
	PM Write Bw	2211 MB/s (rnd/4K/1job/1 PM-Module)
	DRAM Capacity	128GB
OS	Release Version	Ubuntu 20.04.2 LTS
	Kernel	5.4.0-73-generic

Table 6.1: Description of the evaluation platform.

Persistent Memory[4] and the second-generation Xeon Scalable processors. Optane DCPMMs were configured in 100% AppDirect mode (aka, DAX) in which software packages have direct byte-addressable access to PM [158]. Table 6.1 reports the relevant details of the test machine. The eight Optane DCPMM modules were evenly attached to two CPU sockets (i.e., each CPU socket owns four modules). We used two of these modules attached to the same CPU socket in our experiments. As part of our evaluation, applications accessed PM by using a PM-aware file system (e.g., XFS-DAX) to manage the PM device. We relied on the basic interfaces of PMDK (e.g., pmemobjalloc()) to allocate the PM space. We allocated PM space only from each local PM device since [159] has shown that cross-NUMA accesses to Optanes impact overall performance negatively.

Baseline Systems. We compared RDFix with other read-optimized indexes, namely, B-tree[5], hash map[6], and sorted vectors (Hexastore [60]). These systems were selected because the use of data structures like sorted vectors and B-trees for indexing the RDF data is widely accepted as contributing to good performance at this time [114]. We used the PM-based implementations of B-tree and hash map pro-

[4] https://www.intel.com.au/content/www/au/en/products/memory-storage/optane-dc-persistent-memory/optane-dc-128gb-persistent-memory-module.html (April 2021)

[5] https://github.com/pmem/pmdk/tree/master/src/examples/libpmemobj/tree_map (April 2021)

[6] https://github.com/pmem/pmdk/tree/master/src/examples/libpmemobj/hashmap (April 2021)

Name	Scale	#Sub	#Pre	#Obj	#Tri	Stru
BSBM	100K	9,875	40	24,220	97,795	0.94
	1M	83,741	40	166,897	892,853	0.94
	10M	808,154	40	1,085,287	8,798,389	0.95
LUBM	100K	17,175	18	13,948	100,545	0.98
	1M	183,426	18	137,923	1,124,616	0.97
	10M	1,728,834	18	1,287,846	10,629,578	0.98
SP2Bench	100K	19,369	58	50,267	100,073	0.73
	1M	187,066	67	422,646	1,000,009	0.76
	10M	1,730,250	77	2,558,828	10,000,457	0.71

Table 6.2: Statistics of the benchmark datasets. Sub stands for subjects, Pre: predicates, Obj: objects, Tri: triples, and Stru: structuredness.

vided as open-source by Intel. Their code is used without changing the default settings like the order and the minimum number of keys per node in the B-tree and hashset thresholds and coefficients in the hash map. Please note that most native RDF-stores use the B^+-tree, a B-tree variant where the data is stored entirely in the leaves and the internal nodes hold pointers to the leaf nodes. We used B-tree since employing B^+-tree is useful in a block-addressable disk-based setting but not in the byte-addressable devices like main memory or PM as discussed in [160]. We also compared the performance of RDFix with that of Hexastore. Two layers of sorted vectors and a layer of lists were utilized by Hexastore to index RDF data (more details in [60]). To the best of our knowledge, its source code is not publicly available. We implemented a PM-compatible version of Hexastore based on the original design presented in [60]. The *single-threaded* version of all the four systems is used in our experiments.

Benchmark Datasets. We generated datasets using BSBM [54], LUBM [161], and SP2Bench [162] packages ranging from 100K triples (around 50MB on disk) to 10M triples (around 2GB on disk) to illustrate the scale effects on the performance of different operations. Table 6.2 reports the relevant statistics on these datasets including the number of unique subjects, predicates, objects, and the total number of triples as well as their *structuredness* measures. These datasets display different levels of *structuredness*, an RDF data-related con-

cept introduced in the previous chapters (see Section 2.2.3). We included more details about structuredness and discussed its relationship to space consumption of RDFix as part of the performance evaluation of *insert* operation in Section 6.3.2.1.

6.3.2 Measurement and Results

We conducted experiments to analyze the overall performance of RDFix and the other systems. This is based on operations that are explained in Section 6.2.1. Execution times for all the operations were averaged over 5 consecutive runs. Geometric mean[7] of the times are reported in this section.

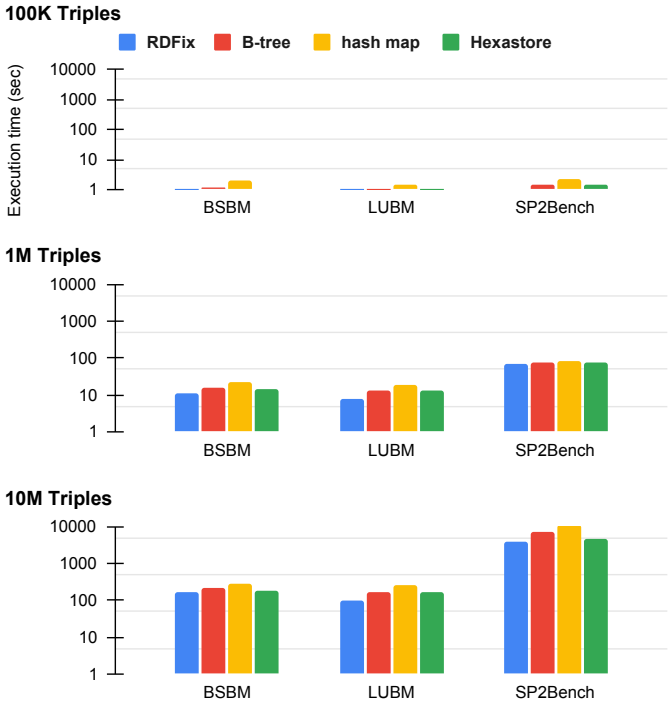


Figure 6.6: Execution times (log scale) of *insert* (aka, *put*) operations for the different datasets and scales.

[7] We used Geometric mean – the nth root of the product of n numbers – instead of the arithmetic mean since it typically provides a more accurate reflection of the total speedup factor [89].

6.3.2.1 Insert

To perform *insert* operation, we first need to allocate memory on PM for the first and second layer arrays. We assume that the total number of unique v1 (i.e., the total number of unique subjects in *spo* order as our running example) and the total number of unique v2 (e.g., the total number of unique predicates) are known a priori (otherwise, upperbounds need to be estimated). Following the memory allocation, triples can be stored by performing the *insert* operation. We computed the execution time by storing all triples of an RDF dataset which can be viewed as performing a *bulk load* operation. This is an end-to-end time that is computed from the time of reading the first triple from an RDF/N-Triples input file (which is also stored on PM) to the time of storing the last triple on PM. For RDFix, the time consumed for the initial memory allocation is also included. Fig. 6.6 shows the execution times (log scale) of the four systems to store triples using our running example (*spo* order) at different scales. The X axes show the datasets and the Y axes show the execution times in seconds (log scale). As can be seen, RDFix is faster than the other systems. Hexastore has the closest execution time (up to 30% slower) followed by B-tree and hash map (over 2x and 3x slower than RDFix, respectively). This trend held at different scales.

The structural overhead of dynamic data structures (like B-trees) plays a major role in the performance of the insert operation. This overhead (sometimes referred to as structural modification operations) refers to inevitable operations internal to the indexing architecture to ensure that the invariants of the underlying data structure hold or to improve performance. For instance, when the nodes in a B-tree overflow (e.g., during insertion), node splits are inevitable internal operations to re-establish the invariants of the indexing architecture. In hash map, re-hashing is the internal operation to keep the constant average cost per further lookup when data size grows. In contrast, the concatenation of static arrays to store *key pairs* enabled RDFix to achieve minimal structural overhead. As mentioned

earlier, we used Intel's implementation of B-tree and hash map without changing the default settings like the order and the minimum number of keys per node in B-tree and the hashset thresholds and coefficients in hash map. Tuning these settings can affect the insert operation performance but it may impose a cost on the lookup operation. Based on this, we decided to not change the default settings. In addition, we sorted the N-Triples input file (in the order of *spo*) before performing the *insert* operation which helped almost all systems to uniformly exhibit faster execution times.

Subsequent to storing all triples of the datasets, we evaluated the space consumption of the four systems on PM. Consistent with our expectations, RDFix consumed more space than the other systems. RDFix used around 3085KB to store the keys of the BSBM-100K dataset compared to approximately 1100KB for hash map, 940KB for B-tree, and 750KB for Hexastore. The space consumption of RDFix depends on the total number of unique v1 and v2 (i.e., #v1 × #v2). Based on this, in the *spo* order, the space consumption of RDFix depends on the total number of unique subjects and predicates (i.e., #subjects × #predicates). For the other systems, this depends on the number of unique v1-v2 pairs in an RDF dataset. The space that is consumed for storing dictionaries and associated values is the same across all the systems and is not included. RDFix used around 4x more space than the most compact one in storing BSBM-100K. This difference is approximately 3x and 9x in storing LUBM-100K and SP2Bench-100K, respectively.

It turns out there is a relationship between the structuredness of a dataset and the space consumption of RDFix. As defined in [50], the structuredness of a dataset D with respect to a type T is determined by how well the instances in D conform to type T. For example, an academic dataset may contain different types such as "Professors", "Students", and "Courses". It may have "Professor1", "Student101", and "COMP100" as instances. If each instance in D sets values for most (if not all) of the predicates of T, then all the instances in D have a similar structure that conforms to T. In other words, the structuredness of T is affected by the *sparsity* of its predicates across

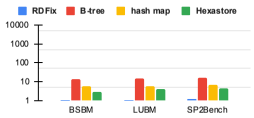
instances. Any RDF dataset's level of structuredness can be quantified by a normalized value in the [0, 1] interval, with values close to 0 corresponding to low structuredness, and 1 corresponding to high structuredness. In an RDF dataset with high structuredness, we expect that for any two instances of the same type, the instances have exactly the same predicates. Based on this, the more structured the dataset, the less the difference between the space consumption of RDFix and the other systems.

6.3.2.2 Lookup

This operation takes a key and returns the associated values if the key has already been inserted. Otherwise, *NULL* will be returned. We consider two common scenarios for performing the lookup operation. The first is to retrieve all associated values with a given *key pair* <v1,v2> while the second is to look for all associated values with a given single key <v1>. Based on this, we evaluated the performance of the *lookup* operation as follows: (i) performing 1M, 10M, and 100M individual lookup operations in which both <v1,v2> are randomly generated, and (ii) performing 1M, 10M, and 100M individual lookup operations in which only <v1> is randomly generated. More details about these two scenarios were presented in Section 6.2.1.2.

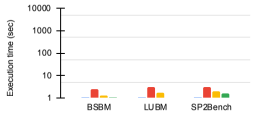
Fig. 6.7 (a)-(c) show the execution times over 100K triples. Fig. 6.7 (d)-(f) and Fig. 6.7 (g)-(i) show the execution times over 1M and 10M triples respectively. RDFix was faster than the other systems. Hexastore had the closest execution times (up to 4x slower) followed by hash map and B-tree (approximately 7x and 16x slower than RDFix, respectively). The higher differences between execution times were observed when higher number of operations were performed like 100M lookup operations in Fig. 6.7 (c) versus 1M operations in Fig. 6.7 (a). The execution times were increased at larger scales for all the systems. However, the performance differences and the trend remained almost unchanged, e.g., Fig. 6.7 (a) versus Fig. 6.7 (g).

(a) 1M operations over 100K triples

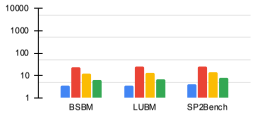


(b) 10M operations over 100K triples

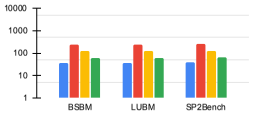
(c) 100M operations over 100K triples



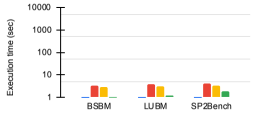
(d) 1M operations over 1M triples



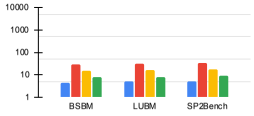
(e) 10M operations over 1M triples



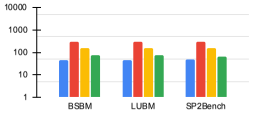
(f) 100M operations over 1M triples



(g) 1M operations over 10M triples



(h) 10M operations over 10M triples

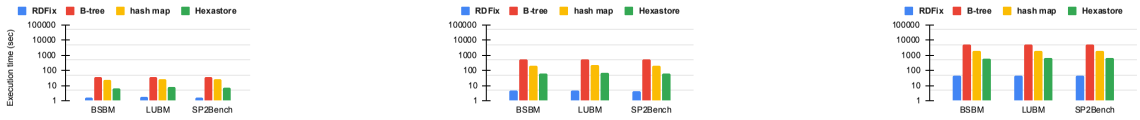


(i) 100M operations over 10M triples

Figure 6.7: Execution times (log scale) of 1M, 10M, and 100M individual *lookup* (aka, *get*) operations for randomly generated <v1,v2> pairs at scale.

It is clear that the minimal overhead of chasing pointers as well as the efficient use of CPU cycles contributed to RDFix's superior performance to perform lookup operations. To measure this, we added the *clock* function of C to the systems' source code to measure the processor time consumed by the execution of the lookup operations. The *clock* function returns clock ticks that are units of time of a constant but system-specific length. It computes the number of clock ticks elapsed since an epoch related to the execution of the lookup operations for each system. Based on this, RDFix exhibited the most efficient use of CPU cycles across all the systems. We also double-checked the correctness of this measurement by using the *perf* tool.

Fig. 6.8 shows the execution times of different systems to perform *lookup* operations for randomly generated <v1> at different scales. Fig. 6.8 (a)-(c) show the execution times in seconds (log scale) over 100K triples. Fig. 6.8 (d)-(f) and Fig. 6.8 (g)-(i) show the execution times over 1M and 10M triples respectively. Similar to the first scenario, RDFix is faster than the other systems followed by Hexastore with the closest execution time along with hash map and B-tree holding the third and fourth places, respectively. In contrast to the first scenario, the performance gap is higher. Hexastore is up to 15x slower than RDFix. Hash map and B-tree are even slower as can be seen in Fig. 6.8 (c). The execution times were higher at larger scales for all the systems. However, the performance differences and the trend remained the same, e.g., Fig. 6.8 (a) versus Fig. 6.8 (g).



(a) 1M operations over 100K triples

(b) 10M operations over 100K triples

(c) 100M operations over 100K triples

(d) 1M operations over 1M triples

(e) 10M operations over 1M triples

(f) 100M operations over 1M triples

(g) 1M operations over 10M triples

(h) 10M operations over 10M triples

(i) 100M operations over 10M triples

Figure 6.8: Execution times (log scale) of 1M, 10M, and 100M individual *lookup* (aka, *get*) operations for randomly generated <v1> at scale.

We have already pointed out that the efficient use of CPU cycles and pointer chasing played major roles in the better performance of RDFix. In addition, allocating contiguous memory locations for storing entries had positive effects on performing lookup operations under the second scenario. This design can take advantage of a higher cache locality with fewer cache misses. We compared the number of cache misses while performing the lookup operations. RDFix showed the minimum number of misses followed by Hexastore (e.g., with around 3x more LLC-load-misses) along with B-tree and hash map (over 6x more LLC-load-misses). We gathered performance counter statistics of the systems (the average of 5 successive runs after compiling the C code with *-O3* parameter) to measure the misses. We used the `perf` tool with `stat -a -e LLC-loads -e LLC-load-misses -e LLC-stores -e LLC-store-misses` events along with `taskset` command to set the CPU affinity and avoid cross-NUMA accesses as well as `nice` command to ensure that the Linux kernel launches the systems with higher scheduling priority to prevent from potential negative effects of context switching on the cache misses. We also qualified the events with `:u` (e.g., `LLC-load-misses:u`) to measure the misses without taking kernel-mode events into account. The advantage arising from the cache spatial and temporal locality (due to array-based design) is a factor in RDFix's performance.

6.3.2.3 Range Scan

This takes two keys k1 and k2 as inputs and returns all associated values where their keys are within the specified range (i.e., |k2 - k1|). Fig. 6.9 shows the execution times of the systems to perform 100M individual *range* scan operations for randomly generated range [k1, k1+1000] (using the *spo* order) at different scales. We searched the systems for a random key k1 (similar to the second lookup scenario) and then retrieved an additional 1000 keys as our range. As already discussed, factors like occupying contiguous memory locations (higher cache locality with fewer cache misses), the efficient use

(a) over 100K triples (b) over 1M triples (c) over 10M triples

Figure 6.9: Execution times (log scale) of 100M individual *range scan* operations for randomly generated range [k1, k1+1000] at scale.

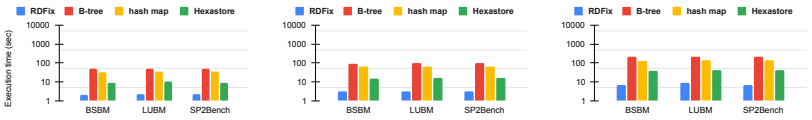
of CPU cycles, and fewer pointer chasing helped RDFix to achieve faster execution times to perform range scans.

In addition to the above-mentioned factors, *sequential access* to PM for performing range scan operations contributed to the RDFix's better performance. This is consistent with findings of other studies like [163, 164]. More details of the effects of sequential access to PM can also be found in [64]. Moreover, Intel mentioned[8] that the Optane DIMM adopts an on-DIMM buffer structure and [165] showed the positive effects of this buffer on the read latency. Based on this, we speculate that the built-in buffer structure of PM helped RDFix to perform range scan operations more efficiently.

6.3.2.4 Delete

This operation removes the value associated with a specified key <v1> (or a key pair <v1,v2>). It first finds the given key and then deletes the targeted associated values. We performed 10K individual *delete* operations for randomly generated <v1> using *spo* order at different scales to illustrate the systems' *delete* performance. Fig. 6.10 shows the results. Once again, RDFix exhibited faster execution times. Hexastore was the closest system followed by hash map and B-tree. Consistent with our initial expectations, efficient searching for the given key played a major role in the faster execution of the delete operations. Performing a delete operation tends to be slower than a lookup operation (approximately 2x slower). This can be

[8] https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html (April 2021)



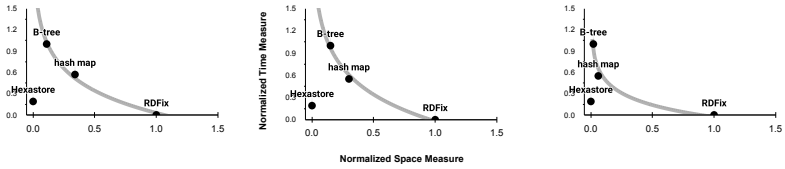
(a) over 100K triples (b) over 1M triples (c) over 10M triples

Figure 6.10: Execution times (log scale) of 10K individual *delete* operations for randomly generated <v1> at scale.

attributed to the write latency of PM since each delete operation incurs flushes to persist the changes after performing the *delete*.

6.3.3 Space-Time Tradeoffs

The tradeoffs between space and time have been a recurring theme in data management and related research. Early studies usually focused on conserving space at the expense of lower performance given the severe limits on memory (e.g., [166]). Based on this, some of the critical aspects of typical designs for space efficiency include: (i) following a two-level storage hierarchy comprising DRAM only for caching and HDD (or SSD) for permanent storage and (ii) minimizing the number of random accesses on the storage devices for speeding up execution times. The advent of in-memory databases inaugurated the trend toward the use of large amounts of main memory in database systems and memory-resident data structures to achieve efficient use of CPU cycles and improved query performance, e.g., [160, 167]. Today, the use of emerging storage technologies such as PMs enables us to directly persist any data structure without the overhead of a file system. PM's larger capacity at close-to-DRAM latency at an affordable cost also allows us to allocate larger space for holding indexes than possible with DRAM to achieve lower execution time. The proposed indexing architecture of this chapter takes advantage of the opportunities presented by PM for processing RDF data through the use of larger amounts of PM (space) to achieve significantly faster performance (time). We are guided by the principle that higher performance can be achieved when the architecture de-



(a) BSBM (structuredness = 0.95) (b) LUBM (structuredness = 0.98) (c) SP2Bench (structuredness = 0.71)

Figure 6.11: Space-time tradeoff curve to perform 100M individual lookup operations over the different datasets with 10M triples for randomly generated <v1>.

sign is tailored to the characteristics of the data and hardware [168, 169, 170, 171].

Fig. 6.11 illustrates the space-time tradeoffs among the systems for performing 100M individual lookup operations over different datasets with 10M triples for randomly generated <v1>. The X- and Y axes show the normalized space and normalized time measure respectively. Prior to the normalization, the execution times were measured in seconds and the consumed space was in MB. We used Max-Min Normalization (aka, Min-Max scaling) to re-scale the measurements to between 0 and 1. Please note that the memory allocated for the text-to-id dictionaries and the space for storing associated values are the same across the systems and are not included. As can be seen in Fig. 6.11, RDFix is faster than the other systems, but it does come at a price in the form of higher space consumption. Each of the three of the compared systems consumes lower space compared to RDFix. However, this is achieved at the expense of inferior time. Hexastore offers the best balance between space and time. Nevertheless, RDFix achieved significantly better time performance by trading off more space.

We have already discussed the relationship between the structuredness of a dataset and the difference between the space consumption of RDFix and the other systems in Section 6.3.2.1. SP2Bench has lower structuredness compared to the other two. Its tradeoff curve exhibits higher convexity than that of BSBM and LUBM (ex-

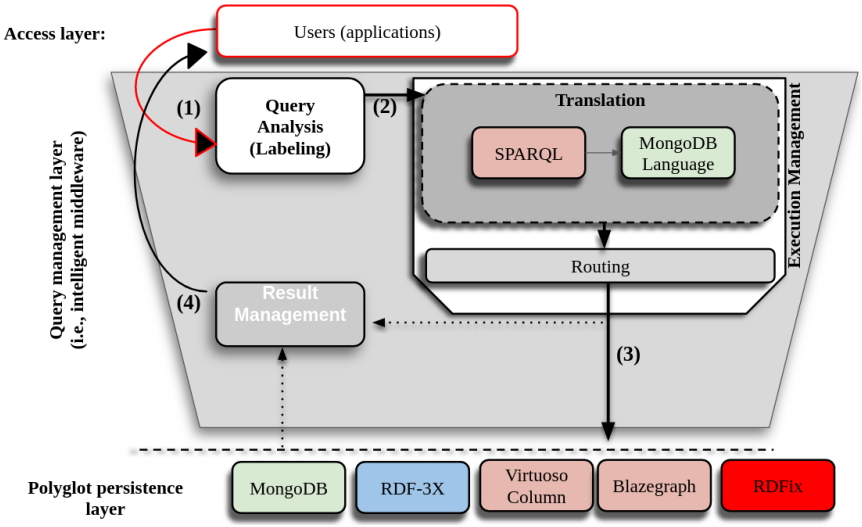


Figure 6.12: A schematic view of integrating RDFix into SymphonyDB's architecture as a stand-alone indexing structure

cluding Hexastore which is an outlier) as can be seen in Fig. 6.11 (a), (b), and (c). In other words, the higher level of *sparsity* in the SP2Bench dataset that goes with the lower structuredness adversely affects the space consumption of RDFix compared to the other two datasets with higher structuredness.

6.4 RDFix Integration into SymphonyDB

In this section, we briefly present a road map for integrating RDFix into SymphonyDB. More specifically, in Chapter 4, we identified the most appropriate DMS for each query type. Drawing on this, we have seen in Chapter 5 that SymphonyDB employed some heuristics to select the likely best-matching one or more of the integrated DMSs for each query type. For instance, SymphonyDB routed queries to both MongoDB and RDF-3X if they contain only a single triple pattern. However, if we integrate RDFix into SymphonyDB, queries with a single triple pattern can be routed to RDFix since queries with a single triple pattern are equivalent to index look-ups (see Section 2.4). More specifically, Fig. 6.12 shows a schematic view of integrating

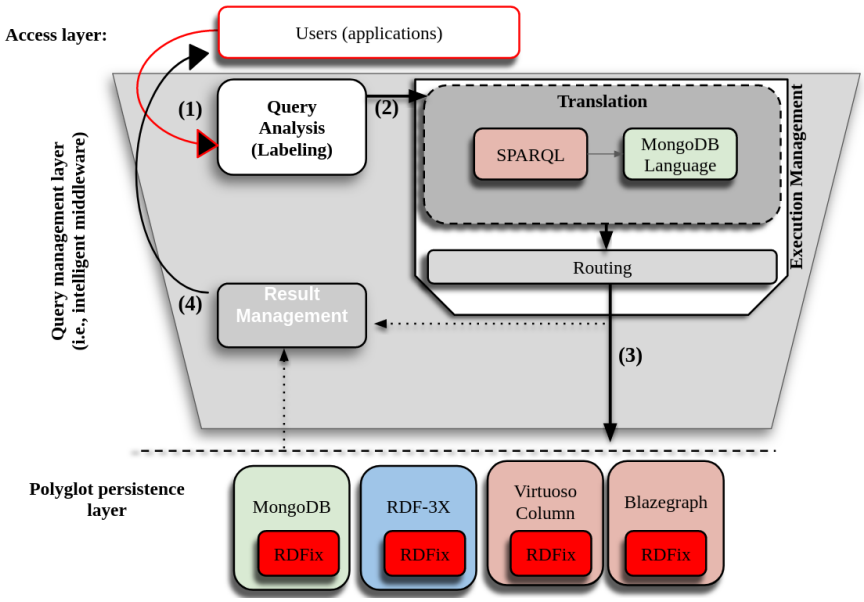


Figure 6.13: A schematic view of integrating RDFix into SymphonyDB's architecture as an embedded indexing structure

RDFix into SymphonyDB's architecture as a stand-alone indexing structure.

Typical of most access methods, RDFix can be also embedded in an RDF-store. For instance, Fig. 6.13 illustrates a schematic view of integrating RDFix into SymphonyDB's architecture by embedding it as an indexing structure into the underlying DMSs. To achieve this, it is required to overhaul each of the DMSs and incorporate RDFix into their architecture which might be very expensive.

Another plausible way to integrate RDFix into SymphonyDB is to mix the two above-mentioned scenarios. In this case, it is needed to consider both simultaneously by incorporating RDFix into the architecture of each of the DMSs as well as deploying it as a stand-alone indexing structure. This is depicted in Fig. 6.14.

6.5 Conclusion

In this chapter, we have investigated a design approach with the goal of achieving high-performance querying of RDF datasets. Specifi-

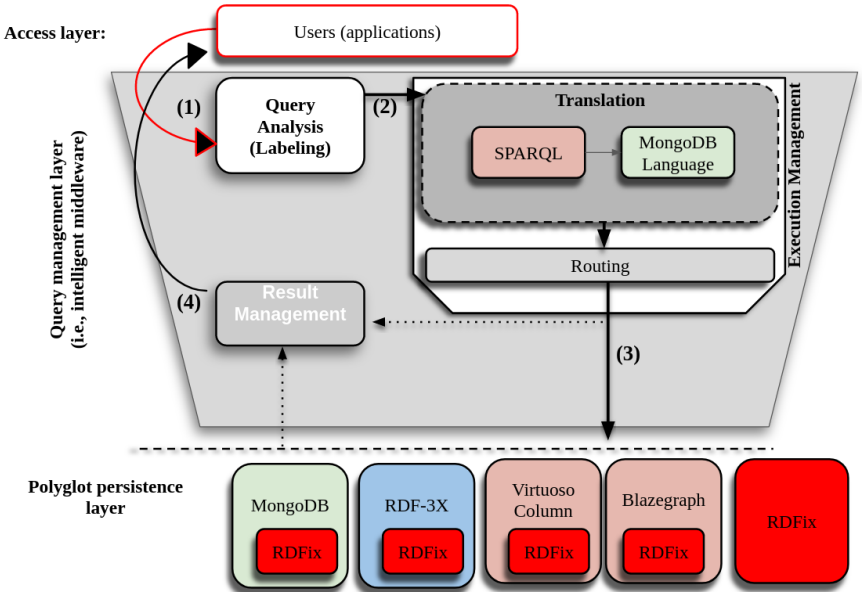


Figure 6.14: A schematic view of integrating RDFix into SymphonyDB's architecture as both a stand-alone indexing structure and an embedded indexing structure

cally, we explored the effects of emerging hardware devices on RDF storage and indexing strategies. Motivated by the critical need to achieve significant performance improvements in RDF-stores, we have developed RDFix, an efficient indexing architecture that is adapted to the triple data structure and takes advantage of the inexpensive Persistent Memory (PM) storage. RDFix's design strategy is based on combining static arrays and single linked-lists to minimize the lookup time of RDF triples and trading off memory for performance to achieve very fast execution times. The experimental evaluation of RDFix showed that it outperforms comparable index structures such as B-trees, sorted vectors (Hexastore), and hash maps by at least a factor of 3. We also presented a discussion of the space-time tradeoffs involved. Finally, reflecting on the results presented in this chapter and our findings in the previous chapters, we presented a road map for integrating RDFix into SymphonyDB.

Chapter 7

Conclusion

This chapter summarizes the primary contributions and main results of this study in Section 7.2 where we return to the questions posed at the beginning of this study and present our key findings and lessons learned. A number of caveats need to be noted regarding the present study. We discuss them in Section 7.3. The findings of this study have a number of implications for future practice. We discuss some of the most important ones for the Knowledge Management, Semantic Web, Linked Data, and Database research communities and suggest possible directions for future research in Section 7.4. Finally, our concluding remarks are presented in Section 7.5.

7.1 Introduction

At the beginning of this thesis, we have presented an extensive study of the performance of state-of-the-art DMSs. Through this study, we analyzed how efficiently major existing DMSs perform to store and process the diverse content of RDF datasets. We have demonstrated that no single system can display good performance across diverse KG query types. Thus, we have attempted to map different types of KG queries onto different types of DMS. This mapping enabled us to present a multi-database approach (i.e., SymphonyDB) to analyze incoming queries and match each of them to the likely best-performing DMS among Virtuoso, Blazegraph, RDF-3X, and MongoDB. Consistent with our expectations, the results of our experiments with SymphonyDB over standard KG benchmark datasets confirmed the efficiency of our multi-database approach across different query types and datasets.

Another avenue we have investigated was exploring the effects of emerging hardware devices on RDF storage and indexing strategies. Existing major RDF-stores are architected typically based on the following design choices: (i) disk-oriented persistent model, (ii) disk-resident indexes, and (iii) in-memory buffer-pooling to reduce latency. However, far too little attention has been paid to the effects of emerging storage devices on KG data management. To help to address this research gap, this study sought to leverage Persistent Memories (PMs) for high-performance KG query processing by introducing *RDFix*. This is a specialized architecture for indexing RDF data from the ground up in which all data persist on PM. It utilizes a multi-layered design based on static arrays and linked-lists. We have provided experimental validation of RDFix including the demonstration of its advantages for indexing diverse RDF datasets as compared to major read-optimized architectures such as B-trees, sorted vectors, and hash maps over PM. Note that typical of most access methods, RDFix can be deployed as a stand-alone indexing structure, or embedded in an RDF-store.

7.2 Summary of Contributions and Results

7.2.1 Definition of Diversity-tolerant RDF Data Management

In our definition, a diversity-tolerant RDF-store can achieve all the following three desiderata simultaneously: (i) support for a diverse range of query features; this includes required and optional graph patterns, aggregation, subqueries, negation, etc., (ii) support for widely varying RDF datasets in terms of structuredness and size; this refers to the need to support datasets with different levels of structuredness and size, and (iii) exhibiting high performance; this is an end-to-end time counted starting from a query submission time to the time the final (and correct) result is returned.

7.2.2 Returning to Research Question 1

- *How efficiently do major existing RDF-stores perform to store diverse RDF datasets and execute archetypal query types over them?*

Returning to this question, it is now possible to state that the behavior and *performance* of existing RDF-stores is not *consistent* against diverse queries. It seems to be that a single, one-size-fits-all DMS is unlikely to emerge to achieve *good* performance across query types over datasets with different levels of structuredness.

7.2.2.1 Factors Influencing Query Performance

We have made remarks on major factors influencing the performance of query processing. We have discussed why it is a challenge to generate optimized execution plans for many queries and what step-by-step access strategy is typically followed by existing RDF-stores for minimizing run time and other types of resource use (e.g., CPU time and I/O access). We have reviewed major techniques and strategies that existing RDF-stores typically follow to efficiently utilize CPU,

RAM, and I/O devices to perform intermediate steps of query processing such as sorting and how RDF-stores generally select the order in which joins would be performed.

7.2.2.2 Exploring Efficiency of Major RDF-stores

As was pointed out in the introduction to this research, it is somehow unclear how efficiently major existing data management systems perform to store diverse RDF datasets and execute archetypal queries over them. This research has gone some way toward enhancing our understanding of query performance across diverse datasets and queries by providing a fine-grained, comparative analysis of major native and non-native RDF-store types against major query types. Our comparative study complemented similar studies (e.g., [45]) that were published over five years ago.

7.2.2.3 Key Findings

The results of our investigation showed that no single RDF-store displays superior query performance across different query types. A summary of our findings is as follows:

- There are significant interaction effects between different types of RDF-stores and query types. In particular, row- and column-stores tend to be faster for tree-like joins, graph-stores tend to perform faster for subject-object joins, and document-stores are likely to be faster for star-shaped queries in most cases.
- The performance differences of existing RDF-stores can be attributed to their design choices such as their underlying storage layout, data locality, suitability of their caching techniques, the accuracy of their *cardinality estimations*, the efficiency of their index implementation, and their query optimization algorithms.

These results showed that there is a clear need to propose solutions that can hide the *variety* of KG datasets and exhibit consistent performance across diverse queries.

7.2.3 Returning to Research Question 2

How to design and develop an RDF data management system that can reduce the negative effects of the variety of RDF data and diversity of queries on the performance?

Drawing on our key findings and the lessons learned from our investigations related to exploring the performance of the major RDF-stores, we introduced a polyglot model of query processing and access languages, called *SymphonyDB*.

7.2.3.1 Key Findings

A summary of our contributions and results are presented below.

- Introducing SymphonyDB as a diversity-tolerant RDF-store that can match diverse incoming queries to the likely best-performing DMS among Virtuoso, Blazegraph, RDF-3X, and MongoDB as representative DMSs that are included in SymphonyDB at this time.
- Our results indicate that SymphonyDB performs consistently across different datasets. Its performance is almost equal to the fastest DMSs in all cases.

Taken together, these results suggest that SymphonyDB can significantly reduce the negative effects of the variety of RDF data and queries on the performance.

7.2.4 Returning to Research Question 3

How can we leverage emerging hardware devices like Persistent Memories (PMs) to design high-performance RDF-stores?

Existing systems were designed typically based on: a disk-oriented persistent model and disk-resident indexes. Although there have been a number of extensions over the past years such as supporting compression, columnar storage, bitmap indexes, and vectored execution, far too little attention has been paid to the effects of emerging hardware devices on RDF storage and indexing strategies. This

study set out to determine these effects and introduced a specialized architecture (called *RDFix*) for indexing RDF data from the ground up in which all data persist on PM.

7.2.4.1 Key Findings

A summary of our contributions and results are presented below.

- We introduced RDFix, a PM-based architecture for indexing RDF. Typical of most access methods, RDFix can be deployed as a stand-alone indexing structure, or embedded in an RDF-store with the goal of providing a high-performance and flexible access path.
- We experimentally demonstrated the advantages of RDFix as compared to major state-of-the-art indexing architectures such as B-trees, sorted vectors, and hash maps over PM. We have discussed how trading off more memory (space) could lead to the superior performance of RDFix.

One of the significant findings to emerge from this study is that new hardware devices like low-cost PMs with larger capacity at close-to-DRAM latency allow us to allocate larger spaces for holding indexes without the need for reconstruction after a crash. This enables us to trade off more memory (space) for performance at a lower cost in many cases.

7.3 Limitations

7.3.1 Redundant Information

This study offered some insight into polyglot model-based KG query processing. Utilizing a polyglot model-based solution such as SymphonyDB has shown to be effective to achieve consistent performance across diverse query types. However, approaches of this kind carry with them various well-known limitations. Perhaps one of the most serious limitations of this method is that KG datasets have to be

replicated which is determined by the number of DMS that are utilized, e.g., this number is equal to four for SymphonyDB. Difficulties may arise when an attempt is made to employ SymphonyDB for write-heavy applications where such replication can lead to higher write latency. Given that most KG datasets tend to be *read-mostly* if not *read-only* [44, 60], the write latency is not expected to be a concern in most use-cases.

7.3.2 Efficient Query Translation

When SymphonyDB selects MongoDB for a query type, a suitable JIT query translation from SPARQL to MQL is also needed. A specific caveat that needs to be noted regarding the query translation is that all SPARQL queries may not be translated to (efficient) MQL queries due to the dissimilarity between the expressiveness of SPARQL and MQL [134].

7.3.3 Maintenance and Integration

SymphonyDB includes multiple DMSs. Each of them may need periodical maintenance such as checking for the release of new versions, software patches, updating their internal statistics, etc. SymphonyDB must keep up with the updates of its underlying data platforms to perform seamlessly. As a result, the maintenance cost of a multi-database approach like SymphonyDB may be more than a conventional single DMS. However, the performance gains can compensate the added maintenance cost.

7.3.4 License

SymphonyDB includes four representative open-source and publicly available DMSs at this stage. Given the growing interest in developing KG-based applications, efficient processing of diverse queries over KGs may require a user to employ commercial DMSs as well. In this case, users may be required to purchase licenses for the commercial DMSs.

7.3.5 Parallel Operations

We compared RDFix with other read-optimized indexes, namely, B-tree, hash map, and sorted vectors (Hexastore [60]). These systems were widely accepted as high-performance indexing architectures [114]. The *single-threaded* version of all the four systems is used in our experiments. A multi-threaded version of these systems can be implemented to advance our understanding of the effects of parallel algorithms on the performance of different operations.

7.4 Future Work

This research has thrown up many interesting avenues in need of further investigation. It is recommended that further research be undertaken in the following areas: KG query typology and cloud-based non-monolithic RDF-stores. Another possible area of future research would be to investigate how update-intensive workloads can be efficiently performed over KGs.

7.4.1 Update Queries

Given that most RDF datasets tend to be *read-mostly* if not *read-only* [44, 60], we have focused our efforts on high-performance read-optimized solutions in this research. However, there is abundant room for further progress in designing high-performance solutions to support update-intensive KGs. A further study with more focus on staging architecture (where updates are deferred and later merged into the main KG dataset in a batched manner) can be suggested.

7.4.2 Programming Languages Effects on Performance

Major existing DMSs are implemented using different programming languages. It is worth exploring different ways in which the language helped the development of high-performance DMSs for KG query execution and situations in which it was less helpful. We speculate that it is needed to overhaul and re-write major existing DMSs in

the same programming language to fairly compare the effects of their architecture and design choices on query execution times.

7.4.3 Knowledge Graph Query Typology

Our experience while performing a comparative analysis of KG query performance raised several new and interesting questions and research directions that need to be addressed in the future. In our experiments, we had four archetypal query types, namely, subject-subject, subject-object, tree-like, and optional queries. However, there may be other query types that we need to consider. Further work is required to establish this.

7.4.4 Energy Consumption

In Chapter 6, we have developed the strategy of combining static arrays and single linked-lists to minimize the lookup time of RDF triples. This strategy appeared to be very effective due to trading off memory for performance resulting in very fast execution times. We have explored and presented a discussion of the space-time tradeoff involved. However, future studies on the three-way tradeoffs among space, time, and energy consumption are recommended given the sustainability imperative of energy conservation in the era of big data.

7.4.5 Cloud-based Non-monolithic RDF-stores

Our polyglot model of query processing potentially can be extendable to the non-monolithic conception of database processing in which the different components such as file systems, index structures compression, query processing engines, concurrency, consistency modules, etc. are made available in the cloud and communicate through high-performance networks (similar to [172, 173]). Further studies, which take the cloud-based non-monolithic conception into consideration, will need to be undertaken.

7.5 Concluding Remarks

Motivated by the vital role that DMSs play in enabling or enhancing a wide variety of KG-based applications, we have presented an extensive study of the efficiency of major DMSs for storing KG content and executing archetypal queries over them. The results of this study pointed to the performance variability of representative systems across diverse datasets and query types. In response, we have introduced a multi-database prototype, called *SymphonyDB*, that provides a genuine polygloty at the level of access languages and data persistence to classify queries, analyze individual query types, and match each to the best performing available platforms. We also developed, evaluated, and experimented with PM-based indexing architecture introduced as part of this thesis, called RDFix. Our results indicate that RDFix provides good overall execution time outperforming other read-optimized indexing architectures. Taken together, we encourage the use of our proposed solutions for high-performance query execution over KGs.

Bibliography

[1] Masoud Salehpour and Joseph G. Davis. "Knowledge Graphs for Processing Scientific Data: Challenges and Prospects". In: *CoRR* abs/2004.06203 (2020). arXiv: 2004.06203. URL: https://arxiv.org/abs/2004.06203.

[2] Masoud Salehpour and Joseph G. Davis. "The Effects of Different JSON Representations on Querying Knowledge Graphs". In: *CoRR* abs/2004.04286 (2020). arXiv: 2004.04286. URL: https://arxiv.org/abs/2004.04286.

[3] Masoud Salehpour and Joseph G. Davis. "A Comparative Analysis of Knowledge Graph Query Performance". In: *Proc. of Int. Conf. on Transdisciplinary AI (TransAI)*. 2021, pp. 33–40. DOI: 10.1109/TransAI51903.2021.00014.

[4] Masoud Salehpour and Joseph G. Davis. "Towards Diversity-Tolerant RDF-Stores". In: *Proc. of the ACM Symposium on Applied Computing (SAC)*. 2022, to–appear.

[5] Masoud Salehpour and Joseph G. Davis. "A Comparative Analysis of Knowledge Graph Query Performance". In: *CoRR* abs/2004.05648 (2020). arXiv: 2004.05648. URL: https://arxiv.org/abs/2004.05648.

[6] Masoud Salehpour and Joseph G. Davis. "SymphonyDB: A Polyglot Model for Knowledge Graph Query Processing". In: *Proc. of Int. Conf. on Transdisciplinary AI (TransAI)*. 2021, pp. 25–32. DOI: 10.1109/TransAI51903.2021.00013.

[7] Gerhard Weikum, Xin Luna Dong, Simon Razniewski, and Fabian Suchanek. "Machine Knowledge: Creation and Curation of Comprehensive Knowledge Bases". In: *Foundations and Trends in Databases* 10.2-4 (2021), pp. 108–490.

[8] Douglas B. Lenat and Edward A. Feigenbaum. "On the thresholds of knowledge". In: *Artificial Intelligence* 47.1 (1991), pp. 185–250.

[9] Gerhard Weikum, Johannes Hoffart, and Fabian M. Suchanek. "Ten Years of Knowledge Harvesting: Lessons and Challenges". In: *IEEE Data Eng. Bull.* 39.3 (2016), pp. 41–50.

[10] Amit Singhal. *Introducing the Knowledge Graph: things, not strings.* [Online] Available: https://www.blog.google/products/search/introducing-knowledge-graph-things-not/. 2012.

[11] Richard Qian. *Understand Your World with Bing.* [Online] Available: https://blogs.bing.com/search/2013/03/21/understand-your-world-with-bing/. 2013.

[12] D. A. Ferrucci. "Introduction to "This is Watson"". In: *IBM J. Res. Dev.* 56.3 (2012), pp. 235–249. DOI: 10.1147/JRD.2012.2184356. URL: https://doi.org/10.1147/JRD.2012.2184356.

[13] Xin Luna Dong, Xiang He, Andrey Kan, Xian Li, Yan Liang, Jun Ma, Yifan Ethan Xu, Chenwei Zhang, Tong Zhao, Gabriel Blanco Saldana, Saurabh Deshpande, Alexandre Michetti Manduca, Jay Ren, Surender Pal Singh, Fan Xiao, Haw-Shiuan Chang, Giannis Karamanolakis, Yuning Mao, Yaqing Wang, Christos Faloutsos, Andrew McCallum, and Jiawei Han. "AutoKnow: Self-Driving Knowledge Collection for Products of Thousands of Types". In: *Proc. of the ACM SIGKDD Int. Conf. on Knowledge Discovery & Data Mining (KDD)*. 2020, pp. 2724–2734. URL: https://doi.org/10.1145/3394486.3403323.

[14] Xin Luna Dong. *Keynote at Int. Conf. on Data Engineering (ICDE): Building a Broad Knowledge Graph for Products.* [Online] Available: http://lunadong.com/talks/BG.pdf. 2019.

[15] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. "Foundations of Modern Query Languages for Graph Databases". In: *ACM Comput. Surv.* 50.5 (2017). URL: https://doi.org/10.1145/3104031.

[16] Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudia d'Amato, Gerard de Melo, Claudio Gutierrez, José Emilio Labra Gayo, Sabrina Kirrane, Sebastian Neumaier, Axel Polleres, Roberto Navigli, Axel-Cyrille Ngonga Ngomo, Sabbir M. Rashid, Anisa Rula, Lukas Schmelzeisen, Juan Sequeda, Steffen Staab, and Antoine Zimmermann. *Knowledge Graphs.* 2021. arXiv: 2003.02320.

[17] Michael Färber, Frederic Bartscherer, Carsten Menne, and Achim Rettinger. "Linked data quality of DBpedia, Freebase, Opencyc, Wikidata, and Yago". In: *Semantic Web* 9.1 (2018), pp. 77–129.

[18] David C. Faye, Olivier Curé, and Guillaume Blin. "A survey of RDF storage approaches". In: *Revue Africaine de la Recherche en Informatique et Mathématiques Appliquées* 15 (2012), pp. 11–35. URL: https://hal.inria.fr/hal-01299496.

[19] Marcin Wylot, Manfred Hauswirth, Philippe Cudré-Mauroux, and Sherif Sakr. "RDF Data Storage and Query Processing Schemes: A Survey". In: *ACM Comput. Surv.* 51.4 (2018), 84:1–84:36.

[20] Lisa Ehrlinger and Wolfram Wöß. "Towards a Definition of Knowledge Graphs." In: *SEMANTiCS (Posters, Demos, SuCCESS)* 48 (2016), pp. 1–4.

[21] Heiko Paulheim. "Knowledge graph refinement: A survey of approaches and evaluation methods". In: *Semantic Web* 8.3 (2017), pp. 489–508.

[22] Muhammad Saleem, Gábor Szárnyas, Felix Conrads, Syed Ahmad Chan Bukhari, Qaiser Mehmood, and Axel-Cyrille Ngonga Ngomo. "How Representative Is a SPARQL Benchmark? An Analysis of RDF Triplestore Benchmarks". In: *Proc. of the Int. Conf. on World Wide Web (WWW)*. 2019, pp. 1623–1633.

[23] Güneş Aluç, M. Tamer Özsu, and Khuzaima Daudjee. "Building Self-clustering RDF Databases Using Tunable-LSH". In: *The VLDB Journal* 28.2 (2019), pp. 173–195.

[24] The RDF Working Group (as a W3C Working Group). *RDF 1.1 Primer*. [Online] Available: https://www.w3.org/TR/2014/NOTE-rdf11-primer-20140624/. 2014.

[25] M. Durst and M. Suignard. *Internationalized Resource Identifiers (IRIs)*. http://www.ietf.org/rfc/rfc3987.txt. 2005.

[26] M. Tamer Özsu. "A Survey of RDF Data Management Systems". In: *Frontiers of Computer Science* 10.3 (2016), pp. 418–432.

[27] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. "Cypher: An Evolving Query Language for Property Graphs". In: *Proc. of the ACM Int. Conf. on Management of Data (SIGMOD)*. 2018, pp. 1433–1445. URL: https://doi.org/10.1145/3183713.3190657.

[28] Marko A. Rodriguez. "The Gremlin Graph Traversal Machine and Language (Invited Talk)". In: *Proc. of Symp. on Database Programming Languages (DBPL)*. 2015, pp. 1–10. URL: https://doi.org/10.1145/2815072.2815073.

[29] Renzo Angles, Marcelo Arenas, Pablo Barcelo, Peter Boncz, George Fletcher, Claudio Gutierrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan Sequeda, Oskar van Rest, and Hannes Voigt. "G-CORE: A Core for Future Graph Query Languages". In: *Proc. of the ACM Int. Conf. on Management of Data (SIGMOD)*. 2018, pp. 1421–1432. URL: https://doi.org/10.1145/3183713.3190654.

[30] Philipp Seifer, Johannes Härtel, Martin Leinberger, Ralf Lämmel, and Steffen Staab. "Empirical Study on the Usage of Graph Query Languages in Open Source Java Projects". In: *Proc. of the ACM SIGPLAN Int. Conf. on Software Language Engineering (SLE)*. 2019, pp. 152–166. URL: https://doi.org/10.1145/3357766.3359541.

[31] Angela Bonifati, Wim Martens, and Thomas Timm. "An analytical study of large SPARQL query logs". In: *The VLDB Journal* 29.2 (2020), pp. 655–679.

[32] Qingyu Guo, Fuzhen Zhuang, Chuan Qin, Hengshu Zhu, Xing Xie, Hui Xiong, and Qing He. *A Survey on Knowledge Graph-Based Recommender Systems*. 2020. arXiv: 2003.00911.

[33] Rouzbeh Meymandpour and Joseph G. Davis. "A semantic similarity measure for linked data: An information content-based approach". In: *Knowledge-Based Systems* 109 (2016), pp. 276–293.

[34] Fabian M. Suchanek and Nicoleta Preda. "Semantic Culturomics". In: *Proc. VLDB Endow.* 7.12 (2014), pp. 1215–1218. URL: https://doi.org/10.14778/2732977.2732994.

[35] Sofia Alexaki, Vassilis Christophides, Gregory Karvounarakis, Dimitris Plexousakis, and K Tolle. "On Storing Voluminous RDF Descriptions: The Case of Web Portal Catalogs." In: *WebDB*. 2001, pp. 43–48.

[36] Sofia Alexaki, Vassilis Christophides, Gregory Karvounarakis, Dimitris Plexousakis, and Karsten Tolle. "The ICS-FORTH

RDFSuite: Managing Voluminous RDF Description Bases." In: *SemWeb.* 2001.

[37] David Beckett. "The design and implementation of the Redland RDF application framework". In: *Proc. of the Int. Conf. on World Wide Web (WWW).* 2001, pp. 449–456.

[38] Stephen Harris and Nicholas Gibbins. "3store: Efficient bulk RDF storage". In: *Proc. of the Int. Workshop on Practical and Scalable Semantic Systems (PSSS).* 2003.

[39] Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. "Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema". In: *Proc. of the Int. Semantic Web Conf. (ISWC).* 2002, pp. 54–68.

[40] Raphael Volz, Daniel Oberle, Steffen Staab, and B. Motik. "KAON SERVER - A Semantic Web Management System". In: *Proc. of the Int. Conf. on World Wide Web (WWW).* 2003.

[41] K. Wilkinson, C. Sayers, H. Kuno, and D. Reynolds. "Efficient RDF Storage and Retrieval in Jena2". In: *SWDB.* 2003.

[42] Li Ma, Zhong Su, Yue Pan, Li Zhang, and Tao Liu. "RStar: An RDF Storage and Query System for Enterprise Resource Management". In: *Proc. of the ACM Int. Conf. on Information and Knowledge Management (CIKM).* 2004, pp. 484–491.

[43] Zhengxiang Pan and Jeff Heflin. "DLDB: Extending Relational Databases to Support Semantic Web Queries". In: *Proc. of the Int. Workshop on Practical and Scalable Semantic Systems (PSSS).* 2003, pp. 109–113.

[44] Thomas Neumann and Gerhard Weikum. "The RDF-3X engine for scalable management of RDF data". In: *Proc. VLDB Endow.* 19.1 (2010), pp. 91–113.

[45] Güneş Aluç, Olaf Hartig, M. Tamer Özsu, and Khuzaima Daudjee. "Diversified Stress Testing of RDF Data Management Systems". In: *Proc. of the Int. Semantic Web Conf. (ISWC).* 2014, pp. 197–212.

[46] Minh-Duc Pham, Linnea Passing, Orri Erling, and Peter Boncz. "Deriving an Emergent Relational Schema from RDF Data". In: *Proc. of the Int. Conf. on World Wide Web (WWW)*. 2015, pp. 864–874.

[47] Thomas Neumann and Gerhard Weikum. "Scalable Join Processing on Very Large RDF Graphs". In: *Proc. of the ACM Int. Conf. on Management of Data (SIGMOD)*. 2009, pp. 627–640.

[48] Peter Boncz, Orri Erling, and Minh-Duc Pham. "Advances in Large-Scale RDF Data Management". In: *Linked Open Data – Creating Knowledge Out of Interlinked Data: Results of the LOD2 Project*. 2014, pp. 21–44.

[49] Alon Halevy, Michael Franklin, and David Maier. "Principles of Dataspace Systems". In: *Proc. of the ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems (PODS)*. 2006, pp. 1–9.

[50] Songyun Duan, Anastasios Kementsietsidis, Kavitha Srinivas, and Octavian Udrea. "Apples and Oranges: A Comparison of RDF Benchmarks and Real RDF Datasets". In: *Proc. of the ACM Int. Conf. on Management of Data (SIGMOD)*. 2011, pp. 145–156. DOI: 10.1145/1989323.1989340. URL: https://doi.org/10.1145/1989323.1989340.

[51] Jiewen Huang, Daniel J. Abadi, and Kun Ren. "Scalable SPARQL Querying of Large RDF Graphs". In: *Proc. VLDB Endow.* 4.11 (2011), pp. 1123–1134.

[52] Sven Groppe. *Data management and query processing in Semantic Web databases*. Springer Science & Business Media, 2011.

[53] Marcelo Arenas, Gonzalo Diaz, Achille Fokoue, Anastasios Kementsietsidis, and Kavitha Srinivas. "A Principled Approach to Bridging the Gap between Graph Data and Their Schemas". In: *Proc. VLDB Endow.* 7.8 (Apr. 2014), pp. 601–612.

[54] Christian Bizer and Andreas Schultz. "The Berlin SPARQL Benchmark". In: *Int. J. Semantic Web Inf. Syst.* 5 (2009), pp. 1–24.

[55] Samantha Bail, Sandra Alkiviadous, Bijan Parsia, David Workman, Mark Van Harmelen, RS Concalves, and Cristina Garilao. "FishMark: A linked data application benchmark". In: *Proc. of the Int. Workshop on Scalable and High-Performance Semantic Web Systems (SSWS)*. 2012, pp. 1–15.

[56] Gianluca Demartini, Iliya Enchev, Marcin Wylot, Joël Gapany, and Philippe Cudré-Mauroux. "BowlognaBench—Benchmarking RDF Analytics". In: *Proc. of of the Int. Symp. on Data-Driven Process Discovery and Analysis (SIMPDA)*. 2012, pp. 82–102.

[57] Hongyan Wu, Toyofumi Fujiwara, Yasunori Yamamoto, Jerven Bolleman, and Atsuko Yamaguchi. "BioBenchmark Toyama 2012: an evaluation of the performance of triple stores on biological data". In: *Journal of Biomedical Semantics* 5.1 (2014), pp. 32–43.

[58] W Ross Ashby. "Requisite variety and its implications for the control of complex systems". In: *Facets of systems science*. Springer, 1991, pp. 405–417.

[59] Michael Stonebraker, Chuck Bear, Uğur Çetintemel, Mitch Cherniack, Tingjian Ge, Nabil Hachem, Stavros Harizopoulos, John Lifter, Jennie Rogers, and Stan Zdonik. "One Size Fits All?-Part 2: Benchmarking Results". In: *CIDR*. 2007.

[60] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. "Hexastore: Sextuple Indexing for Semantic Web Data Management". In: *Proc. VLDB Endow.* 1.1 (2008), pp. 1008–1019.

[61] Joy Arulraj, Andrew Pavlo, and Subramanya R. Dulloor. "Let's Talk About Storage & Recovery Methods for Non-Volatile Memory Database Systems". In: *Proc. of the Int. Conf. on Management of Data (SIGMOD)*. 2015, pp. 707–722.

[62] Diego Didona, Nikolas Ioannou, Radu Stoica, and Kornilios Kourtis. "Toward a Better Understanding and Evaluation of Tree Structures on Flash SSDs". In: *Proc. VLDB Endow.* 14.3 (2020), pp. 364–377.

[63] Björn Daase, Lars Jonas Bollmeier, Lawrence Benson, and Tilmann Rabl. "Maximizing Persistent Memory Bandwidth Utilization for OLAP Workloads". In: *Proc. of the ACM Int. Conf. on Management of Data (SIGMOD)–to appear*. 2021.

[64] Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. "Building blocks for persistent memory". In: *Proc. VLDB Endow.* 29.6 (2020), pp. 1223–1241.

[65] Philipp Götze, Alexander van Renen, Lucas Lersch, Viktor Leis, and Ismail Oukid. "Data management on non-volatile memory: a perspective". In: *Datenbank-Spektrum* 18.3 (2018), pp. 171–182.

[66] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. "Recipe: Converting Concurrent DRAM Indexes to Persistent-Memory Indexes". In: *Proc. of the ACM Symp. on Operating Systems Principles (SOSP)*. 2019, pp. 462–477.

[67] Edward W Schneider. "Course Modularization Applied: The Interface System and Its Implications For Sequence Control and Data Analysis." In: *Association for the Development of Instructional Systems (ADIS)* (1973).

[68] Ezio Marchi and Osvaldo Miguel. "On the structure of the teaching-learning interactive process". In: *International Journal of Game Theory* 3.2 (1974), pp. 83–99.

[69] Pieter Hendrik de Vries. "Representation of scientific texts in knowledge graphs". PhD thesis. Faculty of Behavioural and Social Sciences, University of Groningen, 1989. URL: https://research.rug.nl/en/publications/representation-of-scientific-texts-in-knowledge-graphs.

[70] P. James. "Knowledge graphs". In: *Linguistic Instruments in Knowledge Engineering*. Ed. by Reind P. van de Riet and Robert A. Meersman. 1991 Workshop on Linguistic Instruments in Knowledge Engineering ; [P. James is likely a pseudonym for other authors] ; Conference date: 17-01-1991 Through 18-01-1991. Elsevier, 1992, pp. 97–117.

[71] Lei Zhang. "Knowledge graph theory and structural parsing". PhD thesis. University of Twente, 2002. ISBN: 90-365-1835-0. URL: https://research.utwente.nl/en/publications/knowledge-graph-theory-and-structural-parsing.

[72] Roel Popping. "Knowledge Graphs and Network Text Analysis". In: *Social Science Information* 42.1 (2003). Relation: http://www.rug.nl/ date_submitted:2007 Rights: University of Groningen, pp. 91–106. ISSN: 1461-7412. DOI: 10.1177/0539018403042001798.

[73] Markus Krötzsch and Gerhard Weikum. "Web Semantics: Science, Services and Agents on the World Wide Web". In: *Journal of Web Semantics: Special Issue on Knowledge Graphs* 37-38 (2016), pp. 53–57.

[74] Michael K. Bergman. *A Common Sense View of Knowledge Graphs*. [Online] Available: https://www.mkbergman.com/2244/a-common-sense-view-of-knowledge-graphs/. 2019.

[75] Douglas B. Lenat. "CYC: A Large-Scale Investment in Knowledge Infrastructure". In: *Commun. ACM* 38.11 (1995), pp. 33–38. URL: https://doi.org/10.1145/219717.219745.

[76] George A Miller. *WordNet: An electronic lexical database*. MIT press, 1998. URL: https://direct.mit.edu/books/book/1928/WordNetAn-Electronic-Lexical-Database.

[77] The Association of Research Libraries. *ARL White Paper on Wikidata Opportunities and Recommendations*. [Online] Available: https://www.arl.org/wp-content/uploads/2019/04/2019.04.18-ARL-white-paper-on-Wikidata.pdf. 2019.

[78] Mike Cummings. *Web application shares "Science Stories" of pioneering women at Yale*. [Online] Available: https://news.yale.edu/2019/01/03/web-application-shares-science-stories-pioneering-women-yale. 2019.

[79] Waagmeester Andra, Gregory Stupp, Burgstaller-Muehlbacher Sebastian, Benjamin M Good, Griffith Malachi, Obi L Griffith, Hanspers Kristina, Hermjakob Henning, Toby S Hudson, Hybiske Kevin, et al. "Wikidata as a knowledge graph for the life sciences". In: *eLife* (2020). URL: https://pubmed.ncbi.nlm.nih.gov/32180547/.

[80] Minh Duc Pham. "Emergent Relational Schemas for RDF". PhD thesis. Vrije Universiteit Amsterdam, 2018. URL: https://research.vu.nl/en/publications/emergent-relational-schemas-for-rdf.

[81] Andreas Harth, Katja Hose, and Ralf Schenkel. *Linked data management*. CRC Press, 2014. URL: https://doi.org/10.1201/b16859.

[82] Andreas Harth, Katja Hose, and Ralf Schenkel. *Linked data management; Chapter 1: Linked Data & the Semantic Web Standards*. CRC Press, 2014. URL: https://doi.org/10.1201/b16859.

[83] Harsh Vrajeshkumar Thakker. "On Supporting Interoperability between RDF and Property Graph Databases". PhD thesis. Rheinische Friedrich-Wilhelms-Universitat Bonn, May 2021. URL: https://hdl.handle.net/20.500.11811/9083.

[84] Rouzbeh Meymandpour. "Semantic Analysis using Linked Open Data: An Information Content-based Approach". PhD thesis. School of Computer Sciences, The University of Sydney, 2014.

[85] Medha Atre. "Left Bit Right: For SPARQL Join Queries with OPTIONAL Patterns (Left-outer-joins)". In: *Proc. of the ACM Int. Conf. on Management of Data (SIGMOD)*. 2015, pp. 1793–1808.

[86] Gianfranco E. Modoni, Marco Sacco, and Walter Terkaj. "A survey of RDF store solutions". In: *Proc. of the Int. Conf. on Engineering, Technology and Innovation (ICE)*. 2014, pp. 1–7. DOI: 10.1109/ICE.2014.6871541.

[87] Minh-Duc Pham and Peter Boncz. "Exploiting emergent schemas to make RDF systems more efficient". In: *Proc. of the Int. Semantic Web Conf. (ISWC)*. 2016, pp. 463–479.

[88] Güneş Aluç. "Workload Matters: A Robust Approach to Physical RDF Database Design". PhD thesis. The University of Waterloo, 2015. URL: https://uwspace.uwaterloo.ca/handle/10012/9774.

[89] Daniel J Abadi, Adam Marcus, Samuel R Madden, and Kate Hollenbach. "Scalable Semantic Web Data Management Using Vertical Partitioning". In: *Proc. VLDB Endow.* 2007, pp. 411–422.

[90] Daniel J. Abadi, Adam Marcus, Samuel R. Madden, and Kate Hollenbach. "SW-Store: a vertically partitioned DBMS for Semantic Web data management". In: *Proc. VLDB Endow.* 18.2 (2009), pp. 385–406.

[91] Sherif Sakr, Marcin Wylot, Raghava Mutharaju, Danh Le Phuoc, and Irini Fundulaki. "Centralized RDF Query Processing". In: *Linked Data: Storing, Querying, and Reasoning*. Springer International Publishing, 2018, pp. 33–49. URL: https://doi.org/10.1007/978-3-319-73515-3_3.

[92] Andreas Harth, Katja Hose, and Ralf Schenkel. *Linked data management; Chapter 5: Efficient Query Processing in RDF Databases*. CRC Press, 2014. URL: https://doi.org/10.1201/b16859.

[93] Kevin Wilkinson and Kevin Wilkinson. "Jena property table implementation". In: *Proc. of the Int. Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS)*. 2006.

[94] Katja Hose, Ralf Schenkel, Martin Theobald, and Gerhard Weikum. "Database Foundations for Scalable RDF Processing". In: *Reasoning Web. Semantic Technologies for the Web of Data: 7th International Summer School 2011, Galway, Ireland, August 23-27, 2011, Tutorial Lectures*. Ed. by Axel Polleres, Claudia d'Amato, Marcelo Arenas, Siegfried Handschuh, Paula Kroner, Sascha Ossowski, and Peter Patel-Schneider. Springer Berlin Heidelberg, 2011, pp. 202–249. ISBN: 978-3-642-23032-5. URL: https://doi.org/10.1007/978-3-642-23032-5_4.

[95] Lefteris Sidirourgos, Romulo Goncalves, Martin Kersten, Niels Nes, and Stefan Manegold. "Column-store Support for RDF Data Management: Not All Swans Are White". In: *Proc. VLDB Endow.* 1.2 (2008), pp. 1553–1563.

[96] Michael Sintek and Malte Kiesel. "RDFBroker: A signature-based high-performance RDF store". In: *Proc. of the European Semantic Web Conf. (ESWC)*. 2006, pp. 363–377.

[97] Jianling Sun and Qiang Jin. "Scalable RDF store based on HBase and MapReduce". In: *Proc. of the Int. Conf. on Advanced Computer Theory and Engineering (ICACTE)*. Vol. 1. IEEE. 2010, pp. V1–633.

[98] Jacopo Urbani, Frank van Harmelen, Stefan Schlobach, and Henri Bal. "QueryPIE: Backward Reasoning for OWL Horst over Very Large Knowledge Bases". In: *Proc. of the Int. Semantic Web Conf. (ISWC)*. Ed. by Lora Aroyo, Chris Welty, Harith Alani, Jamie Taylor, Abraham Bernstein, Lalana Kagal, Natasha Noy, and Eva Blomqvist. 2011, pp. 730–745.

[99] Christophe Gueret, Spyros Kotoulas, and Paul Groth. "Triplecloud: An infrastructure for exploratory querying over web-scale RDF data". In: *Proc. of the IEEE/WIC/ACM Int. Conf. on Web Intelligence and Intelligent Agent Technology*. Vol. 3. 2011, pp. 245–248.

[100] Nikolaos Papailiou, Ioannis Konstantinou, Dimitrios Tsoumakos, and Nectarios Koziris. "H2RDF: adaptive query processing on

RDF data in the cloud." In: *Proc. of the Int. Conf. on World Wide Web (WWW)*. 2012, pp. 397–400.

[101] Eleni Stefani and Klesti Hoxha. "Implementing Triple-Stores using NoSQL Databases." In: *Proc. of the Int. Conf. on Recent Trends and Applications in Computer Science and Information Technology (RTA-CSIT)*. Ed. by Endrit Xhina and Klesti Hoxha. 2018, pp. 86–92. URL: http://ceur-ws.org/Vol-2280/paper-13.pdf.

[102] Philippe Cudré-Mauroux, Iliya Enchev, Sever Fundatureanu, Paul Groth, Albert Haque, Andreas Harth, Felix Leif Keppmann, Daniel Miranker, Juan F. Sequeda, and Marcin Wylot. "NoSQL Databases for RDF: An Empirical Evaluation". In: *Proc. of the Int. Semantic Web Conf. (ISWC)*. Ed. by Harith Alani, Lalana Kagal, Achille Fokoue, Paul Groth, Chris Biemann, Josiane Xavier Parreira, Lora Aroyo, Natasha Noy, Chris Welty, and Krzysztof Janowicz. 2013, pp. 310–325.

[103] Craig Chasseur, Yinan Li, and Jignesh M Patel. "Enabling JSON Document Stores in Relational Systems." In: *WebDB*. Vol. 13. 2013, pp. 14–15.

[104] Steve Harris, Nick Lamb, Nigel Shadbolt, et al. "4store: The design and implementation of a clustered RDF store". In: *Proc. of the Int. Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS)*. Vol. 94. 2009.

[105] Orri Erling. "Virtuoso, a Hybrid RDBMS/Graph Column Store." In: *IEEE Data Eng. Bull.* 35.1 (2012), pp. 3–8.

[106] Eugene Inseok Chong, Souripriya Das, George Eadon, and Jagannathan Srinivasan. "An efficient SQL-based RDF querying scheme". In: *Proc. VLDB Endow.* 2005, pp. 1216–1227.

[107] B. McBride. "Jena: a Semantic Web toolkit". In: *IEEE Internet Computing* 6.6 (2002), pp. 55–59.

[108] Sivaramakrishnan Narayanan, Tahsin M Kurc, and Joel H Saltz. "DBOWL: Towards extensional queries on a billion

statements using relational databases". In: *Ohio State University, Tech. Rep. OSUBMI_TR_2006_n03* (2006). URL: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.129.1668&rep=rep1&type=pdf.

[109] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. "C-store: A Column-oriented DBMS". In: *Proc. VLDB Endow.* 2005, pp. 553–564.

[110] Sherif Sakr, Marcin Wylot, Raghava Mutharaju, Danh Le Phuoc, and Irini Fundulaki. *Linked Data: Storing, Querying, and Reasoning*. Springer, 2018. URL: https://link.springer.com/book/10.1007/978-3-319-73515-3.

[111] Miguel A. Martínez-Prieto, Javier D. Fernández, and Rodrigo Cánovas. "Compression of RDF Dictionaries". In: *Pr. of the ACM Symp. on Applied Computing (SAC)*. 2012, pp. 340–347. URL: https://doi.org/10.1145/2245276.2245343.

[112] A. Harth and S. Decker. "Optimized index structures for querying RDF from the Web". In: *Proc. of the Latin American Web Congress (LA-WEB)*. 2005. DOI: 10.1109/LAWEB.2005.25.

[113] Andreas Harth, Jürgen Umbrich, Aidan Hogan, and Stefan Decker. "YARS2: A Federated Repository for Querying Graph Structured Data from the Web". In: *Proc. of the Semantic Web and Asian conf. on Asian Semantic Web*. Ed. by Karl Aberer, Key-Sun Choi, Natasha Noy, Dean Allemang, Kyung-Il Lee, Lyndon Nixon, Jennifer Golbeck, Peter Mika, Diana Maynard, Riichiro Mizoguchi, Guus Schreiber, and Philippe Cudré-Mauroux. Springer Berlin Heidelberg, 2007, pp. 211–224. ISBN: 978-3-540-76298-0.

[114] Cathrin Weiss and Abraham Bernstein. "On-disk storage techniques for Semantic Web data-Are B-Trees always the optimal solution?" In: *Proc. of the Int. Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS)*. 2009, pp. 49–64.

[115] Medha Atre, Jagannathan Srinivasan, and James Hendler. "BitMat: A Main-Memory Bit Matrix of RDF Triples for Conjunctive Triple Pattern Queries". In: *Proc. of the Int. Semantic Web Conf. Posters and Demonstrations (ISWC-PD)*. CEUR-WS.org, 2008, pp. 1–2.

[116] George H.L. Fletcher and Peter W. Beck. "Scalable Indexing of RDF Graphs for Efficient Join Processing". In: *Proc. of the ACM Int. Conf. on Information and Knowledge Management (CIKM)*. 2009, pp. 1513–1516. URL: https://doi.org/10.1145/1645953.1646159.

[117] David Wood. "Kowari: A Platform for Semantic Web Storage and Analysis". In: *Proc. of the XTech Conf.* 2005, pp. 05–12.

[118] Thanh Tran, Gunter Ladwig, and Sebastian Rudolph. "Istore: efficient RDF data management using structure indexes for general graph structured data". In: *Institute AIFB, Karlsruhe Institute of Technology* (2009).

[119] Martin Fowler. *Polyglot Persistence*. https://martinfowler.com/bliki/PolyglotPersistence.html. Accessed on 2019-11-10. 2011.

[120] Zhen Hua Liu, Jiaheng Lu, Dieter Gawlick, Heli Helskyaho, Gregory Pogossiants, and Zhe Wu. "Multi-model Database Management Systems - A Look Forward". In: *Heterogeneous Data Management, Polystores, and Analytics for Healthcare*. Springer, 2019, pp. 16–29.

[121] Jennie Duggan, Aaron J Elmore, Michael Stonebraker, Magda Balazinska, Bill Howe, Jeremy Kepner, Sam Madden, David Maier, Tim Mattson, and Stan Zdonik. "The bigdawg polystore system". In: *ACM Sigmod Record* 44.2 (2015), pp. 11–16.

[122] Divy Agrawal, Sanjay Chawla, Bertty Contreras-Rojas, Ahmed Elmagarmid, Yasser Idris, Zoi Kaoudi, Sebastian Kruse, Ji Lucas, Essam Mansour, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, Saravanan Thirumuruganathan,

and Anis Troudi. "RHEEM: Enabling Cross-platform Data Processing: May the Big Data Be with You!" In: *The VLDB Journal* 11.11 (2018).

[123] Ionel Gog, Malte Schwarzkopf, Natacha Crooks, Matthew P. Grosvenor, Allen Clement, and Steven Hand. "Musketeer: All for One, One for All in Data Processing Systems". In: *Proc. of the ACM Int. European Conf. on Computer Systems (EuroSys)*. 2015, 2:1–2:16.

[124] Bart Samwel, John Cieslewicz, Ben Handy, Jason Govig, Petros Venetis, Chanjun Yang, Keith Peters, Jeff Shute, Daniel Tenedorio, Himani Apte, Felix Weigel, David Wilhite, Jiacheng Yang, Jun Xu, Jiexing Li, Zhan Yuan, Craig Chasseur, Qiang Zeng, Ian Rae, Anurag Biyani, Andrew Harn, Yang Xia, Andrey Gubichev, Amr El-Helw, Orri Erling, Zhepeng Yan, Mohan Yang, Yiqun Wei, Thanh Do, Colin Zheng, Goetz Graefe, Somayeh Sardashti, Ahmed M. Aly, Divy Agrawal, Ashish Gupta, and Shiv Venkataraman. "F1 Query: Declarative Querying at Scale". In: *Proc. VLDB Endow.* 11.12 (2018), pp. 1835–1848.

[125] Harold Lim, Yuzhang Han, and Shivnath Babu. "How to Fit when No One Size Fits." In: *CIDR*. 2013.

[126] Stratos Idreos, Niv Dayan, Wilson Qin, Mali Akmanalp, Sophie Hilgard, Andrew Ross, James Lennon, Varun Jain, Harshita Gupta, David Li, and Zichen Zhu. "Design Continuums and the Path Toward Self-Designing Key-Value Stores that Know and Learn". In: *CIDR*. 2019.

[127] Shrainik Jain, Jiaqi Yan, Thierry Cruanes, and Bill Howe. "Database-Agnostic Workload Management". In: *CIDR*. 2019.

[128] Philipp Marian Grulich, Steffen Zeuch, and Volker Markl. "Babelfish: Efficient Execution of Polyglot Queries". In: 15.2 (2021), pp. 196–210. URL: https://doi.org/10.14778/3489496.3489501.

[129] Shi Qiao and Z. Meral Özsoyoğlu. "RBench: Application-Specific RDF Benchmarking". In: *Proc. of the ACM Int. Conf. on Management of Data (SIGMOD)*. 2015, pp. 1825–1838. URL: https://doi.org/10.1145/2723372.2746479.

[130] Domingo De Abreu, Alejandro Flores, Guillermo Palma, Valeria Pestana, José Pinero, Jonathan Queipo, José Sánchez, and Maria-Esther Vidal. "Choosing Between Graph Databases and RDF Engines for Consuming and Mining Linked Data." In: *Cold* 13 (2013), pp. 37–49. URL: http://ceur-ws.org/Vol-1034/DeAbreuEtAl%5C_COLD2013.pdf.

[131] Daniel Hernández, Aidan Hogan, Cristian Riveros, Carlos Rojas, and Enzo Zerega. "Querying wikidata: Comparing sparql, relational and graph databases". In: *Proc. of the Int. Semantic Web Conf. (ISWC)*. 2016, pp. 88–103.

[132] Riham Abdel Kader. "ROX: Run-Time Optimization of XQueries". PhD thesis. University of Twente, 2010. ISBN: 978-90-365-3111-5. DOI: 10.3990/1.9789036531115. URL: https://ris.utwente.nl/ws/portalfiles/portal/6035872/Thesis%5C_R%5C_Kader.pdf.

[133] Patrick O'Neil. *Database Systems: Principles, Programming, Performance*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994. ISBN: 1558602194.

[134] Franck Michel. "Integrating heterogeneous data sources in the Web of data". Theses. Université Côte d'Azur, 2017. URL: https://tel.archives-ouvertes.fr/tel-01508602v3/file/2017AZUR4002.pdf.

[135] Dominik Tomaszuk et al. "Document-oriented triplestore based on RDF/JSON". In: *Studies in Logic, Grammar and Rhetoric,(22 (35))* 130 (2010).

[136] Cody Cutler, M. Frans Kaashoek, and Robert T. Morris. "The benefits and costs of writing a POSIX kernel in a high-level language". In: *Proc. of the Symp. on Operating Systems Design and Implementation (OSDI 18)*. 2018, pp. 89–105.

ISBN: 978-1-939133-08-3. URL: https://www.usenix.org/conference/osdi18/presentation/cutler.

[137] Petros Tsialiamanis, Lefteris Sidirourgos, Irini Fundulaki, Vassilis Christophides, and Peter Boncz. "Heuristics-Based Query Optimisation for SPARQL". In: *Proc. of the Int. Conf. on Extending Database Technology (EDBT)*. 2012, pp. 324–335.

[138] Franck Michel, Catherine Faron Zucker, and Johan Montagnat. "A Mapping-based Method to Query MongoDB Documents with SPARQL". In: *Proc. of the Int. Conf. on Database and Expert Systems Applications (DEXA)*. 2016. URL: https://hal.archives-ouvertes.fr/hal-01330146.

[139] Vladimir Mironov, Nirmala Seethappan, Ward Blondé, Erick Antezana, Andrea Splendiani, and Martin Kuiper. "Gauging triple stores with actual biological data". In: *BMC bioinformatics* 13.1 (2012), S3.

[140] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, and T.W. Keller. "Energy management for commercial servers". In: *Computer* 36.12 (2003), pp. 39–48.

[141] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. "Octopus: An RDMA-Enabled Distributed Persistent Memory File System". In: *Proc. of the USENIX Conf. on Annual Technical Conference (ATC)*. USA, 2017, pp. 773–785. ISBN: 9781931971386.

[142] Stephen M. Rumble, Diego Ongaro, Ryan Stutsman, Mendel Rosenblum, and John K. Ousterhout. "It's Time for Low Latency". In: *Proc. of the USENIX Conf. on Annual Technical Conference (ATC)*. 2011, p. 11.

[143] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. "A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology". In: *Proc. of the Int. Symp. on Computer Architecture (ISCA)*. 2009, pp. 14–23.

[144] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. "FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory". In: *Proc. of the*

Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS). 2020, pp. 1077–1091.

[145] Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. "Persistent Memory I/O Primitives". In: *Proc. of the Int. Workshop on Data Management on New Hardware (DaMoN)*. 2019.

[146] Ismail Oukid and Wolfgang Lehner. "Data Structure Engineering For Byte-Addressable Non-Volatile Memory". In: *Proc. of the ACM Int. Conf. on Management of Data (SIGMOD)*. 2017, pp. 1759–1764.

[147] Ian Neal, Ben Reeves, Ben Stoler, Andrew Quinn, Youngjin Kwon, Simon Peter, and Baris Kasikci. "AGAMOTTO: How Persistent is your Persistent Memory Application?" In: *Proc. of the USENIX Symp. on Operating Systems Design and Implementation (OSDI)*. 2020, pp. 1047–1064.

[148] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. "Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree". In: *Proc. of the USENIX Conf. on File and Storage Technologies (FAST)*. 2018, pp. 187–200.

[149] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. "Bztree: A High-Performance Latch-Free Range Index for Non-Volatile Memory". In: *Proc. VLDB Endow.* 11.5 (2018), pp. 553–565.

[150] Xinjing Zhou, Lidan Shou, Ke Chen, Wei Hu, and Gang Chen. "DPTree: Differential Indexing for Persistent Memory". In: *Proc. VLDB Endow.* 13.4 (2019), pp. 421–434.

[151] Mengxing Liu, Jiankai Xing, Kang Chen, and Yongwei Wu. "Building Scalable NVM-Based B+tree with HTM". In: *Proc. of the Int. Conf. on Parallel Processing (ICPP)*. 2019, pp. 101–110.

[152] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. "FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory". In: *Proc. of the Int. Conf. on Management of Data (SIGMOD)*. 2016, pp. 371–386.

[153] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. "NV-Tree: Reducing Consistency Cost for NVM-Based Single Level Systems". In: *Proc. of the USENIX Conf. on File and Storage Technologies (FAST)*. 2015, pp. 167–181.

[154] Shimin Chen and Qin Jin. "Persistent B^+-Trees in Non-Volatile Main Memory". In: *Proc. VLDB Endow.* 8.7 (2015), pp. 786–797.

[155] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. "Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory". In: *Proc. of the USENIX Conf. on File and Stroage Technologies (FAST)*. 2011, pp. 1–5.

[156] Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang, and Jiwu Shu. "UTree: A Persistent B+-Tree with Low Tail Latency". In: *Proc. VLDB Endow.* 13.12 (2020), pp. 2634–2648.

[157] Eugene Inseok Chong, Souripriya Das, George Eadon, and Jagannathan Srinivasan. "An Efficient SQL-Based RDF Querying Scheme". In: *Proc. VLDB Endow.* 2005, pp. 1216–1227.

[158] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. *Basic Performance Measurements of the Intel Optane DC Persistent Memory Module*. 2019. arXiv: 1903.05714 [cs.DC].

[159] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. "An Empirical Guide to the Behavior and Use of Scalable Persistent Memory". In: *USENIX*

Conf. on File and Storage Technologies (FAST). 2020, pp. 169–182.

[160] Tobin J. Lehman and Michael J. Carey. "A Study of Index Structures for Main Memory Database Management Systems". In: *Proc. VLDB Endow.* 1986, pp. 294–303.

[161] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. "LUBM: A Benchmark for OWL Knowledge Base Systems". In: *Journal of Web Semantics* 3.2–3 (2005), pp. 158–182.

[162] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. "SP^2 Bench: A SPARQL Performance Benchmark". In: *Proc. of the IEEE Int. Conf. on Data Engineering (ICDE)*. 2009, pp. 222–233.

[163] Philipp Götze, Arun Kumar Tharanatha, and Kai-Uwe Sattler. "Data Structure Primitives on Persistent Memory: An Evaluation". In: *Proc. of the Int. Workshop on Data Management on New Hardware (DaMoN)*. 2020.

[164] Yinjun Wu, Kwanghyun Park, Rathijit Sen, Brian Kroth, and Jaeyoung Do. "Lessons Learned from the Early Performance Evaluation of Intel Optane DC Persistent Memory in DBMS". In: *Proc. of the Int. Workshop on Data Management on New Hardware (DaMoN)*. 2020.

[165] Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson, and Jishen Zhao. "Characterizing and Modeling Non-Volatile Memory Systems". In: *The IEEE/ACM Int. Symp. on Microarchitecture (MICRO)*. 2020, pp. 496–508.

[166] Burton H. Bloom. "Space/Time Trade-Offs in Hash Coding with Allowable Errors". In: *Communications of the ACM (CACM)* 13.7 (1970), pp. 422–426.

[167] Arthur C Ammann, Maria Hanrahan, and Ravi Krishnamurthy. "Design of a Memory Resident DBMS." In: *Proc. of the IEEE Compcon*. 1985, pp. 54–58.

[168] Stratos Idreos, Kostas Zoumpatianos, Manos Athanassoulis, Niv Dayan, Brian Hentschel, Michael S. Kester, Demi Guo, Lukas M. Maas, Wilson Qin, Abdul Wasay, and Yiyou Sun. "The Periodic Table of Data Structures". In: *IEEE Data Eng. Bull.* 41.3 (2018), pp. 64–75.

[169] Manos Athanassoulis and Stratos Idreos. "Design Tradeoffs of Data Access Methods". In: *Proc. of the ACM Int. Conf. on Management of Data (SIGMOD)*. 2016, pp. 2195–2200.

[170] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. "Monkey: Optimal Navigable Key-Value Store". In: *Proc. of the ACM Int. Conf. on Management of Data (SIGMOD)*. 2017, pp. 79–94.

[171] Stratos Idreos, Niv Dayan, Wilson Qin, Mali Akmanalp, Sophie Hilgard, Andrew Ross, James Lennon, Varun Jain, Harshita Gupta, David Li, and Zichen Zhu. "Design Continuums and the Path Toward Self-Designing Key-Value Stores that Know and Learn". In: *CIDR*. 2019.

[172] Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, Vijendra Purohit, Hugh Qu, Chaitanya Sreenivas Ravella, Krystyna Reisteter, Sheetal Shrotri, Dixin Tang, and Vikram Wakade. "Socrates: The New SQL Server in the Cloud". In: *Proc. of the ACM Int. Conf. on Management of Data (SIGMOD)*. 2019, pp. 1743–1756.

[173] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. "Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases". In: *Proc. of the ACM Int. Conf. on Management of Data (SIGMOD)*. 2017, pp. 1041–1052.

List of Acronyms

DMS Data Management Systems. 25, 57

IRI Internationalized Resource Identifier. 34

KGs Knowledge Graphs. 1, 22, 25–27, 29–32, 42, 52, 53, 84

LOD Linked Open Data. 5

PMs Persistent Memories. 17, 163

RDF Resource Description Framework. 3

SPARQL SPARQL Protocol and RDF Query Language. 7

Turtle Terse RDF Triple Language. 36, 37, 40

URI Uniform Resource Identifier. 34

XML Extensible Markup Language. 3