Universal Workload-based Graph Partitioning and Storage Adaption for Distributed RDF Stores

Dissertation

zur Erlangung des Doktorgrades
Doctor of Philosophy (Ph.D.)
der mathematisch-naturwissenschaftlichen Fakultäten
der Georg-August-Universität zu Göttingen

im Promotionsprogramm Computer Science (PCS)
der Georg-August-University School of Science (GAUSS)

vorgelegt von

Ahmed Imad Aziz Al-Ghezi
aus Baghdad

Göttingen
im October 2020

Printed with the support of the German Academic Exchange Service.

Betreuungsausschuss

Erster Betreuer	Prof. Dr. Lena Wiese Institut für Informatik, Goethe-Universität Frankfurt
Zweiter Betreuer	Prof. Dr. Ramin Yahyapour Institut für Informatik, Georg-August-Universität Göttingen

Prüfungskommision

Referent	Prof. Dr. Lena Wiese Institut für Informatik, Goethe-Universität Frankfurt
Koreferent	Prof. Dr. Ramin Yahyapour Institut für Informatik, Georg-August-Universität Göttingen

weitere Mitglieder

Prof. Dr. Dagmar Krefting
Institut für Medizinische Informatik, Georg-August-Universität Göttingen

Prof. Dr. Burkhard Morgenstern
Institut für Mikrobiologie und Genetik, Georg-August-Universität Göttingen

Prof. Dr. Ulrich Sax
Institut für Medizinische Informatik, Georg-August-Universität Göttingen

Prof. Dr. Armin Schmitt
Department für Nutztierwissenschaften Züchtungsinformatik,
Georg-August-Universität Göttingen

Tag der mündlichen Prüfung: 03.12.2020

Acknowledgment

I do thank God for enlightening my way. I would like to thank my family who supported, encouraged, and motivated my PhD works in the long days especially my wife and parents. I thank all my colleagues and friends who always provided me with their generous support.
I would like to thank the German Academic Exchange Service (DAAD) for granting a scholarship that provided the fund for my PhD study.

Abstract

The publication of machine-readable information has been significantly increasing both in the magnitude and complexity of the embedded relations. The Resource Description Framework(RDF) plays a big role in modeling and linking web data and their relations. In line with that important role, dedicated systems were designed to store and query the RDF data using a special queering language called SPARQL similar to the classic SQL. However, due to the high size of the data, several federated working nodes were used to host a distributed RDF store. The data needs to be partitioned, assigned, and stored in each working node. After partitioning, some of the data needs to be replicated in order to avoid the communication cost, and balance the loads for better system throughput. Since replications require more storage space, the important two questions are: what data to replicate? And how much? The answer to the second question is related to other storage-space requirements at each working node like indexes and cache. In order to efficiently answer SPARQL queries, each working node needs to put its share of data into multiple indexes. Those indexes have a data-wide size and consume a considerable amount of storage space. In this context, the same two questions about replications are also raised about indexes. The third storage-consuming structure is the join cache. It is a special index where the frequent join results are cached and save a considerable amount of running time on the cost of high storage space consumption. Again, the same two questions of replication and indexes are applicable to the join-cache.

In this thesis, we present a universal adaption approach to the storage of a distributed RDF store. The system aims to find optimal data assignments to the different indexes, replications, and join cache within the limited storage space. To achieve this, we present a cost model based on the workload that often contains frequent patterns. The workload is dynamically analyzed to evaluate predefined rules. Those rules tell the system about the benefits and costs of assigning which data to what structure. The objective is to have better query execution time.

Besides the storage adaption, the system adapts its processing resources with the queries' arrival rate. The aim of this adaption is to have better parallelization per query while still provides high system throughput.

List of Figures

2.1 RDF graph example by [42] 12
2.2 Query graph example 23
2.3 Query plan based on sort-merge join 25
2.4 Query plan based on hash-index join 26
2.5 METIS output example 32
2.6 1-hop guarantee example 34
2.7 2-hop guarantee example by [42] 35
2.8 The hash-based partitioning of graph in Figure 2.1 37

3.1 Chapter's scope 46
3.2 Components of the adaption model 49
3.3 Average number of hits per day versus the DBpedia version, as appeared on [78] 51
3.4 Percentages of queries exhibiting a different number of triples (in colors) for each dataset for Valid (left hand side of each bar) and Unique queries (right-hand side of each bar) as appearing in [12] 54
3.5 Heat query evolving from four queries 63
3.6 Heat join map evolving from four queries 64
3.7 Workload rules' maps 66

4.1 Chapter's scope 70
4.2 Average hard disk price through the years 1980-2019 as appeared in [52] 71
4.3 Average hard disk price through the years 2015-2019 as appeared in [52] 71
4.4 The map of indexes' rules 80
4.5 The map of cache index rule 82
4.6 The Running times of adaptable indexes and join cache vs fixed approaches under storage capacity of 6 84
4.7 The Running times of adaptable indexes and join cache vs fixed approach under storage capacity of 3 85

5.1 Chapter's scope 92
5.2 The map of replications' rules 102

6.1 Chapter's scope 108
6.2 Abstract system architecture 110
6.3 The process of storage space adaption 111

7.1 The systems' performance comparison of the 12 runs 127
7.2 Short heterogeneous queries vs capacity 129
7.3 Short non-heterogeneous queries vs capacity 130
7.4 Long heterogeneous queries vs capacity 130
7.5 Long non-heterogeneous queries vs capacity 131
7.6 The response of the systems towards non-uniform workload access with respect to capacity 132

8.1 Speedup of bounded-queries execution with respect to working threads 141
8.2 Speedup of unbounded-queries execution with respect to working threads 142
8.3 The systems' performance comparison of queries stream 145

List of Tables

2.1 Basic index types notations 20
2.2 Basic hashed indexes notations 21
2.3 The most related systems which employ workload adaption 42

4.1 The running time of a single-triple query that uses SPo index with respect to different data sizes 87
4.2 The running time of a single-triple query that uses PSo index with respect to different data sizes 87
4.3 Bounded chain query behavior with data set size 88
4.4 Unbounded chain query behavior with data set size 88

7.1 Parameters of runs 1-4 with systems' running times 123
7.2 storage distribution of runs 1-4 (in millions of triples) 123
7.3 Storage distribution of runs 5-7 125
7.4 Parameters of runs 5-7 with systems' running times 125
7.5 Storage distribution of runs 8-11 126
7.6 Parameters of runs 8-11 with the systems' running times 126
7.7 Workload Properties of The Non-uniform Workload 132

8.1 Bounded-queries speedup with respect to working threads 141
8.2 Unbounded-queries speedup with respect to working threads 142
8.3 The speedup with respect to border triples 144
8.4 The speedup with respect to border triples with full replication . . . 144
8.5 The query streams runs specifications 145

Contents

1 Introduction **1**

1.1 Problem and Motivations 2
1.2 Our Solution 5
1.3 Thesis Contributions 6
1.4 Thesis Structure 6

2 Background **9**

2.1 Resource Description Framework (RDF) 10
2.1.1 Overview 10
2.1.2 The Data Model Object Types 10
2.1.3 Resources and Objects Naming 10
2.1.4 RDF Graph 11
2.1.5 RDF Vocabularies 12
2.1.6 Serialization Format 13
2.2 SPARQL 14
2.3 Triples Stores 16
2.3.1 Non-Native DBMS-based Approaches 16
2.3.2 Native RDF Storage Approaches 17
2.4 RDF Indexing 17
2.4.1 Key-value indexes 18
2.4.2 Graph-based indexes 19
2.5 Index Notation 20
2.6 SPARQL Queries Processing 20
2.6.1 The Bounding of Queries 21
2.6.2 Conceptual Execution 22
2.6.3 Data Access Paths 22
2.6.4 Join Evaluation 24
2.7 Distributed Triples Store 27

2.7.1 Distributed Storage and Indexing . . . 28
2.7.2 Data Partitioning . . . 30
2.7.3 Graph-based Partitioning . . . 30
2.7.4 Hash-based Partitioning . . . 34
2.7.5 Data Partitioning Summary and Conclusion . . . 36
2.7.6 Discussion . . . 38
2.8 Most-Related Work . . . 38
2.9 Summary . . . 41

3 Workload Analysis 45
3.1 Why Adaption? . . . 46
3.2 Universal Adaption . . . 47
3.2.1 The Cost Model . . . 49
3.2.2 The Resources' Access Rate . . . 50
3.3 The Role of the Workload . . . 51
3.3.1 Real-world Workload Analysis . . . 53
3.3.2 Evaluation Locality . . . 54
3.4 Workload Rules . . . 56
3.4.1 Basic Measurements for The General Rules . . . 58
3.5 Heat Queries . . . 59
3.5.1 Heat Query Generation . . . 59
3.5.2 Implementation Notes . . . 60
3.5.3 Generalized Rules . . . 61
3.5.4 Heat Query Anonymization [1] . . . 62
3.5.5 Triples Access Rate By Heat Queries [2] . . . 63
3.6 Heat Query Specific Rule . . . 65
3.7 Summary . . . 66

4 Local Storage 69
4.1 Storage Scarceness . . . 70
4.2 System Storage Hierarchy . . . 72
4.3 Indexes . . . 72
4.4 Problem of fixed Indexes . . . 73
4.5 Dynamic Indexes . . . 74
4.6 Indexes in The Cost Model . . . 75

[1] Part of this subsection appeared in our publication [3].
[2] This derivation is also given in our publication [3].

4.6.1 Index Cost . . . 75
4.6.2 Index Benefit . . . 76
4.6.3 Index Access Rate . . . 77
4.7 Index Rules . . . 78
4.7.1 Index General Rules . . . 78
4.7.2 Index Specific Rules . . . 78
4.8 Index Rules Aggregation . . . 79
4.8.1 Finalizing Index Rules . . . 79
4.9 Cache Index . . . 80
4.9.1 Cache-index Specific Rules . . . 81
4.10 Dynamic Indexes Evaluation . . . 82
4.10.1 Detectable workload and High storage space availability . . . 83
4.10.2 Scalability of Queries Processing . . . 84
4.11 Summary . . . 88

5 Distributed Storage and Replication **91**
5.1 Replication Motivations . . . 92
5.2 Distributed RDF Storage . . . 93
5.3 Initial Graph Partitioning . . . 94
5.3.1 METIS based Partitioning . . . 95
5.4 Border Region . . . 98
5.5 Border Replication . . . 99
5.5.1 General Border Access Rule . . . 99
5.5.2 Specific Access Rule . . . 100
5.5.3 Aggregating Border Replication Rules . . . 101
5.6 Load-balancing Replication . . . 101
5.6.1 Load-balancing Replication in The Cost Model . . . 102
5.6.2 Load-balancing Replication Rules . . . 103
5.7 Replication Aggregated Rules . . . 104
5.8 Summary . . . 104

6 Universal Adaption **107**
6.1 System Architecture . . . 108
6.2 Storage Space Optimizer . . . 109
6.2.1 Universal Adaption . . . 110
6.2.2 Better Algorithm: Rules-based Space Adaption Algorithm . . . 114
6.3 Creating The Proposed and Assigned Rules . . . 116

6.4 Summary 117

7 Universal Adaption Evaluation 119

7.1 Generation of Data-sets and Queries 120

7.2 Data-set size 120

7.2.1 System Capacity 121

7.3 Universal Adaption 121

7.3.1 Starting point 121

7.3.2 Adaption Parameters 122

7.3.3 Non-frequent Workload 127

7.3.4 Non-uniform Workload to Partitions Access 129

7.4 Summary 131

8 Threading 135

8.1 Adaption to Queries Arrival Rate 136

8.2 Queries Queuing Model 136

8.3 Adaption of The Processing Resources 139

8.4 Evaluation 139

8.4.1 Working Threads 140

8.5 Distributed Working Nodes 142

8.5.1 Queries Stream 144

8.6 Summary and Conclusion 146

9 Conclusion and Future Work 147

9.1 Points of Strength 148

9.2 Limitations/Points of Weakness 149

9.2.1 Overheads 149

9.2.2 Worst Cases Scenarios 150

9.2.3 Partitioning Limitations 151

9.3 Future Works 151

9.3.1 Partitioning 151

9.3.2 Workload Analysis 152

9.3.3 Optimization's Overheads 152

9.4 Summary 152

A Basic Theoretical Foundations 155

A.1 Queries Shape 155

A.1.1 Star Queries 155

A.1.2 Chain Queries . . . 156
A.1.3 Tree Queries . . . 156
A.1.4 Cyclic Queries . . . 156
A.1.5 Queries Length, Size, and Evaluation Size . . . 156
A.2 Workload Quality . . . 157
A.2.1 The Basic of The Adaption Algorithm . . . 158
A.3 Index on Hard Disk . . . 159
A.3.1 Access Time . . . 159
A.4 Triples in Main Memory . . . 160

B Mathematical Symbols **163**
B.1 Mathematical Symbols Used in Chapter 3 . . . 163
B.2 Mathematical Symbols Used in Chapter 4 . . . 165
B.3 Mathematical Symbols Used in Chapter 5 . . . 166

Chapter 1

Introduction

This chapter introduces the Resource Description Framework (RDF), presents the thesis's problem statement, and summarizes its contributions. It ends with outlining the main structure of the thesis.

Contents

1.1 Problem and Motivations **2**

1.2 Our Solution **5**

1.3 Thesis Contributions **6**

1.4 Thesis Structure **6**

1.1 Problem and Motivations

The Resource Description Framework (RDF) [34] has been widely used to model the data on the web. Despite its simple triple-based structure, RDF showed a high ability to model the complex relationships between the web entities and preserve their semantic. It provided the scalability that allowed the RDF data to grow big from the range of billions [19] to the range of trillions of triples [62]. The naming rules of Tim Berners-Lee [11] defined the methodology to provide a unique URI-based name to each *thing* modeled by an RDF data-set. This allowed data from different sources to be linked into one big cloud of linked RDF data [79] and enabled querying this cloud. Accompanied with Web Ontology Language (OWL), the RDF graph represents a big knowledge graph [8]. That enables the web to build an “understanding” of human knowledge, and evolve its applications. The medical and health semantic knowledge graphs are important examples in this regard [73, 1, 20]. As a result, RDF data experienced a rapid increase both in the size and complexity of the embedded relationships [22]. To keep up with that increase, specialized and dedicated systems have appeared to store the RDF triples and provide the service of querying them. However, these systems had to deal with many challenges regarding the management of such big data, and efficiently process their queries. This management operation requires many data structures including multiple data-wide indexes, replications, dictionary, statistics, and materialized queries results. In the context of the huge RDF data size, these structures put the RDF system in extreme storage space requirements which become even more challenging in a main memory environment.

RDF Indexing

One of the most important challenges is how the data should be indexed to provide the required efficiency of query answering, while at the same time, trying to avoid the high storage overhead coming from data redundancy in indexes. The indexing in RDF triple stores emerges as a hot research topic as the queries evaluation was feasible only with the existence of the required indexes. However, the objective of indexing is always to decrease the query execution time, and the constraint is the extra storage space. The system’s index needs are tightly related to the workload trends, and the storage constraint is related to the ratio of space availability to data-size, besides the space needs of other data structures in the system. Unfortunately, all the known triple stores made a fixed design choice regarding the objective and constraint of the indexes and thus used a fixed design scheme which the system had to live with. Some of them were very space conservative like Stratustore [75] who

used only one index, and others were very federated like RDF-3X [56] and Hexastore [81] that used more than six indexes, while others preferred to stay average and use three indexes like Rya [64], MAPSIN [69], and AMADA [15]. However, it is only suitable to evaluate the performance of these systems under fixed circumstances of workload and space. For example, the single index of Stratustore could show enough performance if the workload is merely single-type such that it only requires the SPO index. On the other hand, a diverse workload might need the comprehensives indexes of RDF-3X to provide the expected performance, but only if the system can assign the required space. Space availability is highly related to the data size and the system's space requirements which are far from being fixed parameters.

Data Replication

As the size of RDF sources is rapidly increasing, the resources of the centralized systems have been facing difficulties in maintaining such big data and efficiently querying them. This highly motivated the move toward a distributed RDF triple store where several working nodes are cooperating in storing and querying the global RDF data set. However, this move has marked more challenges. The RDF data set which is also modeled as a graph needs to be partitioned such that each working node receives at least one partition. In this case, a query that needs data from more than one working node needs to pay the communication cost, which is the network cost required to move data across the physical network. This data can be relatively big and may overwhelm most of the total query execution time. There exist two main directions to overcome the cost of these intermediate results:

1. Performing better partitioning to decrease the size of queries' cross-nodes intermediate results.

2. Supporting the initial partitioning by replications.

Recalling the complexity and linkage of the RDF data-sets, performing an optimal partitioning as mentioned in the first point is a difficult task. For this reason, Partout [26] proposed to adapt the partitioning with workload using initial workload-sample at system startup. Unfortunately, the performance badly degrades when the workload does not keep the same trend as the used sample. Thus, the attention shifted towards supporting the initial partitioning with replication. Instead of moving the queries' intermediate results across the network, a working node may find the needed data in replications and avoid the expensive communication cost. However, replication consumes more storage space, and there should be a wise decision about the

triples to be replicated in order to increase the ratio of replication utilization. Since the replication is performed to support the partitioning, the utilization of the replication depends on the strategy of the used partitioning (e.g graph partitioning or hash-based partitioning). Moreover, the replication is highly related to the workload, because the shape, length, locality, and arrival rate of the queries determine which triples are highly needed for replication. Considerable work has been done to utilize the replication (works are reviewed in Chapter 2), where part of the works considered using a workload history to identify the more important data for replication and aim to save storage space. However, all of the related works either assume the existence of some initial workload, or fixed parameters and thresholds which are not clearly connected or calculated from the workload. In spite of that the storage space is already identified as the replication constraint, non of the related work has implemented the adaption as a function such that it is dynamically delimited by the space. In this context, if the data size happens to be small or big compared to the available storage size, the given systems have no ability to replicate more or fewer data accordingly.

Universal Adaption

Assume that a working node has a given limited amount of unused space, should the node employ it in building more replications or use it to support its local indexes? The objective function of the replication is to decrease the queries execution time by avoiding the communication cost of the intermediate results as well as balancing the load between the working nodes. The constraint is again the storage space. Recalling what we have introduced earlier about the indexes objective and constraint, we can identify that the indexes and the replication share the same objective and constraint. As a matter of fact, building more indexes can be seen as replicating data locally for faster processing, while replication is replicating remote data for faster access. This makes a clear baseline for a single optimization operation that considers both of them in the same domain. Moreover, replications and indexes are not the only storage consumers in an RDF triple store. Materializing queries results or caching some of the join operations provides considerable benefit under the usual workload environment. These cached results share also the same objective and constraint of the indexes and replications, and thus they can flexibly fit in the universal adaption model.

Workload Analysis

As was earlier pointed out by [26, 37, 60, 31], the historical workload can be an effective tuning subject for the system resources to have a more efficient future workload.

However, there should be no fix assumptions about the RDF workload [12] as the workload properties change values with many practical factors like the data-sets, the applications and the temporal factors. To deal with this dynamic workload status, the system should adapt its analysis to the workload and measure its effectiveness in order to increase the impact of the effective parameters and obliterate the impact of those with low effectiveness.

1.2 Our Solution

Workload Analysis

Our system collects the queries, normalizes them to remove outliers, and transfers them into a set of queries graphs where common items are shared and connected. The system keeps the frequency values of these common items within each structure which we call heat query. The heat query keeps also the count of each index usage at each item. An anonymization process is used to generalize a heat query to more data within the RDF graph. The effectiveness of this generalization is measured and tuned to avoid bad influence. The workload analysis assumes no fixed thresholds or setting. The set of heat queries provides the probability of access to the RDF triples inspired by the workload. The system keeps a set of predefined rules. The rules are well designed and formalized such that any new rule can easily be plugged into the optimization system. Moreover, to allow the workload adaptability, the rules have two types: general rules which are based on the statistics of the average behaviour, and specific rules which are based on the specific behaviour drawn by the heat queries. The general rules represent a backline to support the system in case of that the workload was on low-quality levels.

Universal Adaption

The optimization process divides the storage space into units such that each unit is seen as a resource. Each resource can be utilized (consumed) by a block of triples with the same size, which is located either locally or remotely, and this utilization can be on one of the different indexes options. The workload is analysed and used to assign a benefit to each consumer-option pair. Since this pair has a known space cost, we result in a concrete cost model that can utilize each storage unit with the

best option of structured data.
The above cost model can be directly extended to include another storage consumer which is the materialized queries results or cached join results. These cached data might give an extreme benefit to the total throughput of the system and the queries execution time, especially in the case of the existence of small and hot frequent patterns. Such case is detected for instance in a real-world scenario where more than 90% of the queries target only 163 frequent sub-graphs [60]. However, the storage cost and performance benefit of such cached data are integrated into the cost model and optimized with the indexes and replication.

1.3 Thesis Contributions

This thesis presents the problem of universal adaption in distributed RDF triple stores and its impact on the performances of queries execution. This main contribution is composed of the following points:

1. We formulate a dynamic and integrated cost model for indexes, replication, and cached join results, where the benefits and costs of each structure are comparable in the same domain.
2. We provide a workload analysis approach that is adaptable with the workload quality and requires no fixed thresholds.
3. We present UniAdapt, a distributed triple store that implements the universal adaption of its storage layer with both of the workload and storage space.
4. The thesis provides diverse practical evaluations to the universal adaption focusing on the areas where this approach is highly performing as well as the areas where this adaption is difficult to show differences.

1.4 Thesis Structure

The thesis is structured as follows. Chapter 1 is this introduction. Chapter 2 starts with the foundations. We introduce RDF structure, maintenance, and processing. We review the related works focusing on the distributed approaches that considered the problem of RDF graph partitioning and replications. We then focus the review on the most related works that considered adaption. Chapter 3 considers the analysis of the workload, the formulation of the adaption problem, and its cost model.

The workload is structured and analyzed by the concept of heat query and average statistics. We present the concept of access and operational rules. In Chapter 4, we consider the local storage adaption in terms of the indexes and join cache. We define their benefit and cost functions and transfer the cost model into operational rules. The chapter concludes with a practical evaluation of the dynamic indexes and cache join approaches. In Chapter 5, we present the distributed system architecture and the replications problem. We consider two types of replications and define their access rules. The two rules are aggregated into one operational rule that represents the replication. That rule is comparable with the operational rules of both the indexes and join cache. The optimization process based on the three rules is carried out in Chapter 6. An efficient rules-based universal adaption algorithm is presented. Chapter 7 shows our evaluation results to the universal storage adaption and its impact on the performance under varying workload environment parameters and scenarios. Chapter 8 considers the adaption of the local and processing resources with the queries arriving rates, aiming for a better query execution time. Chapter 9 concludes the thesis, discusses the points of strength as well as weaknesses, and provides the directions of future works.

Chapter 2

Background

This chapter presents the foundations of the thesis. It provides the essential background knowledge on which the following considerations are based. We provide an overview about RDF as a data model and the specifications of its standard query language SPARQL. We then introduce the requirements and structure of RDF-triples stores, giving special focus on their storage layer, where the RDF indices are built and where the main part of query processing takes place. We then state the main challenges of moving the storage layer towards a distributed environment. While we provide a review of the literature and related works during the chapter's sections, we provide more detailed descriptions and issues of the works which considered the workload adaption.

Contents

2.1 Resource Description Framework (RDF) 10
2.2 SPARQL 14
2.3 Triples Stores 16
2.4 RDF Indexing 17
2.5 Index Notation 20
2.6 SPARQL Queries Processing 20
2.7 Distributed Triples Store 27
2.8 Most-Related Work 38
2.9 Summary 41

2.1 Resource Description Framework (RDF)

2.1.1 Overview

The RDF in general is a model to represent data. Its basic idea is to make statements about resources using a triple based format. Each triple is in the form of (subject, predicate, object). The subject represents a certain resource given by a textual identifier that is unique within a data set. The object either denotes another resource or a constant, while the predicate states a certain relationship between the subject resource and the object resource/constant. As an example, the piece of information that is embedded in the following phrase: *"Newton was born in England"*, can be modeled using RDF by the following triple (:newton, :was_born, :england). The given triple states one fact about the resource :newton; however, since the object :england is also a resource, it can have triples on its own, where it appears as a subject, and further facts can be related. For instance (:england, :located, :europe). This methodology of stating information about resources makes the RDF very suitable to represent web resources and their relations, in a way that is compact and efficient in terms of storing, exchange and querying.

2.1.2 The Data Model Object Types

The basic RDF data model consists of three data types:

- Resource: which is the *"thing"* described by any RDF statement. It can be a web page, a part of a web page, a certain file resource, an entire website, or not directly accessible resources such as a printed book. In the triple format, the resource can be placed as Subject or as Object.

- Properties: is a specific aspect, characteristic, attribute, or relation used to describe a resource. It reflects this role in the predicate position of triple format.

- Statements: is a resource plus the property describing it, and a value. The value can be either another resource or a literal. A statement represents an RDF triple in the form *subject, predicate, object*.

2.1.3 Resources and Objects Naming

One of the important specifications of RDF as a data model, is how resources are represented or identified and characterized. The W3C in the 1999 RDF recommen-

dation[1] uses the *Universal Resource Identifier* or URI to clearly and uniquely identify any resource within any domain. However, it is directly possible to use the URLs (Uniform Resource Locators) for the same purpose as they are essentially a subset of URIs and each URL is ensured to be unique within the web as the domain name part within a URL is globally unique. Using URLs for resources naming enabled the move one step further by introducing the concept of Linked Data, where RDF triples from different sources can be combined, stored and queried.

Besides the resources, the data model allows Literals to be used as values in a triple's object. They are either plain (with an optional language tag) or typed. A typed literal is annotated with a datatype URI, e.g. the commonly used XML Schema datatypes.

The third type of value might be found within an RDF data set which is the Blank Nodes. They represent anonymous resources that are used if an entity is only used in a local context, e.g. a relation between two entities is modeled as a *blank node* with specific attributes that specify the relationship in more detail. The identifiers of blank nodes are only defined for the local scope of an RDF graph. Therefore, they are not unique and cannot be used in a global context.

2.1.4 RDF Graph

Since a triple represents a semantic relationship between two resources, a set of triples can be directly modeled as a graph, where each resource is modeled as a vertex, and each edge represents a labeled relationship between two vertices if a corresponding triple exists in the triples set. The edge's label is the triple's predicate. We can formally state the definition of the RDF graph as follows:

Definition 2.1 (RDF Graph) *Let $G = \{V, E, P\}$ be a graph representing the RDF data set. V is a set of all the subjects and objects in the set of RDF triples D; $E \subseteq V \times V$ is a set of directed edges representing all the triples in the data set; P is a set of all the edges' labels in the RDF data, and we denote p_e as the property associated with edge $e \in E$. The RDF data set is then defined as $D = \{(s, p_e, o) \mid \exists e = (s, o) : e \in E \wedge p_e \in P\}$*

The mapping of an RDF data set to a mathematical graph is a very important step with respect to the methods of the data management since all the graph algorithms can be directly applied. For example, the problem of RDF partitioning can be mapped to a graph partitioning problem.

[1] https://www.w3.org/TR/1999/REC-rdf-syntax-19990222/

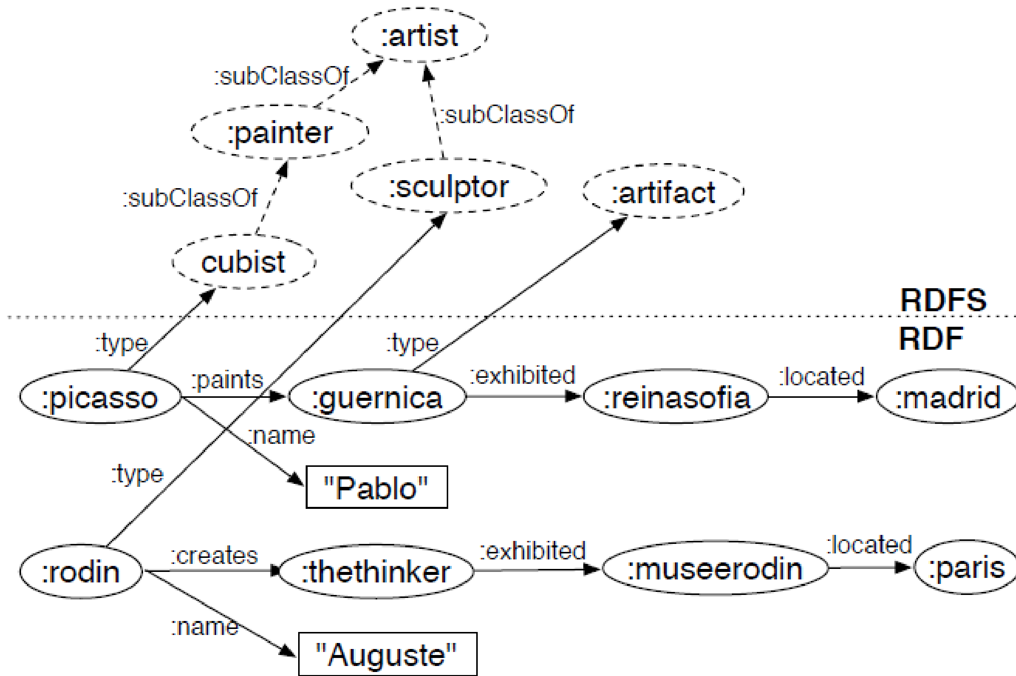

Figure 2.1: RDF graph example by [42]

From Definition 2.1, each $e \in E$ can be mapped to exactly one triple $d \in D$. Moreover, each $v \in V$ can be mapped to a list of edges, and that can be mapped to a list of corresponding triples. We define the functions which perform these mappings in the following definition.

Definition 2.2 (Mapping of Graph Elements to Triples) *We define $mapToTriple(e)$ as the function that maps any given edge $e \in E$ to its corresponding triple $d \in D$. In addition, we define $mapToTriples(v)$ as the function that maps any vertex $v \in V$ to its corresponding list of triples.*

2.1.5 RDF Vocabularies

The ability of RDF to represent the semantic of information is one of the most important properties that makes it heavily used to model web data. The RDF depends on its standard vocabulary to simplify the storing and extracting of hidden relations that build the semantic. Such vocabularies are basically defined by the RDF Vocabulary Description Language (RDF Schema) [13] and the Web Ontology Language (OWL) [51], as classes, properties, and the relations between them.

The RDF schema (RDFS) allows the user community to extend the vocabulary by adding a set of predefined classes, where any new class is an instance of a previously given class similar to the Object-Oriented Paradigm. The **rdfs:class** is the parent of all classes, and any class in the schema is eventually rooted to it by the property **rdfs:subClassOf**. Any class that describes the relation between two RDFS classes or the relation between a resource and an RDFS class is called property, and it must be a subclass of **rdfs:property**. For instance, **rdfs:type** is very important property used to state that a certain resource is an instance of a defined class. Consider in this regards the RDF triple: **ex:JeffPollock rdf:type ex:Person**, which states that the resource **ex:JeffPollock** is a person.
Some of the well known RDF vocabularies used to describe RDF documents are: *Friend Of A Friend (FOAF)* [2] and Dublin Core [3]. FOAF is used to describe people and their personal information and provides vocabularies for things like name, address, and occupation. Dublin Core defines necessary vocabularies for describing metadata of documents like the title of publication, date of publication, and author related information.

The RDFS is directly mapped into a graph that is connected to the main RDF graph. However, it is often necessary for the user to understand the basic shape of the RDFS related to the target RDF data set in order to write correct SPARQL queries. Thus, it is a requirement that each RDF data set is accompanied by a well-structured and a small-sized RDF schema graph.

2.1.6 Serialization Format

The RDF data set can be conceptually represented as a graph. However, in order to maintain the data set in a textual format that is suitable to be stored as files, the W3C has different standards to serialize RDF triples. We survey the most popular serialization types in the following:

RDF/XML

The first serialization format defined by W3C [27] followed the well-known XML format. Although XML is widely used to serialize documents on the web and easily interpreted by different platforms and tools, it is hard to read by humans, and the XML-tree is not naturally compact enough when used to represent the RDF graph. An example of an RDF/XML document is shown in Listing 2.1.

[2] http://www.foaf-project.org/
[3] http://dublincore.org/

Listing 2.1: RDF/XML Example

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax#"
 xmlns:exterms="http://www.example.org/terms/">
<rdf:Description rdf:about=
 "http://www.example.org/index.html">
<exterms:creation-date>August 16,1999</exterms:creation-date>
</rdf:Description>
</rdf:RDF>
```

Turtle

Another RDF serialization format is called by W3C as Turtle [10]. It is more suitable to represent the concept of triples. It is highly compact, such that a human can easily interpret the triples by directly investigating the documents. Also, the format is easily interpreted by a turtle parser. The header of a turtle document contains the list of prefix name-spaces defined by the keyword *@prefix*. Each prefix defines a short name-space for a long URL, which allows the use of the name-space as prefix anywhere in the document. This highly saves space and simplifies document reading by humans. An example of a turtle document is shown in Listing 2.2. The example shows the header of the document and three triples separated by semicolons. Turtle is derived from a more general notation called N3[4]. Thus the turtle file is usually ended with the extension *.n3*.

Listing 2.2: RDF Turtle Example

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax#>.
@prefix dc: <http://purl.org/dc/elements/1.1/#>.
@prefix exterms: <hhttp://www.example.org/terms/>.

<http://www.example.org/index.html>
exterms:creation-date "August 16, 1999";
dc:language "en";
dc:creator <http://www.example.org/staffid/85740>.
```

2.2 SPARQL

SPARQL [63] is the W3C official language used to query RDF-based data. It was designed conceptually to the semantic web, where it hides the details of internal data

[4]https://www.w3.org/TeamSubmission/n3/

management. "Trying to use the Semantic Web without SPARQL is like trying to use a relational database without SQL.", explained Tim Berners-Lee, W3C Director. The general shape of the SPARQL is inspired by the shape of the classical SQL, starting with the query keyword, the attributes, and the test or condition tail given by the keyword *WHERE*. However, while the shapes of SPARQL and SQL are similar, there are essential differences :

- The SPARQL query is expected to run against the whole data set, thus there is no table name field.

- The attributes' names projection in the SQL query takes place within the *WHERE* clause in the SPARQL query while the variable list is only a selection from the variables that appear in the *WHERE* clause.

The *WHERE* clause in a SPARQL query is a set of so-called *triple patterns*. This part of the query is the important part where most of the query evaluation is carried out. In the context of this thesis, we refer to the SPARQL query as a set of triple patterns. Each triple pattern is composed of three elements; each element is either a variable or a constant. The variable is recognized by the prefix ? followed by the variable name. Each triple in the data set can be checked against a query triple pattern and return a binary result of either true, if the constants of the triple pattern exactly match the value and order of the checked triple, or false otherwise. A variable in the triple pattern may match any value in its position within the checked triple.

For a set of n triple patterns, we get respectively n sets of matching triples. Since the triple patterns contain variables, then for any two triple patterns that share the same variable name but with different locality, we need to perform join between their set of matching triples on the location of shared variables. For instance, if $t_1 = \{c1, c2, ?a\}$ and $t_2 = \{?a, c3, c4\}$ are two triple patterns in the same query, and if we have the sets A_1, A_2 of matching triples for t_1 and t_2 respectively, we need to further join A_1 and A_2 and produce the triples that have a match on the value of the variable $?a$. We further detail the SPARQL execution in Section 2.6.

Listing 2.3: SPARQL Query Example

```
SELECT ?name ?city
WHERE {
?who <Person#fname> ?name ;
<Person#addr> ?adr .
?adr <Address#city> ?city ;
<Address#state> "Berlin"
}
```

2.3 Triples Stores

An RDF triple store is a specialized database for storing, processing and retrieval of the RDF triples. There are two general groups of triple store systems: *DBMS based approaches* and *Multiple indexing frameworks*.

2.3.1 Non-Native DBMS-based Approaches

The typical ground for any data storage system is the well-known relational database management systems, because such systems are heavily studied, researched, and optimized for high performance of queries that target relation data maintained in tables. Thus, the initial RDF stores relied on a classical relational DBMS such as MYSQL and ORACLE. SPARQL processing layer is built on the top of these systems, where it maps the user SPARQL queries to classical SQL queries. The data is physically stored in tables and indexed by the systems' classical indexes. Such approach is called *Triple Table*. However, a lot of work aimed to provide methods to enhance the efficiency, robustness, and scalability of these systems. The enhancement works mainly fall in two aspects: *property table* and *vertical partitioning*. We briefly describe the triple table and both of its relating approaches.

Triple table

The data schema is simply one table that has three columns: subject, predicate, and object. Each triple could then directly fit in the table. Indexes are built on top, to make the self join on the single table less expensive. Since there is only one table, the system can't make use of its most optimization techniques originally ready to serve multiple relational tables. As a result, such systems may be feasible either to run simple statement-queries, or to host small data set size; since the size of the single table may easily grow very large, and the query with multiple triple patterns would

require a lot of costly self-joins.
However, multiple systems used this approach like 3store [32], Redland [9], rdfDB [67], and commercial systems like Oracle [17].

Property table

Instead of the single table system explained above, some systems like Sesame [14], Jena2 [82], RDFSuite [4] and 4store [33] define a property table, in which subjects that have similar properties are clustered into tables, where the fixed properties are defined as columns, and the stored values are the objects. The direct advantage of having these properties tables is to avoid the expensive joins of triple patterns on the subject. However, the drawbacks are to have many NULL values as not all subjects have values for all the properties in a table. Another problem is when a triple pattern has the property as a variable; such a case would require the scan of all tables. Moreover, costly unions and joins might be required for the processing of queries that target several property tables.

Vertical partitioning

swStore [2] suggested to have one table per property in the data set, where each table has two columns for the corresponding subject and object. The vertical tables are maintained in column store [77] to speedup processing of all subjects or objects of some given property. This approach is more compact and flexible than the property table, however, we would see in the next section that the SPO index outperforms both of the property table and vertical partitioning approaches.

2.3.2 Native RDF Storage Approaches

To overcome the shortcomings of using the classical database management systems to handle the RDF data, native RDF triple stores were developed. The native stores are specifically designed for storing RDF data and its special structure. The storage layer and query processing engine are optimized to serve the RDF needs. The rest of this chapter focuses on the methods of indexing and query processing used in the native RDF system while paying special attention to the native distributed systems.

2.4 RDF Indexing

As we have earlier mentioned, the initial systems that dealt with the RDF modeled data used the classical relational DBMS. However, it was soon found that the

efficiency of such systems would be feasible only with the support of well designed indexes. The later native RDF stores were basically classified by their indexes structure, and to some extent, it is not possible to differentiate between the indexes, the tables, or any other data container in the system, as the whole RDF data are actually stored in indexes. The RDF indexing approaches can be classified here into: *key-value indexes* and *Graph-based indexes.*

2.4.1 Key-value indexes

The main objective of the index design is to speed up the query processing by decreasing the cost of joins and providing fast triple data retrieval. The SPARQL query in section 2.2 is defined as a set of triple patterns. The system should be able to provide the answer to any triple pattern using its indexes. Each triple pattern is a set of exactly three elements, where each element can be either a constant or a variable, given that we have at least one constant and at least one variable (excluding rare cases where a triple pattern may have zero variables). An optimal index would receive a triple pattern and return all the triples in the data set that matches it. For this purpose, the constant, or a combination of two constants in the triple pattern are used as the index's key, and the index should deliver the triples that match as output. Consider for instance the following triple pattern: t_1 = (:newton, ?y, ?x). This pattern has one constant that is in the location of the subject, and two variables in the locations of the predicate and object. In order to evaluate t_1, we need an index that has the subject as key (it is usually called S index). We perform a lookup on that index using the key ":newton" expecting the index to return a list of all triples that have ":newton" as subject. However, if the t_1 =(:newton, :was_born, ?x), then we need an index that has both the subject and predicate as key (usually called SP index). In an extreme case, the triple patten may have three constants as t_1 =(:newton, :was_born, :england). In this case, the required index should use subject, predicate and object as key (usually called SPO). Depending on the implementation, the SPO index may answer all the three triple patterns.

The implementing of the index preformed using two main approaches: sorted list and hash table. The sorted index is a list that contains all the triples such that they are sorted in the order of the key. The SPO index is then sorted on the subject, then on the predicate and finally on the object. In this manner, the SPO index may answer all the three triple patterns about Newton. The behaviour of the lookup operation in such index is logarithmic on the data size.

The hash index is performed by using a hash table that contains all the triples hashed on the key. In such a case, the SP index uses a key that is a combination of the subject and the p, and both of them must be given to perform a lookup operation. Thus, we need three hashed indexes to answer the given three triple patterns about Newton. However, the hashed index is faster and in average has a constant time-behaviour with respect to the data size. To have the benefits of the fast data access and low storage space, a hybrid indexes are used. This index is hashed on the first element of the key and sorted on the second and the third. An SPO index of such type can answer the Newton's three triple patterns. The index is hashed on the subject and thus may lookup any subject in constant time, and any of its predicates in logarithmic time since it is sorted on the predicates.

Depending on which of the triple's elements are the key, we may have 6 indexes types (Assuming hash-sort index): SPO, SOP, OPS, OSP, POS, and PSO. RDF-3X [56] builds all the given six types of indexes, allowing high index efficiency and flexibility in answering any triple pattern. However, due to the high storage overheads of having full-set indexes, some systems preferred to only build the most referenced indexes, identified as SPO, POS, and OSP which are maintained by typical key-value stores in separate containers. As a contrast to the hashed-based indexes, fully sort-based indexes can be used to enable the process of range queries more efficiently. As an example, the sort-based SPO index is built by sorting the triples on S then on P and O.

2.4.2 Graph-based indexes

Another method of RDF storage is by holistically storing an RDF data set as a graph. Each unique subject or object in the data set is a vertex that is associated with one adjacency list for the outgoing edges, and another list for the incoming edges. Each property edge in this regard is listed in the outgoing edges of its subject, and in the incoming edge of its object. This allows the literal graph processing of queries as is shown in Section 2.6.

However, on the practical aspect, the system still needs a general hash index that looks up the vertex of any subject and object in the data set. Thus, we can still consider the two lists besides the general index equivalent to the SPO and OPS indexes which were explained in the previous section. Such a storage approach is followed by Trinity.RDF [85] which is built on Trinity [72], a key-value store that serves as a distributed graph processing system.

2.5 Index Notation

In this section, we fix our notation to the different index types that the system may use. The notation should provide information about which combination of the triple's elements (S,P,O) should be used as the key, and what is the index type (i.e. sorted or hashed). In this context, any index type is given three letters, each letter can be either S, P, or O. The letter that is part of the index key is written in the capital form, and written in the left side of the index type. The index is always hashed on the first letter of the key. However, if the key contains more than one letter, it can be either sorted or hashed on them. In order to enable the notation of differentiating the hashing or sorting state, we insert the character '-' at the end of the key part in the notation. Table 2.1 shows the notation of the basic six index types which hashed on the first element and sorted on the second, besides one example of the notation in case the index is also sorted on the third element. Table 2.2 shows the notation of the basic indexes in case the index is hashed on both of the first and second elements.

SPo	The index is hashed on S and sorted on P. A key requires a constant value for S and optionally for P.
SOp	The index is hashed on S and sorted on O. A key requires a constant value for S and optionally for O.
PSo	The index is hashed on P and sorted on S. A key requires a constant value for P and optionally for S.
POs	The index is hashed on P and sorted on O. A key requires a constant value for P and optionally for O.
OPs	The index is hashed on O and sorted on P. A key requires a constant value for O and optionally for P.
OSp	The index is hashed on O and sorted on S. A key requires a constant value for O and optionally for S.
SPO	Similar to SPo except that the index is sorted also on O. A key requires a constant value for S and optionally for P and O.

Table 2.1: Basic index types notations

2.6 SPARQL Queries Processing

In this section, we state the general methods of SPARQL query evaluation in centralized systems, as a basis for describing in more detail, the methods of queries

SP-o	The index is hashed on S and P. A key requires constant values for both S and P.
SO-p	The index is hashed on S and O. A key requires constant values for both S and O.
PS-o	The index is hashed on P and S. A key requires constant values for both P and S.
PO-s	The index is hashed on P and O. A key requires constant values for P and O.
OP-s	The index is hashed on O and P. A key requires constant values for O and P.
OS-p	The index is hashed on O and S. A key requires constant values for both O and S.
PP-x	A special cache index. The index is hashed on two P values. A key requires two constant values for two predicates.

Table 2.2: Basic hashed indexes notations

evaluation in a distributed environment. As has been shown in Section 2.2, the core of a basic SPARQL query execution resides in the execution of its WHERE clause. As this clause is seen as a set of triple patterns (Definition 3.2), the execution of this set can be classified into two levels, the first is the execution on level of a single triple pattern, which is called *data access path*, and the second is on the level of joining data access paths' results, which is referred as *join evaluation*.
We consider first the execution on the conceptual level, as an introduction to detail the execution of both the data access path and join evaluation. Then both of the execution levels are seen with respect to a further classification that is related to the underlying storage and indexing structure. This classification goes as seen before in section 2.4, into: *key-value indexing*, and *graph-based indexing*.

2.6.1 The Bounding of Queries

As was introduced in Section 2.2, we refer to the query by its WHERE clause which is a set of triple patterns. This set is mapped into a graph and the query evaluation is the process of finding all the sub-graphs in the RDF graphs that match the query graph. The answer of some queries can be no more than one sub-graph which we called the bounded queries, while the unbounded queries may produce many disjoint sub-graphs. We present the properties of each type in the following:

- Bounded queries: the query graph has at least one constant vertex within its structure. Since that we know from Definition 2.1, that a vertex is guaranteed to be unique within the RDF graph, then if a query graph contains at least one constant in any of its vertices, and given that the query graph is connected (Definition 3.2), there must be no more than one connected sub-graph within the RDF graph that answers the query. Thus, such query execution is going to stick within a limited locality in the RDF graph, which is the location of the constant vertex and its neighbours.

- Unbounded queries: if the query graph contains no constants in any of its vertices, but has constants within its predicates (which stand for the edges' label within the graph structure), the number of sub-graphs that may match the query graph is unlimited; because, the predicates are not unique, and usually frequent within the data set or the RDF graph.

The bounding type of the query can be detected only by simple check of the query; however, it has a strong impact on how the query is handled and processed.

2.6.2 Conceptual Execution

On the conceptual level, we model the SPARQL query as a graph as was mathematically defined in section 2.2. A SPARQL query example with its graph representation is shown in Figure 2.2. Each triple pattern in the query is transferred into two vertices, the first models the subject while the second represents the object, with a directed edge from the subject to the object. The label of the edge represents the predicate of the modeled triple pattern. Some of the vertices are variables, and the variables which are shared between triple patterns share the same vertex in the query graph. This graph-based model is defined in Definition 3.2. The evaluation of this query graph is the process of finding all the sub-graphs in the RDF graph that match the query graph and substituting its variables. This operation is subdivided into *data access paths* and *join evaluation*.

2.6.3 Data Access Paths

The evaluation of a single triple pattern is conceptually the process of finding all the triples in the data set that are matching it. For a database physical layer, this level of triple processing is implemented in a dedicated scan operator. The efficiency of the scan process is directly related to the complexity of looking up the triples in the system using the available indexes structures. If the appropriate indexes are

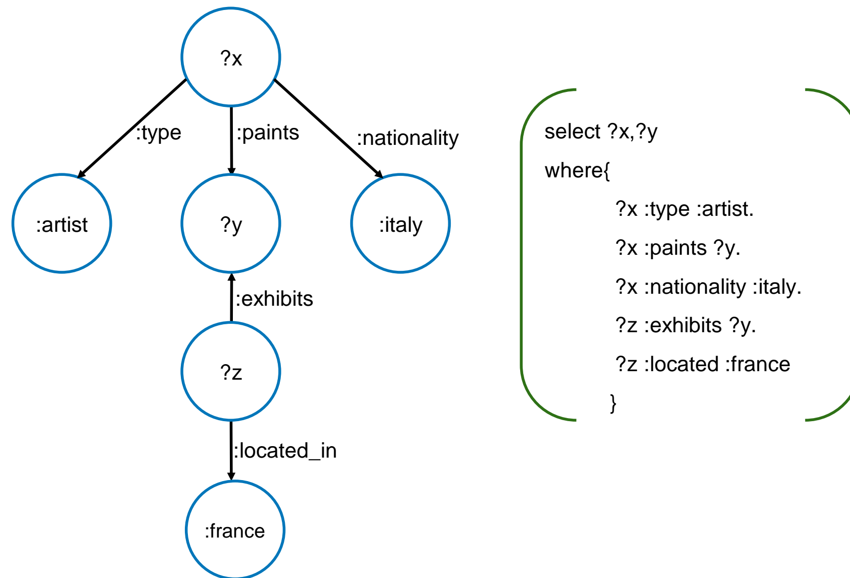

Figure 2.2: Query graph example

available, the scan operator complexity could be mapped to the complexity of a hash table, which requires in average constant time for a lookup operation [5]. We are going to review the process of data access path from two main system perspectives. The first is the key-value indexes, and the second is the graph exploration.

Key-Value Indexes

Given a triple pattern $t = \{S, P, ?O\}$, the fastest way for a key-value store to find all the triples in the data set that match t, is to use SPo index. The constants S and P would be used as the first and second keys, while all of the returned triples from the SPo index represent the direct answer to triple pattern t. More about indexes notations are mentioned in Section 2.5.
Systems like RDF-3X [56] and Hexastore [81] have the full flexibility in triple pattern evaluation, since they implement the six possible indexes types, and are thus able to directly evaluate any triple pattern independent of the triple pattern's variables counts or locations. However, if the system that is executing t, doesn't have the SPo index, but has the SOp index instead, it would only be able to use the constant S as a key. Extra filter operation is required to filter out the triples that do not have constant P as a predicate. The worst-case happens, when the system doesn't have

[5] Depending on the implementation, the worst-case lookup time could drop to linear

an index that uses S or P as keys, as this would require a full scan to the whole data set. To avoid this problem, most RDF key-value stores have at least three indexes such that we have always subject, object, or predicate used as a key in one of the indexes.

Graph Exploration

If the system has graph-based indexes like Trinity.RDF [85, 72], the evaluation of a triple pattern t would start by finding a vertex, either from a constant subject or constant object in t. This can be directly achieved by using a global hash-based index that returns a starting vertex. From this starting vertex, the system can check either the incoming in case of that the starting vertex is an object in t, or checks the outgoing adjacency list, in the case of the starting vertex is a subject in t. The evaluating system then outputs the answer of t by filtering only the triples that have P as a predicate (or edges' label). This filtering operation is also optimized by sorting (or hashing) the adjacency list on the predicates.
If neither the subject nor the object are constant in the triple pattern, the system needs to use a separate POS or PSO indexes to answer the triple pattern efficiently.

2.6.4 Join Evaluation

The execution of a single triple pattern takes place in the scan operator and feeds its result to a join operator. The join process is dependent on the storage and indexing structure available in the system.

Key-Value Stores

Given the SPARQL query[6] $q = \{t_1, t_2\}$, where t_1 and t_2 are having one common vertex. The query planner of a typical key-value store has two options to join t_1 and t_2. The first option is to use a scan operator on both t_1 and t_2, and look-up the matching triples using the available indexes, then join the results of the two scans by using sort-merge join. This approach is shown in Figure 2.3. The second option would be to use one scan operator on either t_1 or t_2, and then for each of the triples that are delivered by the scan operator, the join operator would look for matching triples using again one of the indexes, following the method of hash join. This approach is shown in Figure 2.4.
Assume for instance that $t_1 = (c_1, c_2, ?a_1)$, and $t_2 = (?a_1, c_3, c_4)$. The execution plan

[6]The query is defined as set of triple patterns as given later in definition 3.2

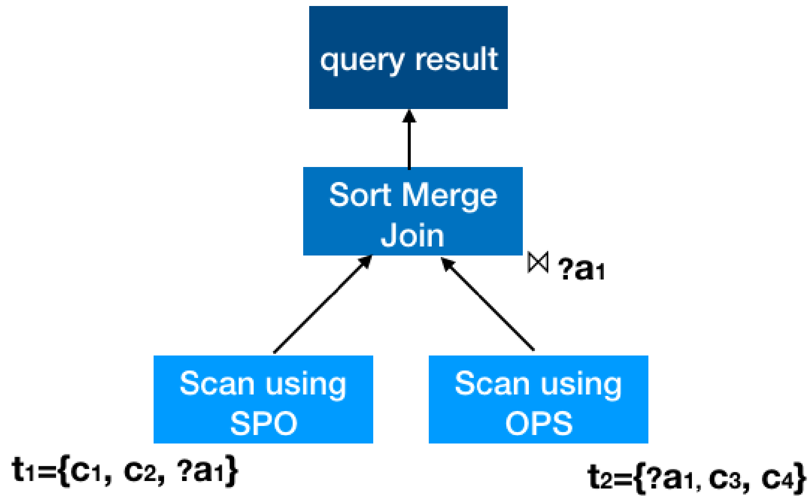

Figure 2.3: Query plan based on sort-merge join

may have a scan operator for t_1 that would use the SPO index, and another scan operator for t_2 that would use the OPS index. The triples resulted from both scans would be joined on a_1 using a sort-merge join operator. Now, assume further that the scan operator of t_1 produces only 4 triples, while the scan of t_2 produces 100 triples. Instead of the merge-join, it would be more efficient for the join operator to take the 4 triples of t_1 and use the SPO index to look directly for matching triples. The selectivity estimation and the availability of indexes in the system are the factors that play the biggest role in determining the more optimized choice in this regard and can save huge computations.

The join evaluation of q in the above example was straight forward since there are only two triple patterns, and thus there is one possible order of join. However, if q is composed of three triple patterns: $q = \{t_1, t_2, t_3\}$, then there are three possible orders of joining:
$(t_1 \bowtie t_2) \bowtie t_3$,
$(t_1 \bowtie t_3) \bowtie t_2$,
$(t_2 \bowtie t_3) \bowtie t_1$.
From the rich literature of the classical relational database, we know the following points: the number of join plans grows exponentially with the number of relations (the number of triple patterns in our context), the cost of each plan is related to the selectivity of each relation (the data access path size of each pattern), and the cost estimation depends on collecting statistics in the form of a histogram. Moreover, the error in cost estimation at some level of the join tree propagates exponentially to the

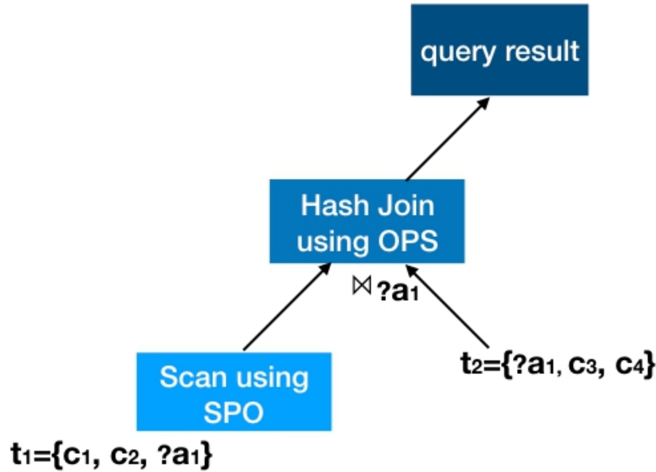

Figure 2.4: Query plan based on hash-index join

higher levels [40].

The problem becomes more challenging in SPARQL queries due to the high variation in selectivity between elements that compose the triple patterns [76], and the schema-free nature of RDF data.

To deal with the selectivity estimation problem, the authors of RDF-3X [56] suggested to have two kinds of statistics: *histograms* and join paths. The histograms are basically the count of occurrence of each predicate in the data set. However, since this way of selectivity estimation does not tell how two predicates count in the data set, RDF-3X calculates some frequent join paths and their counts in the data set. It should be clear that those frequent paths are found in the data set and not in any workload. For any query, the system tries to look first in its join paths to estimate the selectivity, then moves to the use of the generic histograms. Finding join paths in the RDF data set is also studied by [29, 50, 81], and even used to inspire the RDF data partitioning in [84].

Why too many indexes?

In spite of the storage overhead, RDF-3X [56] and Hexastore [81] proposed to use all possible six indexes in order to have the full flexibility and query execution efficiency. However, for the part where we would like to answer a single triple pattern (data access path), we need at most the following set of indexes: $\{SPo, OPs, POs, OSp\}$. This set makes it possible to answer any triple pattern where the number of variables

is at least one. However, when we need to perform join evaluation, we may also need PSo and SOp. consider for example the query $q = \{t_1, t_2\}$, where $t_1 = (?a_0, c_2, ?a_1)$, and $t_2 = (?a_1, c_3, ?a_2)$. Assume that the optimizer decided to have the following evaluation plan: evaluate both t_1 and t_2, then perform a sort-merge join on the results. The evaluation of t_1 should be carried out using the POs index in order to have the result sorted on the object. On the other hand, the evaluation of t_2 should be carried out using the PSo index in order to have the result sorted on the subject. This allows the merge join to take place directly without having to perform the sort step, which would be necessary if one of the indexes is not available in the system.
Having more indexes provides the benefit of fewer queries execution time at the cost of more storage space. We study and formalize this relation further in Chapter 4 and show how it fits in our space adaption model where we consider other things that require space like replication and cache.

Graph-based Stores

In a system with graph-based indexes, the query execution follows the graph exploration algorithms.
Recalling our previous query example $q = \{(c_1, c_2, ?a_1), (?a_1, c_3, c_4)\}$, the query execution would either begin from the vertex of c_1 or from the vertex of c_4. The graph exploration continues as it was explained in the data access path evaluation in section 2.6.3. Obviously, the selection of a good starting point would dramatically affect the query execution efficiency. Trinity.RDF [85] suggested the use of dynamic programming to solve such optimization problem. The authors also proposed a selectivity estimation technique by considering the correlation between pairs of triples.

2.7 Distributed Triples Store

A distributed database is a concept applied to any database that resides on multiple machines communicating by a network. Usually, this distribution should be transparent to the user who issues queries and expects unified, correct, complete answers. The reasons for having a database to be distributed are similar to the general reasons of having distributed systems:

- **Resources sharing**. The most important shared resource for a distributed database, is the storage space on all of its levels. For example, sharing the main memory between 10 working nodes is more scalable and feasible than increasing the main memory of a single node by a factor of 10.

- **Reliability**. Given the fact that any hardware is associated with a probability of failure, the ability of the system to keep a high availability ratio is related to the design of a backup system that can replace any failed part. This is reflected in a distributed database system by making replication of data. Each piece of data should reside on at least two physical machines or working nodes. However, these replications create another challenge to the system in terms of keeping the data consistency and employing these replications in queries processing.

- **Processing speedup**. The database management system needs processing power to answer users' queries. A distributed system has generally more processing power; however, the challenge is to design the system such that the queries execution speedup behaves linearly with respect to the number of processors.

The scale of current RDF triples is in the range of tens of billions, while some commercial RDF data sets have already reported going beyond 1 trillion triples [26]. This big scale clearly justifies the needs of having distributed triple stores to provide resources sharing and system reliability, while trying to achieve considerable processing speedup. We present next the main methods of storing the big RDF data in a distributed environment.

2.7.1 Distributed Storage and Indexing

This section focuses on the main distributed storage approaches, which can be grouped and classified into the following categories:

- systems which are built on Distributed File Systems;

- systems which are built on "NoSQL" key-value stores;

- systems which use a native centralized RDF store (Section 2.3.2) at each working node.

We briefly present each approach and discuss its performance as well as its challenging issues.

Relying on Distributed File Systems (DFS)

Distributed file systems are designed to support scalable and reliable storage of data across a cluster of working nodes. Systems like Hadoop [24] automatically take care

of replicating the data to provide fault tolerance, and directly support scalable and parallel query engine based on the Map-Reduce paradigm. However, the DFS stores the data in files, and thus provides very poor grained data access. To overcome this problem, some systems [39, 65, 87] vertically partition the triples into files upon their predicates. However, a linear scan is needed on every file in order to search for matching subject or object for a given predicate. Under this approach, some files that represent frequent predicate within the data set, would end up to be very big files which amplify the linear scan problem. HadoopRDF [39] suggested to detect such files and further split them. Another approach would be to perform the partitioning by subject or object, but it would end up facing the problem of having a high number of small files.

Relying on distributed key-value stores

Key-value stores provide efficient non-relational storage of data with fine-grained access suitable to the level of RDF indexing. Systems like HBase [25], Cassandra [23], SimpleDB [7], and DynamoDB [70] work in clustered distributed environment. The data model requires typically to define key-value indexes, which is specified for the RDF data by choosing one or two[7] of the triple elements as the index key, as explained in section 2.4. The access to the data is fined-grained to the level of the index key; however, the partitioning and assignment on the cluster is also performed on the level of index-key. This makes the system pay considerable communication cost to shuffle the data across the cluster, and perform the required join especially in queries that have many connected triple patterns.

Relying on federated triple stores

This approach generally divides the system into set of working nodes, where each node is responsible on its share of the RDF data in terms of indexing and query processing. The partitioning and assignment is carried out by a master node. The key point is that the partitioning is not done on the level of indexes as was the case in the previous distributed key-value stores, but on chunks of connected data, or by graph partitioning. For this purpose, METIS [45] is widely used. It contains a collection of graph partitioning tools that follow the min-cut algorithm, with various configuration parameters.

When the RDF data is considered as a graph, and is partitioned on this basis, the working nodes would receive a connected bunch of data. This gives each node a better

[7]in some rare cases the index key may contain the three triple elements as key

chance to execute SPARQL queries locally, while decreasing the communication cost of moving the intermediate triple pattern results across the cluster. We explain this problem and its proposed solutions in the next section.

2.7.2 Data Partitioning

For a typical distributed RDF triple store, the underlying distributed storage approach determines how the data is distributed. For a distributed file system and NoSQL key-value stores, the underlying system handles the task of the data partitioning, assigning, and synchronization. On the other hand, the federated triple stores need to take the responsibility of the data partitioning themselves. As was already introduced, a general federated triple store composed of n working nodes, such that each node hosts an independent triple store that manages its own share of data and handles the queries execution. The system needs to generate at least n partitions out of the global RDF graph. In spite of that several methods exists to handle RDF data partitioning, two main directions got more researchers attention which are: Graph-based partitioning and Hash-based partitioning. We review both directions in the next subsections.

2.7.3 Graph-based Partitioning

This direction of data partitioning makes use of the natural mapping of an RDF data set to a graph by using one of the general purpose graph partitioning algorithms. The behaviour of these algorithms is well known and highly researched. In this context, the widely used algorithms for the purpose of RDF graph partitioning are the min-cut algorithms. Such algorithm divides a graph into n partitions aiming to have a minimum number of edges between the resulted partitions. The basic hypothesis that justifies why such division could work for RDF graphs comes from the idea that the most related parts of an RDF graph should be also highly connected with a bigger number of edges. Thus, the partitioning objective when using a min-cut algorithm is to keep the most related parts of the data in a single partition. Moreover, since the query execution has been mapped to a subgraph matching problem, the number of queries that require a flow from one partition to another is expected to be reduced when there is a minimum number of edges between the partitions, under the assumptions of uniform data access by the workload, and the uniform partitions size in term of the number of vertices. In this context, the required min-cut algorithm is expected to produce equal in size partitions that have a minimum number of edges in between. Since such algorithm is known to be NP-complete [44],

an approximation algorithm is used that tries to achieve the best approximation for both of the optimization objectives. METIS [45] is a collection of algorithms and software tools that are known to give good results in this regard. Huang et al. [38] were the first known to use METIS for this purpose. However, METIS is also used by multiple works for the same purpose [37, 86, 28]. According to the authors of METIS it requires $O(|E| + |V| + k \cdot log(k))$, where k is the number of partitions. However, the practical performance of METIS is highly dependent on parameters like the required balance of the partitions' sizes, the required accuracy of the min-cut between partitions, and the edges density distribution of the graph. Setting strict conditions would result in slow performance as noted by [80, 31].

Border Region

Applying the METIS partitioner on the example RDF graph in Figure 2.1 and setting the required partitions number to 4 would result in the output shown in Figure 2.5. Consider the queries shown in Listing 2.4 below:

Listing 2.4: Sample Queries

```
q1: {:rodin ?p ?o}
q2: {:rodin ?p ?o . ?o :exhibted ?s}
q3: {?s :exhibted ?o}
q4: {?s :exhibted ?o . ?o :located ?s}
```

The first query q_1 is a single pattern query and only Partition 3 processes it to output the triple: *:rodin :create :thethinker*. However, q_2 contains another pattern, and its execution requires some data that exist in both Partition 3 and Partition 4. If we assume that each partition is assigned to one machine, then processing q_2 requires data synchronization from two machines which are connected over a network, and communication cost needs to be considered when evaluating the query performance. Taking into consideration the big size of the RDF graph and diversity in its relations and connections, the intermediate results of a SPARQL query can be considerably big. This motivates the use of the min-cut algorithm to partition the RDF graph in the first place. However, since METIS represents an approximation algorithm which is required to produce balanced partitions in term of their sizes, we still have a *border region* at each of the output partitions. This border region contains vertices that have edges coming or going to other partitions. We define this border region in the context of this thesis in Chapter 5, Definition 5.2. This border region exists in Figure 2.5 for the four partitions and composed of one vertex. Partition 1: {:guernica}, Partition 2: {:reinasofia}, Partition 3: {:thethinker}, and Partition 4: {:museerodin}. Every

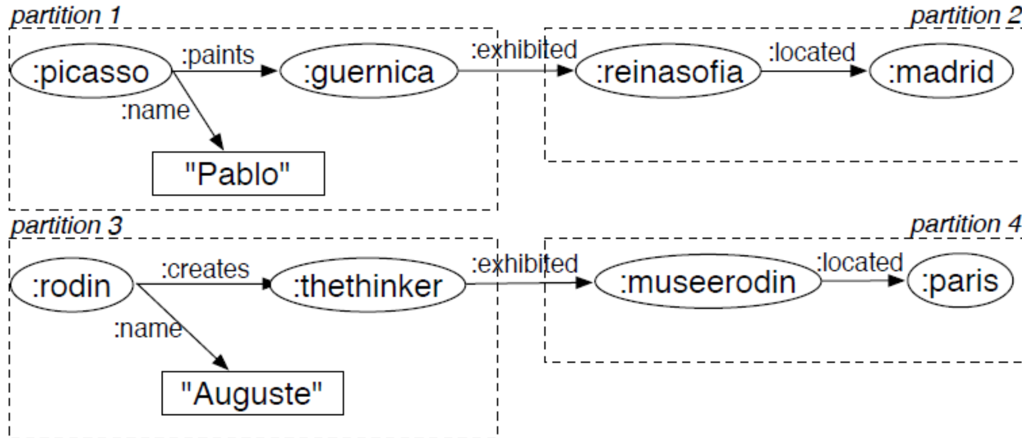

Figure 2.5: METIS output example by [42]

time a query like q_2 touches a border region, we should expect the communication cost of moving intermediate results across the network.

Another point that makes this border region require special attention is that the min-cut algorithm assumes uniform data access by the queries. This means each vertex and edge in the data set have equal probability to contribute in the queries. However, this is not true about a typical RDF workload, where some parts of the data are more frequently accessed by the workload. This is more explained in Section 3.3. Thus, if some vertex that happens to be part of the border region is also part of the most frequent workload, high communication costs are going to be paid in term of the query execution performance.

Border Region Solutions

To overcome the problem of the METIS border region, several solution approaches are proposed. First approach proposed by Huang[38] is to replicate the border region to all the concerned partitions. A similar approach is also followed by [55, 47, 37]. Figure 2.6 shows how can this be applied on the METIS output of Figure 2.5. This replication approach is called 1-hop guarantee since it is now assured that any query can be locally answered if it is requiring vertices which are located no more than one hop from the border. We refer to this hop measure as the distance from the border as given later in Chapter 5. Considering the 1-hop guarantee partitioning, q_2 can now be answered locally using only Partition 3. A 2-hop guarantee increases the

replication's depth one step further as seen in Figure 2.7 allowing the local answering of longer queries with no need for network communication.

The obvious drawback of the above solution is its storage space requirement which is showing exponential growth with the increasing depth or number of hops. The second drawback of fixed replication distance, is the lack of a systematic methodology to determine the appropriate distance value that the system should consider for replication. To overcome these issues WARP [37] suggested looking to the workload. All the initial steps of partitioning and assignment followed by Huang are still followed by WARP. However, WARP proposed to perform limited fixed replication from the border region as Huang suggested but only to a *small* number of hops H_r, then use the workload to perform further replication for more important triples. The process of workload analysis is based on Partout [26]. It sees the workload as a set of given queries that represent a sample of historical queries. Each query is a set of triple patterns, each triple pattern is a set of exactly three items, and each item can be a constant or a variable. The operation of workload analysis starts with a process called *normalization*. It counts the number of occurrences per item within all the triple patterns. Then, a certain threshold γ is assumed to exist such that an item with a number of occurrences that is fewer than γ is considered non-frequent. All non-frequent items and all the variables within the triple patterns are now replaced with a single variable α. The process creates a single set of normalized triple patterns p_n, where each pattern is associated with a frequency value. The WARP uses the information provided by this set to detect the most important triples that are located at the partitions border, and replicates them. Those most important triple are determined when they match one or more patterns in p_n such that the frequencies summation of the matching patterns is above a certain threshold C_{rep}.

As WARP provided a good methodology of solving the border problem depending on the workload, it didn't provide any clear approach to detect or calculate the two thresholds γ and C_{rep}, beside the initial static Huang replication distance H_r, which is described by WARP as small distance. Setting the values of these thresholds and constants represents open challenges to the applicability of the approach. Another issue with the WARP approach is that it presents a binary importance function to the triples, such that a triple that has an importance greater than C_{rep} is considered hot; however, all hot triples are treated with the same level of importance. This doesn't simulate the practical workload trend which has continuous behaviour. Thus, the binary importance decision would give a weak performance, and goes weaker in case the workload is varying, the storage space available for replication is limited, and

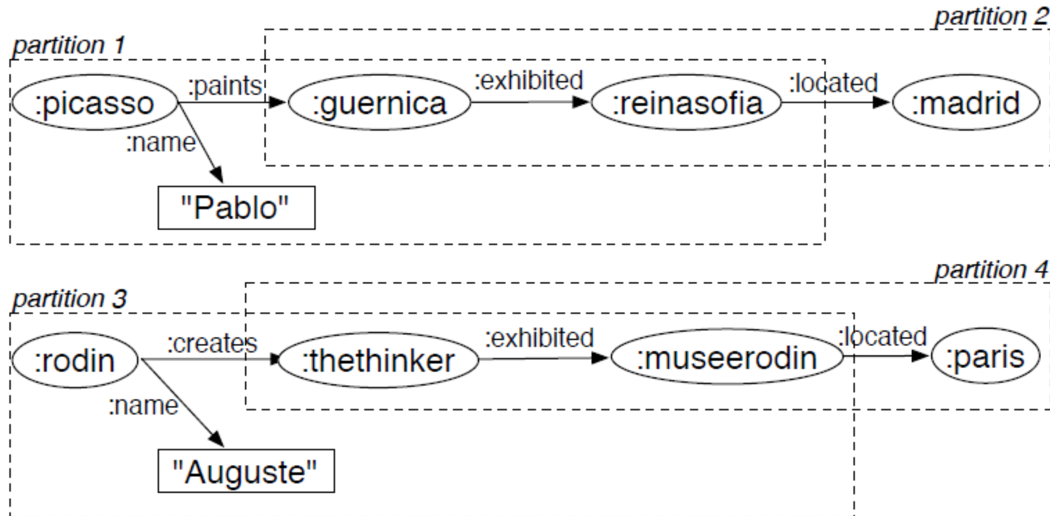

Figure 2.6: 1-hop guarantee example by [42]

there are other storage requirements by the system. We will show in Chapter 6 that our universal adaption system provides more advanced solutions to all of these issues. Our important function to the border region is continuous and is derived within the same domain and relativity of other storage requirements such as indexes. The system adapts itself with the workload and has no fixed threshold that needs to be set at run time.

2.7.4 Hash-based Partitioning

The second direction of RDF data partitioning is by applying a hash function to decide to which partition the triple should be assigned. The hash is typically applied to the subject of each triple in AdPart[31]. However, some systems like TriAD [28] perform another hash assignment on the object. There are three basic advantages of hash-based partitioning which are:

1. Fast and lightweight process since it requires only one round of hashing that is linear with the number of triples. In term of the graph measures, this operation requires $O(|E|)$ steps.

2. The locality of any triple can be easily determined by direct applying of the hash function on the subject.

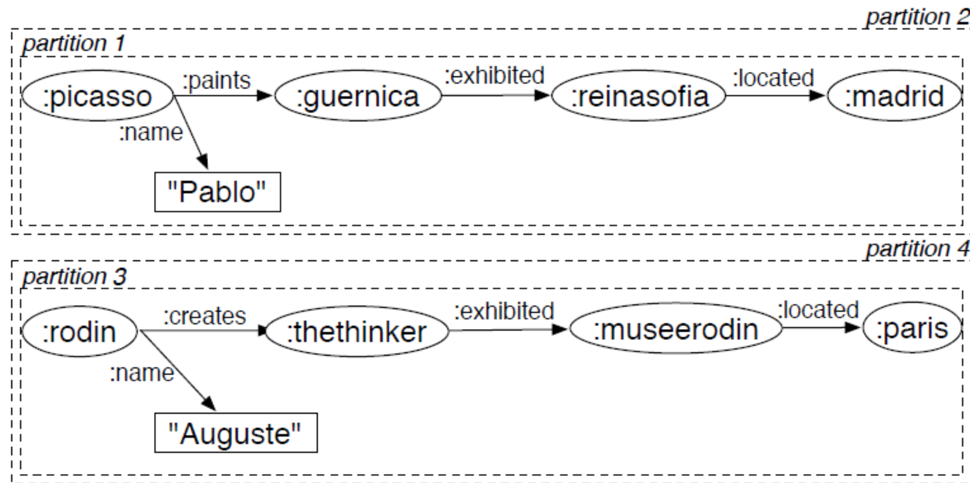

Figure 2.7: 2-hop guarantee example by [42]

3. Achieving high parallelization of some queries is possible. These queries are mostly the short in length unbounded queries.

However, the main disadvantage of hash-based partitioning is that it may cause high communication costs for most types of queries other than those mentioned by point 3 above.

A possible hash-based partitioning of the RDF graph in Figure 2.1 is shown in Figure 2.8. Any star query that is bounded on the subject can be evaluated by a single machine; however, there is no distributed parallelization in this case. An example of such a query is q_1 given by Listing 2.4 which will be answered by the machine hosting Partition 2. However, q_2 needs both Partition 2 and Partition 4. Generally, the more hops a query performs, the more the probability of hopping to another node (machine), because of the hash-based assignment of subjects. If the query is not bounded, which means that it has no constant at any subject or object, then there is a better chance for parallelization. Consider for example q_3 in Listing 2.4, the query is only bounded by the predicate *:exhibited*. Since we are performing a hash on the subject, we may assume the subjects who have the predicate *:exhibited* are fairly distributed around the partitions (working nodes). Thus, the system can evaluate this query by all nodes in parallel. However, the results of this query could have strong dependency on the selectivity of the predicate *:exhibited* in the

big data set. The results need to be unified at some node and would still need the communication cost. If q_3 is extended to q_4, which is still unbounded, but requires a second round of execution that performs the exchange of the intermediate results between Partition 3 and Partition 4 then joins them. The longer the query, the more probability of the indeterminate results exchange; and the more the selectivity of the predicates the bigger the size of the moved data.

Applying the hash of the subject of any triple pattern would directly locate where the triples that match this pattern or in which partition they are. For instance, if the triple pattern of q_1 is been processed by the node hosting Partition 1, it can directly know that only Partition 2 has the related triples, and thus may directly route it to the designated node. Such a strategy has been employed by AdPart [31] to route the intermediate results only to the hosting node. However, the METIS-based partitioning faces the problem of intermediate results synchronization only when it touches the border area. Since this area is cut with edges that go to(or come from) other partitions, and each node can know exactly to which node each edge is connected (by having 1-hop guarantee), it can easily route the intermediate result only to the designated node. Moreover, any destination node can determine if any subject exists within its partition by one call to the SPo index, which is a hash-based index on the subject of each triple. This operation would cost similar or comparable time with respect to applying the hash function on the subject at the source node.

2.7.5 Data Partitioning Summary and Conclusion

The applicability of each partitioning strategy is highly related to the type of the data graph and the type of queries that are expected to be evaluated on it.

The hash partitioning is generally faster, easier, and more deterministic. On the other hand it, causes higher communication costs for longer and more complex queries. The graph partitioning is more sophisticated and thus it is less deterministic in terms of the size-balance between resulted partitions and the density of edges that cross the cut or what we called the border region. However, in terms of queries evaluation, it requires communication costs only when the evaluation touches the border region. Thus, graph partitioning can more efficiently process longer and more complex queries, if the border region is small or limited compared to the total data graph.

In both approached, the problem of communication costs can be overcome with replications; however, for graph partitioning it is easier to handle, manage, and measure since it is only related and limited to the border region. In hash partitioning,

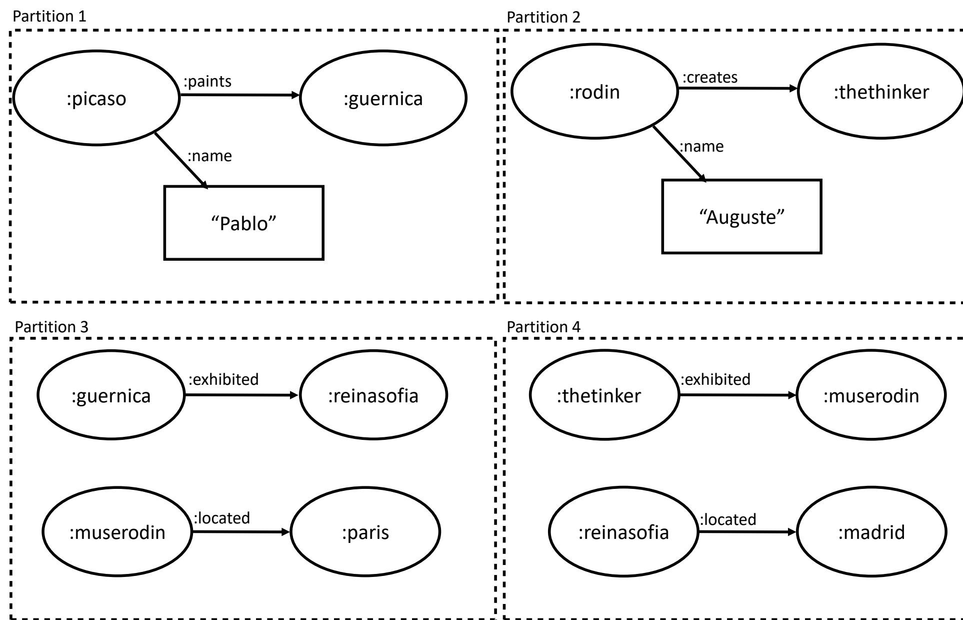

Figure 2.8: The hash-based partitioning of graph in Figure 2.1

it is more difficult and challenging to decide which data to replicate.

Another point of comparison is the distributed speedup or the ability of the system to process a single query by multiple nodes. Since the graph-based partitioning aims to increase the chance of local query execution, it is more difficult to utilize all of the system processing resources during the evaluation of a single query. The hash partitioning has more balanced data distribution and thus we may expect better load balancing. The benefit of speedup may overtake the increasing communication cost, in case of short unbounded queries. However, when we consider the general queries arrival trend in a practical server [78], the system has many queries in its queue, and thus utilizing the system resources with multiple queries gives better throughput than trying utilizing the processing resources with a single query bearing the communication cost. In this context, [41] showed different cases where the queries perform better on hash-based partitioning compared to graph based partitioning. However, the authors used single queries (total 20), one at a time, and measured their performance.

We will show in Chapter 7, the impact of each of the above points, and the practical effect of each partitioning approach on the query performance, system throughput, and the adaptation of the system.

2.7.6 Discussion

From the different approaches that have been presented in this section, we can identify that when the RDF management system is relying on existing general technologies, it inherits their optimized scalable performance and the simplification of implementation, but lacks enough flexibility to tune the RDF system towards its specific and customized needs. For example, a system may gain benefits by relying on distributed file systems but looses the fine-grained data access. On the same regard, using a general key-value store to store RDF data, makes the system showing high performance when querying a record by its key which is called data access path in our used terminology. However, the RDF system has lost the control of data partitioning and replication which is handled by the underlying key-value store. Giving that, except for short SPARQL queries, the system may require several indexes access to evaluate a single query, the high performance of the data access path would be ruled out by the cost required to move the intermediate results across the distributed nodes and join them.
From the given analysis of used RDF management approaches, a native RDF storing system provides the ability to handle the special challenges emerging by storing RDF data in a distributed system, optimize the performance of SPARQL query specifically, and adapt the system with workload and storage capacity. This provides the basis we used to construct our decision of using a native RDF system where we have a cluster of federated triple stores or working nodes, such that each working node has its own share of the data that it should store, manage and query.

2.8 Most-Related Work

In this chapter, we have presented so far the techniques, and the approaches used to store, manage and query RDF data, which are related to our work in this thesis. In this section, we review and summarize the works that are closely related to the main contribution of our thesis, although some of them were already mentioned and discussed within the sections of this chapter.

RDF-3X[56] is one of the first native centralized systems that was specifically designed to store and manage RDF triples. RDF-3X uses an excessive index scheme by implementing all the 6 possible indexes arrangements. To decrease the storage overhead, RDF-3X uses a dictionary, where each textual element in the RDF data set is mapped to a small integer code. The query needs to be translated using the dictionary before the execution, while its result needs also to be translated using a

dictionary to print the final textual results. The six-indexes scheme is used before RDF-3X by Hexastore [81]. However, besides the six indexes, RDF-3X uses aggregate indexes where it does not store the actual triples but the number of its occurrence for a certain key. These aggregate indexes are used for selectivity estimation and the generation of an optimal query evaluation plan. RDF-3X is used as the base of multiple federated distributed triple stores such as Huang [38], WARP [37], and Partout [26]. We use RDF-3X as the baseline of the part that provides the hard disk support to UniAdapt.

The **H-RDF-3X** system by **Huang et al.** [38] was the first distributed system that used a grid of centralized systems, such that each node is hosting an RDF-3X triple store. At each node, there is an add-on that handles the partitioning and distributed query processing. The data is partitioned using METIS such that each node receives a partition. To reduce the communication cost, H-RDF-3X forces k-hops replication performed equally on all the vertices in the border region that are located within k-hops from the border. Any query that is shorter than k is guaranteed to be locally executed. However, for longer queries, they may require joining intermediate results from different partitions. H-RDF-3X performs this using MapReduce joins over Hadoop. Unfortunately, the storage overhead of the replication increases exponentially with k, and H-RDF-3X did not provide any systematic method to practically calculate the value of k. Given the RDF workload behaviour (Section 3.1), it could leave most of the replicated data unused despite its high communication cost.

Partout [26] was the first system that implements workload awareness on the level of partitioning. It uses the minterms principle to horizontally partition the data set into fragments inspired by the classical approaches of partitioning relational tables [16, 57, 48]. The system tries to assign fragments to partitions such that the most related fragments are assigned to the same partition. The fragments are more related when they contain triples that appear more frequently in the workload. The main problem of this partitioning is that its result is highly affected by the quality of the used workload in terms of its queries' frequency of appearance and whether the prospected queries are still going to follow the same trend of the used workload. As a result, it could end up with small fragments representing the workload and a big single fragment containing everything else.

WARP [37] proposed to use a combination of Partout and H-RDF-3X aiming to overcome their problems and emphasis their benefits. Initially, the system is partitioned and replicated using H-RDF-3X approach and with a *small* value of k.

Then, the workload is used to decide on making more replications from the border by recognizing the most important triples. The workload analysis is basically similar to that of Partout. The workload is normalized [8] and its items are aggregated on the frequency. An item is considered frequent when its frequency exceeds a *certain* threshold. Unfortunately, there is no specific method given in [37] to calculate this threshold. Moreover, WARP uses initial static replication of k-hops, but lack the specification of determining it. Another issue with WARP is that it treats the frequent items equally even if they are very various.

Peng [60] proposed a partitioning and assignment approach inspired by Partout [26]. It detects frequent patterns in the workload and uses them to generate two types of fragments. The first is vertical fragments which are basically similar to the Partout horizontal fragments. The objective of this vertical fragmentation is to decrease communication costs during queries evaluation. The other type of fragmentation is called by the authors: horizontal fragmentation. This fragmentation process tries to distribute data matching frequent patterns to many fragments such that the working nodes in the system may process them in parallel when a related query is evaluated. As a result, the first fragmenting operation aims to increase the system throughput, while the second aims to increase the system distributed speedup. While the workload analysis process is similar to Partout and WARP, Peng has a continuous benefit function to sort the fragments by their importance. However, the approach amplifies Partout's problem of workload dependability. If the system has very limited frequent patterns or the queries happen to come in different trends of the frequent patterns, the system may behave very badly.

AdPart [31] is an in-memory distributed triple store. It aggressively partitions the data set by hashing the subject of each triple. As this is known to produce high communication costs, AdPart proposes two solutions. The first solution is by updating the dynamic programming algorithm [56, 53, 85, 28] that is used to find the optimal query execution plan, to include the cost of communication. The objective of the algorithm becomes to find the optimal plan to reduce both the join cost and the communication cost. However, this algorithm depends on the accuracy of the cost estimation which is already a challenging issue regarding calculating the optimal join plan in a centralized system like RDF-3X [56]. The second solution to the communication cost problem is by adding workload-driven replications. As the basic idea of workload analysis still similar to Partout [26] in term of frequent patterns, normalization, and global items graph, AdPart does not assume the existing

[8] More details about the normalization process is given in Section 2.7.3

of any workload at the system startup, but instead collects, builds the workload, and adapts its replications with time dynamically. Similar to the previous systems Partout and WARP, AdPart requires the hard setting of a frequency threshold that is used to differentiate between frequent and non-frequent items, making it a non fully automated system.

We propose **UniAdapt** as an in-memory distributed triple store. It increases the level of system adaption to include both the workload and storage space. The system looks to the workload and adapts the structures used to employ its storage space. Initially, the system starts with METIS base partitioning, which provides a solid ground in case the workload comes in lower quality and its trends were not well recognizable. UniAdapt proposes a cost model to estimate the relative benefits of replications as well as the local indexes and caches. The benefit functions are continuous and on a single domain which enables the system to always fill the storage space with the best-known employment option. As a contrast to other systems, UniAdapt does not need fixed settings or thresholds, and shows better immunity to bad quality of the workload due to its unique layering of workload rules. It has three types of rules: specific, general, and generalized. Each rule is factorized with a ratio of impact that goes down when the rule is less effective. The whole workload adaption is working within the area of space adaption. Thus, when the storage space becomes abundant, the workload adaption constraints are automatically relaxed, and this is translated with more replication, more indexes, and more caches.

Table 2.3 states the abstract, specifications, main advantages, and drawback of related RDF triple stores, most of them employ workload for some level of adaption.

2.9 Summary

We presented in this chapter, the principles of Resource Description Framework (RDF) as a complete and standalone data model that has its specification, vocabulary, and serialization format. The RDF model is directly mapped into a graph, and thus can be queried in a graph-based querying language like SPARQL. The performance of query processing is highly related to how well the triples-data are structured and indexed. In this context, some systems proposed to use an extensive number of indexes to reach enough flexibility in query evaluation aiming to enhance the performance. However, this requires a lot of storage space.

The RDF data model is used for web-scale data, which forces the storing system to be designed on big-data principles. The move towards a distributed RDF storage system is seen as vital in this regard. However, this imposes new challenges to keep

	Partitioning	M.Memory	Adaption	Advantages	Disadvantages
Huang	METIS	No	No	Lower communication cost	High storage overhead
WARP	METIS	No	Replication with workload	Decreased Storage overhead	Fixed thresholds, poor workload adaption
Partout	Horizontal	No	Partitioning with workload	Presented the base of workload analysis	Fixed Thresholds, requires high workload quality
Peng	Horizontal, vertical	No	Partitioning with workload	Tries to Increase throughput and parallel speedup	Fixed Thresholds, requires high workload quality
AdPart	Hash	Yes (no H.D.)	Replication with workload	Fast partitioning, decreased communication cost and high parallelization in short unbounded queries	High communication cost in long queries and low workload quality, fixed threshold needed.
UniAdapt	METIS	Yes+H.D.	Replication+ Indexes+ processing resources, with workload and space	Flexible, continuous, and universal adaption, no fixed thresholds, high throughput, lower communication cost	Slower startup, may show limited distributed speedup on the scale of single query.

Table 2.3: The most related systems which employ workload adaption

the system scalable and optimized in terms of the storage space, which should in this case, not only host many indexes but also replications from the distributed partitioned data, keeping in mind that these replications on their turn need to be well indexed.

The methods to manage the RDF graph partitioning were considered by many research works. However, two important directions are graph partitioning and hash partitioning. While both have their points of strength and weakness, some works

suggested to look for the workload and have workload-aware partitioning. Since this can create non-stable behavior in case the workload is not good enough, other works suggested having workload-aware replication to support the static partitioning (graph or hashed). We identified the storage space as the common constraints that should tune the workload adaption for replication and indexes. In this thesis, we will propose and describe in the upcoming chapters a universal adaption concept where the indexes, replication, and join cache are competing for the limited storage space aiming to increase the system performance.

Chapter 3

Workload Analysis

In this chapter, we formulate the adaption system and its related cost model. Initially, we define our cost model that is used to enable the system's universal adaption. We detail the role and effect of the workload and how we structure and analyse a collection of workload queries in order to estimate the resources access rates which represents the moving heart of the cost model triangle. This cost model is used to adapt the storage space in terms of the: indexes, replication, and join cache. We provide for each option the necessary formulas of cost, benefit, and access rate. For a summary and description of all the mathematical symbols used in this chapter please refer to Appendix B.2.

Contents

3.1 Why Adaption? **46**

3.2 Universal Adaption **47**

3.3 The Role of the Workload **51**

3.4 Workload Rules **56**

3.5 Heat Queries **59**

3.6 Heat Query Specific Rule **65**

3.7 Summary **66**

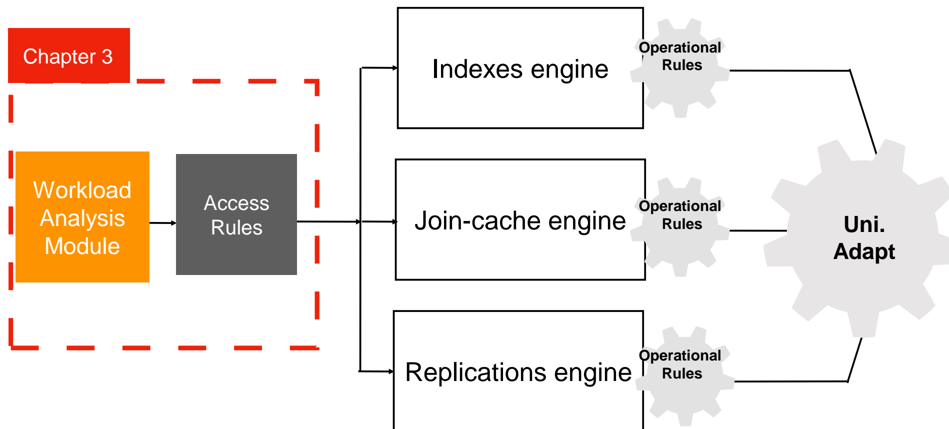

Figure 3.1: Chapter's scope

3.1 Why Adaption?

Database tuning is already a well-known concept in database management systems; it refers to the directive change in system parameters towards better optimization of available system resources. The effect of applying such tuning has a deeper impact on RDF management systems, where the indexes are very vital, and the storage space is very precious. Consider for a case example, the SPARQL online service of *dbpedia.org*[1]. The service is hosted on two CentOS 6 virtual machines. Each one has 8 Intel Xeon 2.30 GHz cores, SSD of 200 GB, and 64 GB main memory [78]. Assuming we run two federated RDF stores on each working node, and each node receives its partition of the RDF data set using a graph partitioning tool like METIS [45]. The system needs further to decide at each node about the replication's size and type, the number and type of indexes, the main memory and/or hard disk allocation, and the number of threads to be used for each query, given that each node has 8 hardware threads. According to an analytic study of a big of real users' queries log [12], about 80% of the queries that targeted a certain data set which is DBpedia were short queries. Moreover, the access rates of the given online service were suddenly doubled within a period of three months. Given only these two measurements, the

[1]Available at: http://dbpedia.org/sparql/

system may perform important adaption steps to substantially enhance its total execution time considering the following points:

- The ratio of 80% short queries means that the system does not need to have replication at the border of the partitions (needed to support long queries as shown in Section 5.1), unless it is able to recognize the size and locality of the 20% percent longer queries.

- The high number of arriving queries could mean that there are waiting queries in the queue most of the time. This means that the system should focus on having better throughput (i.e. executing more queries) rather than having high parallelization rate per query. By this, the system avoids the threads synchronization cost[2] which typically increases with the number of used threads per query in single working node. Moreover, the distributed nodes should try to locally execute each query and avoid any network communication cost (See details in Section 8.2).

- Given that the system does not need border replication for most queries, the storage space can still be employed by replications to enhance the load balance between working nodes on the cluster scale, and reflect this on the number of waiting queries within the queue of each working node.

From the above example, we see the system's ability to perform adaption steps to increase the performance, by the knowledge of only two or three measurements which are relatively easy to measure and maintain. However, we can also notice an overlap between the storage optimization decisions with respect to the replications, indexes, and the queries' arrival rate. This is clearly motivating the needs of universal adaption decisions that considers the overlapping measures.

3.2 Universal Adaption

In the previous section, we have shown the importance of optimizing the system's resources which can be classified into space resources and processing resources. Our adaptable system aims to make its resources adaptable with its knowledge of the environmental parameters that affect the performance. These environment parameters include workload, queries arriving trend, and data-sets types. We denote these parameters as *adaption subjects*. The storage space is divided into a set of space resources, such that each *resource* is the smallest unit of storage that the system

[2]Details on the threading cost are given in Chapter8

considers for optimization. Each resource can be filled with a unit of data equal to its size which we refer to as the *consumer*. The storage resource may maintain the unit of data in one of multiple indexes where each index represents a potential *option*.
Each resource can be assigned to one consumer and one option which forms the adaption triangle shown in Figure 3.2. As different triangles can contain the same resource and may deliver different benefits to the system, the optimizer may select the most optimal triangle for each resource. In this context, we can now set our **optimization problem** as the following:
Given a set of resources, a set of consumers, a set of employment options, and a current knowledge of adaption subjects, what is the best assignments of consumers to resources and in which options such that we gain the best query execution time?

In order for the system to answer the above optimization problem, it needs to have a cost model that can anticipate the benefits of the potential triangles associated with each resource, and this is what the adaption subjects are used for. The system builds its knowledge about these subjects and uses it to anticipate the benefit of employing the system's resources with different choices. For space resources, these choices are: the unit of data and the chosen type of index. The selected unit of data should exist in another source index prior to the optimization process, and it must be different in place(e.g. memory, hard disk, or remote) or type from the destination index. This difference generates a performance benefit to the system which the optimizer considers when deciding which unit of data to assign for each available storage space resource.
From the above context, we can describe the space-adaptable system as the system that is able of making optimized decisions about how to employ each of its storage units in order to achieve the best query execution performance within the current knowledge of the workload. At the same time, these accumulative decisions make the whole layer of indexes dynamically changeable in size, and the different local indexes are competing for space aiming to maintain more data. Moreover, looking from a higher level, we see the distributed system able to dynamically set its replications and local indexes, and the triples are moving around the nodes in the direction of achieving lower queries execution times.

In order for the system to have the required level of universal adaption, it needs to have a unified cost model considering all of the data sources and storage options. The cost model should set both benefit and cost values to the proposed decision options, while keeping in mind that these cost and benefit values need to be relative

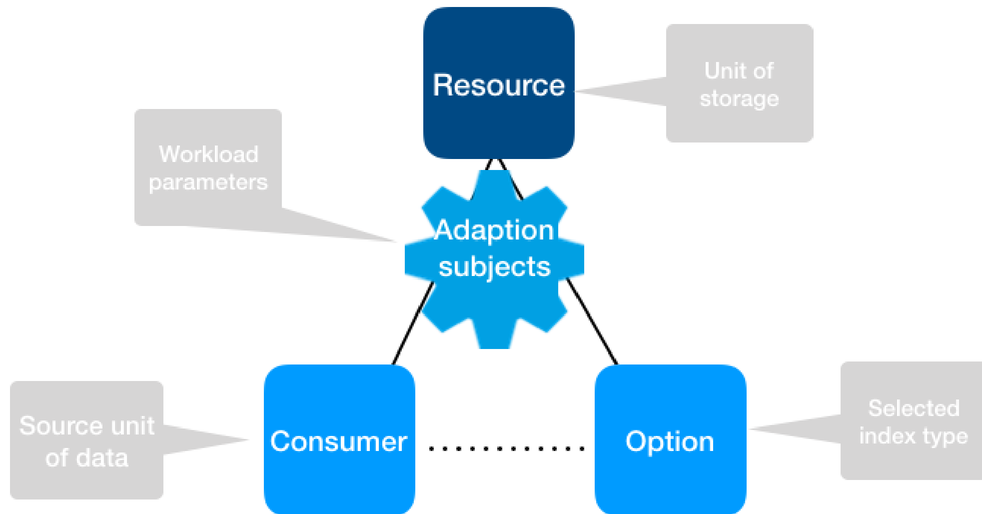

Figure 3.2: Components of the adaption model

and on single measurement scale, in order to make them comparable on a single selection line.
The space-adaptable method can be generalized to the adaption of the processing resources in the system keeping in mind that it is related to the queries arriving trends more than the workload knowledge.

3.2.1 The Cost Model

In order to achieve the universal adaption objective that has been described in the previous section, we aim to have a generalized cost model, in which we keep its input on the level of a single resource unit r. Please refer to Appendix B.2 for clarification of any used mathematical symbol.

We assume that each resource unit can be consumed by any of c consumer units. As it was mentioned earlier, there are multiple options $op(r, c)$ to utilize resource unit r with the selected consumer c. We refer to these options here as a function $op(r, c)$ which returns for each resource and consumer a set of available unique options' identifiers. Each option $op_i \in op(r, c)$ is going to deliver a benefit to the performance that we denote $\eta(op_i)$, and this is more precisely described as the ratio of the maximum execution time that could be saved when the system uses the option op_i, to the execution time under the case of option op_i is not available to the system. However, the effective system benefit of having option op_i is related to the system's needs to

use the resource r, which we denote as the resource's access, and that is in turn mapped to the probability of this access ρ. The general benefit formula of exploiting a certain resource unit with a certain option can be defined in the following:

$$benefit(op_i) = \eta(op_i) \cdot \rho(op_i) \quad (3.1)$$

$$op_i \in op(r, c)$$

Besides a unified and generalized benefit formula, the system needs to find a generalized cost of each option within each resource unit. A suitable and measurable value is the ratio of resource consumption needed to employ this option.

In order to apply this model to the storage space, we need to have the following three points:

- define the storage employment options per single storage unit;
- derive a method of evaluating the benefit of each option, given that the cost of each option is easily measured by the space size needed for each option with respect to the total available size; and
- derive a method to evaluate the probability of accessing of each option depending on the analysis of adaption subjects, as we explain in the next subsection.

3.2.2 The Resources' Access Rate

According to the cost model of the previous section, the benefit of utilizing some resource r with a certain consumer c is not used in the optimization process until it is related to the extent of its future usage. We derive this future usage which is stated in Formula 3.1 from the system previous knowledge of the performance parameters which we have previously called *adaption subjects*. Those subjects are related to the workload and the queries arriving trends. The system collects the history of those subjects, analyses them towards deriving a resource's access value. That access value represents the system's usage to that resource when employed by certain data that is structured in certain option.
The workload analysis process aims to derive resources' access value. It can be classified into three categories:

- General analysis, by gathering general statistical metrics measured over the entire collected history without specifying certain data parts. This includes for example the average query's length, the average query's shape, and the average query arriving rate per time unit. That kind of analysis simulates the

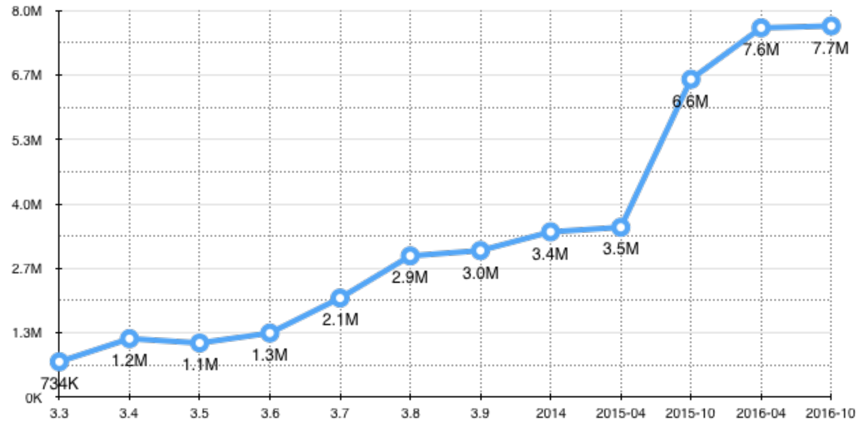

Figure 3.3: Average number of hits per day versus the DBpedia version, as appeared on [78]

hard observations used to make the fixed assumptions about the workload by non-adaptable systems.

- Specific analysis, which targets specific data parts that play the role of a consumer c in Formula 3.1. An example is the count of (frequency of) specific data vertex in the workload.

- Generalized analysis, which is originally specific analysis that have been generalized to give expectations about other parts of the data or other consumers. An example is when generalizing a specific vertex's frequency to other vertices that have an edge with the same predicate. The motivation behind having this generalization approach is to make the system able to work with a smaller amount of collected adaption subjects. However, we treat such generalized analysis with more caution by measuring their effectiveness to decrease the impact of bad generalization.

3.3 The Role of the Workload

The ultimate objective of the RDF triple store is to run users' queries. The metric to measure the system performance is related to the query response time, or to queries-set total response time. In this context, we refer to the workload as a set of queries at a certain time period within the system's life span. If the workload's time is in the past, the query set is considered a history workload. Otherwise, the future workload is referred to as set of queries that the system is expected to receive at some points of

time in the future. The expectation is recognized by a numerical probability value. The system's adaption with the workload is the steps that the system takes in order to make use of its history workload aiming to increase the performance of future workload. In order to give a precise definition of the workload, we state first the definitions of the basic concepts that are used within this thesis. We have defined the RDF graph in Chapter 2, Definition 2.1.
The basic element of a SPARQL query is the triple pattern which we define in the following:

Definition 3.1 (Triple Pattern) *A triple pattern t is defined as triple* $t = (\hat{s}, \hat{p}, \hat{o})$*; each element is either a constant or variable. The triple pattern answer is defined as* $t^a = \{d_t \in D \mid match(t, d_t) = 1\}$*; and*

$$match(t^a, d_t) = \begin{cases} 1, & \text{if } \forall x_i \in t^a, \forall y_i \in d_t, i = \{0,1,2\} : \\ & (x_i = y_i) \text{ or } x_i \text{ is a variable} \\ & \\ 0, & \text{otherwise} . \end{cases}$$

The SPARQL query and its answer can now be precisely defined:

Definition 3.2 (Query, Query Answer) *We refer to a query q as a set of triple patterns* $\{t_1, t_2, ..., t_n\}$*. This set composes a query graph* $q_G = \{q_V, q_E\}$*;* q_V *is a set of graph vertices given by* $q_V = \{v \mid \exists t \in q : t = (v, \hat{p}, \hat{o}) \vee t = (\hat{s}, \hat{p}, v)\}$ *The query answer* q_a *is the set of all the sub-graphs in an RDF graph G that are matching the query graph* q_G *and substituting the corresponding variables. A query graph* q_G *matches* $G_1 = \{V_1, E_1\}$ *that is a connected sub-graph of G if* $|q_E| = |E_1|, |q_V| = |V_1|$ *and the following condition holds:*
$\forall e_1 \in E_1, \exists! e_2 \in q_E : match(t, d) = 1, t = mapToTriple(e_1), d = mapToTriple(e_2)$, *where* $mapToTriple(e)$ *is the function that maps a data graph edge e to the corresponding triple as given by Definition 2.2.*

The query length is an important measurement of a query and given by the following definition:

Definition 3.3 (Query length) *Given query q and its query graph* $q_G = \{q_V, q_E\}$*; and let* $\hat{q_G}$ *be the undirected version of* g_G*. The distance between any two vertices* $v_1, v_2 \in q_V$ *which we denote as:* $d(v_1, v_2)$ *is the count of vertices in the shortest*

path from v_1 *to* v_2 *in* $\hat{q_G}$. *The length of* q *is the maximum distance between any two vertices in its graph which is given by:*

$$q_l = \max_{\{\forall v_1, v_2 \in q_V, d(v_1, v_2) \neq \infty\}} d(v_1, v_2) \tag{3.2}$$

$$q_l = \max_{\forall v_i, v_j \in v} dist(v_i, v_j)$$

We can now state a clear definition of the workload:

Definition 3.4 (Queries Workload) *A collected workload up to time* t *is defined as a set:* $Q^t = \{(q_1, f_1), (q_2, f_2), ..., (q_m, f_m)\}$, *where* q_i *is a SPARQL query, and* f_i *is the frequency of its appearance in the workload. The workload answer* Q^{t^a} *is the set of the query answers of* Q^t.

$Q(t_1, t_2)$ *refers to the workload collected in the time period* $[t_1, t2)$.

3.3.1 Real-world Workload Analysis

There are many live services that provide SPARQL endpoints allowing users to run their own queries on RDF data-sets. However, the users' queries logs are not available publicly, but there are multiple research works [12, 30, 68, 61] that deeply analyzed the real queries logs and produced their properties such that testing queries that are simulating the real log, can be relatively easy generated.

- Frequent patterns often exist with different levels of distribution and impact. Some of these patterns are frequent in a very limited time period [12]. These limited periods are justified by users tuning their queries until getting satisfying results.
- There is a detectable correlation between the used data sets and the complexity distribution among the issued queries. See Figure 3.4.
- A correlation between the queries' shapes and both of the evaluation time and the result size.

From the above points, a workload aware system in which a workload is used as a measurement subject to adapt the storage structure, should not assume fixed trends. Instead, the system should adapt also with the workload itself. This implies evaluating the workload properties dynamically at run time and measure their effectiveness. These measurements are further used to increase the impact of highly

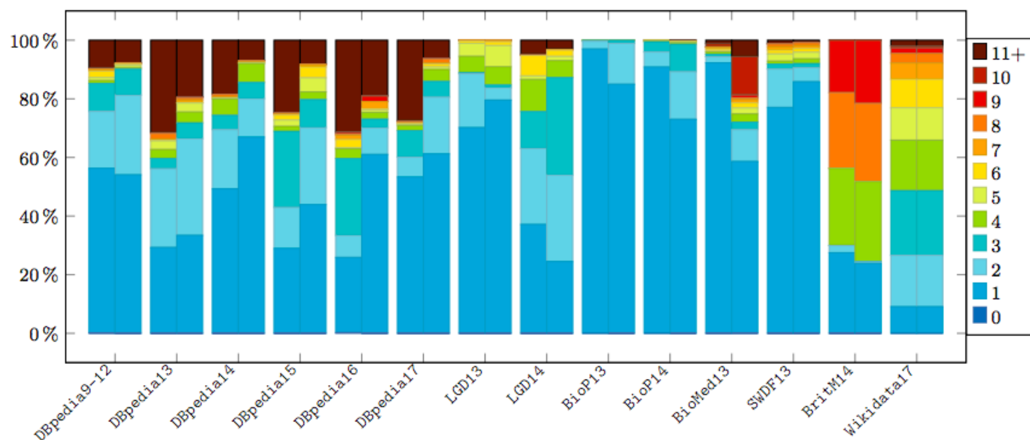

Figure 3.4: Percentages of queries exhibiting a different number of triples (in colors) for each dataset for Valid (left hand side of each bar) and Unique queries (right-hand side of each bar) as appearing in [12]

.

effective properties and obliterate the impact of those with low effectiveness. Having this functionality allows the system to apply the adaption in different levels of workload quality.

3.3.2 Evaluation Locality

In this subsection, we consider how workload queries are projected and interacted with an RDF graph. From Section 2.6 and Definition 3.2, we have seen that the query execution is the process of finding all the subgraphs in the main RDF graph which match the given query's graph. The way of how those subgraphs are widespread in terms of locality has a big effect on the execution complexity and performance. From an analytical point of view, we classify this locality interaction between the query and the data-set into the following two aspects:

- The data-set aspect.
- The query-graph aspect.

Locality with respect to the data-set

The RDF data sets that form a big graph have usually non-equal access rate. This also means that some parts of the graph show heavy access, while most parts of the

graph are been accessed with a much lower access rate [66]. In a set of real-world queries that were targeting DBpedia, more than 90% of the queries target only 163 frequent sub-graphs [60]. Detecting these hot parts of the RDF graph is one of the fundamental methods to make the systems more adaptable to the workload. However, there are different methods and approaches to detect those frequent parts, and there are different strategies to employ them for the sake of system benefits. Some systems use the detected frequent patterns to recognize which parts of the data-graph are more relevant to replication [37, 31], others consider those parts when doing the data partitioning [26, 60], and others use them to detect the most important pieces of data for caching in main memory [59]. Those methods with their challenges were reviewed in Chapter 2. One of the most challenging factors that affect the outcomes of these approaches, is their flexibility to cope with different workload heterogeneity levels, and their ability to tune themselves dynamically when the workload changes over time.

Locality with respect to the Query-graphs

The shape of the query itself and its layout has a very important effect on the amplitude of its spread within the RDF graph. In most cases, this can be estimated merely by observing the query graph, and the distribution of variables and constants within its structure. That was already introduced earlier in Section 2.6.1, in which a query can be classified into bounded and unbounded and each type has its own localization impact:

- Bounded queries: the query graph has at least one constant vertex within its structure. The query execution is going to stick within a limited locality in the RDF graph, that is the location of the constant vertex and its neighbours, given that the system has the appropriate index to locate that vertex within its indexing structure, as was explained in Section 2.4.

- Unbounded queries: if the query graph contains no constants in any of its vertices, but has constants within its predicates, the number of the sub-graphs that may match the query graph is unlimited. The only possible way to define an answer for such query is to follow how the constant-edges are connected within the query-graph and find matching sub-graphs in the RDF graph. Thus, the execution in this case is widespread in terms of the data locality.

3.4 Workload Rules

In Section 3.2.2, we classified the workload analysis used to generate resources access rate into general, specific, and generalized. The basic purpose of that analysis is to find resources access rates under certain employment options. However, we could have multiple access functions each is derived from a certain analysis. Moreover, some of the access functions are targeting the same vertices, which creates multiple access values. In order to systematically deal with these issues, each independent access function is encapsulated within an *access rule*. An access rule has its source, access function, and set of affected vertices.

Definition 3.5 (Access Rule) *An access rule ϖ is defined as the following element:*

$$\varpi = (s, \hat{V}, a)$$

where a is a function that assigns an access rate value [3] *to each $v \in \hat{V}$, and s is a set of pattern functions that defines a set of vertices $\hat{V} \subseteq V$ as the following: $s = \{s_1, s_2, .., s_n\}$ such that $\forall s_i \in s$ there exist function $f(s_i) = V_s, V_s \subseteq V$, and $\hat{V} = f(s_1) \cup f(s_2) ... \cup f(s_n)$.*

Mapping the workload analysis into access rules enables the power of comparing and aggregating different rules. It also makes it quiet easy to plug new rules into the adaptable system. For example we define in Chapter 5, two rules about the border replication. The first include a general access function, while the second has a specific access function. The two access rules are aggregated into one rule that sketches the net access values of the data in the border region.

The access rule draws the resources access rates under given employment options. However, the cost model needs further the benefit and cost functions. For that, we define the operational rule. An access rule can be converted into an operational rule for the storage adaption purpose by providing the performance benefit function as well as a destination index. The performance benefit is measured relative to the cost.

Definition 3.6 (Operational Rule) *An operation rule is composed by associating an access rule ϖ, by a destination index χ and a relative performance gain function Δ:*

$$\varpi_{op} = (\varpi, \chi, \Delta)$$

By applying Formula 3.1, a benefit function for each operational rule can be defined:

$$b(v) = \Delta(v) \cdot a(v)$$

[3] The access rate is explained further in Section 3.2.1.

A rule targets a set of vertices that is part of the RDF graph. However, there are cases when more than one rules target the same vertex. That requires stating the method of aggregating the rules such that each vertex has a net rule targeting it and represents its net access rate. For that purpose, we state in the following the aggregation properties for access and operational rules. Moreover, we state two other properties which are the *projection* and *source ordering*, which will be used later when stating the rules about indexes, replication, and join cache.

- *Property 1 (Rules Aggregation).* For two access rules ϖ_1 and ϖ_2, if $\hat{V_1} \cap \hat{V_2} \neq \emptyset$, then a new rule ϖ_g can be defined, that is the aggregation of ϖ_1 and ϖ_2 as $\varpi_g = aggregate(\varpi_1, \varpi_2, e_1, e_2) = (s_g, \hat{V_g}, a_g)$, such that $s_g = s_1 \cup s_2$, $\hat{V_g} = \hat{V_1} \cup \hat{V_2}$, and a_g is defined as the following:

$$a_g(v) = \begin{cases} a_1(v) \cdot e_1(v) + a_2(v) \cdot e_2(v), & \forall v \in \hat{V_1} \cap \hat{V_2} \\ a_1(v) \cdot e_1(v), & \forall v \in \hat{V_1} \wedge v \notin \hat{V_2} \\ a_2(v) \cdot e_2(v), & \forall v \in \hat{V_2} \wedge v \notin \hat{V_1} \end{cases}$$

  where e_1 and e_2 are weighting functions representing the effectiveness of ϖ_1 and ϖ_2 respectively.
  We also refer to the function $aggregate(\varpi_1, \varpi_2, e_1, e_2)$ as $aggregate(\varpi_1, \varpi_2)$ to indicate $e_1 = e_2 = 1$.

- *Property 2 (Operation Rules Aggregations).* For two operations rules ϖ_{op1} and ϖ_{op2}, if $\hat{V_1} \cap \hat{V_2} \neq \emptyset$, and if they share the same destination index χ, then a new rule ϖ_g can be defined, that is the aggregate of ϖ_{op1} and ϖ_{op2} as $\varpi_g = aggregate_{op}(\varpi_{op1}, \varpi_{op2}) = (\varpi, \chi, \Delta)$, such that $\varpi = aggregate(\varpi_1, \varpi_2)$, and Δ is defined as the following:

$$\Delta(v) = \begin{cases} \Delta_1(v) + \Delta_2(v), & \forall v \in \hat{V_1} \cap \hat{V_2} \\ \Delta_1(v), & \forall v \in \hat{V_1} \wedge v \notin \hat{V_2} \\ \Delta_2(v), & \forall v \in \hat{V_2} \wedge v \notin \hat{V_1} \end{cases}$$

- *Property 3 (Rule Projection).* For an access rule ϖ_1, if there exist a pattern function s_p such that it defines $\hat{V_p} \subseteq \hat{V_1}$, then a new rule ϖ_p can be defined, that is the projection of ϖ_1 on s_p as $\varpi_p = proj_{\varpi_1}(s_1) = \{s_p, \hat{V_p}, a_1\}$

- *Property 4* (Source Ordering). For an access rule $\varpi = \{s, \hat{V}, a\}$, where $s = \{s_1, s_2, ..s_n\}$, then the following elements of ϖ can be ordered:

1. vertices by their access.
2. source pattern functions s by their average access values $a_{avg}(s)$. There is no loss in accuracy if for each $s_i \in s$, the access function a assigns the same access value to each $v \in f(s_i)$.
3. for an operational rule $\varpi_{op} = (\varpi, \chi, \Delta)$, its sources can be ordered by their average benefit values $b(s) = a_{avg}(s) \cdot \Delta_{avg}(s)$. where $\Delta(s)$ is the average performance gain for each source in the source set. There is no loss in accuracy in the case of $a(v)$ assigns the same access value to each $v \in f(s_i)$, and $\Delta(v)$ assigns the same performance gain value to each $v \in f(s_i)$ for each $s_i \in s$.

After ordering the sources set s, the head source that stands at the top of the sources set is referred to as $\bar{s}$.

We explain next the basis used to derive the general rules based on the collected workload, then describe the concept of heat query map in order to use it for finding the access values of the specific rules.

3.4.1 Basic Measurements for The General Rules

We mention in the following points, the average measurements that represent the basis to build general rules about indexes and replications in the next two chapters.

- **The average query length.** Given a query q, its length q_l is defined by Definition 3.3. For a collected workload Q, we can find the average length q_{lm} by calculating the arithmetic means for all the queries it contains. This value represents the expected length of the next query the system receives.

- **The average query size.** Definition A.4 determines a query size in terms of its graph, evaluation, and result. Similar to the previous point, we extend the measurement from the query level to the level of a collected workload, by calculating the arithmetic mean for each of the given measurements. The mean values serve as the general expectation of the system's next query size.

- **The average indexes usage.** The execution of a query is carried out by using indexes (Section 2.6). The system observes the execution of the collected workload on the level of each index, and record for each index χ the count of usage or frequency of access. The relative value of this frequency with respect to the total system's indexes usage represents the general rank of importance of that index.

3.5 Heat Queries

In Definition 3.4, we have defined the workload as a set of pairs where each pair represents a query with its frequency. However, we need to store the workload in a structure that keeps the relationships between the queries as well as their frequencies. That will provide the following advantages:

- It helps following up the workload shape development over time.
- It measures the access rate of specific part of the data-set, thus it fits within the concept of the specific rules.
- It Allows the generalization of the workload to other areas in the data set.

The *heat query* has a concept inspired from the *heat map* but instead of the matrix of heat values in the heat map, we have a graph of heat values in the heat query. The workload is then seen as a set of heat queries. While the heat query extends the original concept of *global queries graph* originally proposed by Partout [26], it provides better generalization approach as we explain in Section 3.5.4.

The heat query are divided into two types: the main heat query graph which simulates the structure of the query graph, and the secondary heat join map that simulates the structure of the join across queries triple patterns. Before going into the method of heat queries and heat join maps generation, we formally define both of them.

Definition 3.7 (Heat Queries sets) *We define the following two sets:*

- *set of main heat queries:* $H = \{h|h = (q_G, F, X)\}$*, where* q_G *is a query graph as given by Definition 3.2,* F *is a function that assigns frequency values to each vertex in* q_G*, and* $X = \{x|x = (\chi, a, b)\}$ *is a set of the indexes types used to evaluate* h *with their access and benefits values given by* a *and* b *respectively.*
- *set of heat join maps:* $H_j = \{h_j|h_j = \{P_j, E_j, l\}\}$*, where* $P_j \subseteq P$*,* P *is the RDF graph predicates set (Definition 2.1),* $E \subseteq P_j \times P_j$*, and* l *assigns a heat value to each* $e \in E$*.*

3.5.1 Heat Query Generation

We explain in this subsection the generation of the heat queries set out of a workload Q and an RDF graph G. During the accumulative building of the workload, each time a query q is executed, it forms a new heat query h as by Definition 3.7, with

heat frequency set to 1. Next, h is either added to the heat queries set H or combined with H, if there is a heat query $h_i \in H$ that has at least one shared element. The shared element is either a non-variable vertex or one or more triple pattern(s).
Combining two heat queries creates a bigger one, such that the shared vertices would be hotter by getting the summation of heats of both heat queries. The combining process is shown in Figure 3.5. When Q_2 is received, it makes some part of the previous heat query hotter by increasing its frequency. The same applies for Q_3 and Q_4. Any variable in the query is replaced by a single variable x to allow the variables to be directly combined. This is because the heat value in the heat query should eventually reflect the frequencies of the data-set vertices. This also happens when Q_4 is combined. It increases the heat value of C_1 in Figure 3.5 by one degree, and create a node of variable x with a heat value equal to 1. By this process, a heat query would be bigger in size with more workload queries getting combined regardless of their order. The vertices of a heat query h_q record two pieces of information: the count of this query as frequency or heat value, and the count of each index used (or to be used if the system doesn't have yet the optimal index for executing this query). The heat query shown in Figure 3.5 keeps a record of the queries vertices which can be directly mapped to the data graph vertices by finding the answer of the heat query. However, the heat query does not provide direct heat values about the predicates. For instance, we cannot directly tell from 3.5 how many times p_1 is joined with p_2. This is especially important for queries that have only predicates as constants. To overcome this problem, the system keeps a set of heat join maps beside the set of main heat query graphs. Figure 3.6 shows how a single join map is evolving.

3.5.2 Implementation Notes

Similar to the normal RDF graph, the heat queries are stored into its own indexes. However, we only need one basic operation on these indexes which is the lookup of any stored query by any of its constants. In this context, we need one index that is hashed on the constants of the vertices and one index that is hashed on the predicates. Each time a query is added, references to it (or pointers) are stored accordingly in the two indexes. For instance, if Q_3 in Figure 3.5 is received, two references are stored in the constants-index on C_4 and C_5, and further two references are stored in the predicate index on P_2 and P_3. A lookup operation on any constant would return a list of all heat queries that this constant appears in it at least once. For each triple pattern in the heat query, there is one value for the heat, one value for the *effect* (explained in Section 3.5.4), and references to two vectors. Each vector

has one entry value corresponding to each index type in the system. The first vector contains the access values, while the second contains the benefit values.
Whenever a query q is received, lookup operations are performed on its constants and then on its predicates. One or two of the following cases will happen.

1. No match is found, then a new heat query is created and added to the indexes accordingly.

2. A matching heat query h is found on one constant vertex. The found vertex heat is increased by one. A further match check is performed on the triple patterns of q and h, the heat values of the matching patterns are increased. Any non-matching patterns in q are added to h and to both of the constants and predicates indexes.

3. If Point 1 is fulfilled, and another heat query h_2 is found, h_2 is combined with h, and the heat of shared vertices are increased accordingly. The operation is repeated if further matching heat queries are found.

Upon the execution of q, the indexes' access rates and benefits in h are updated. A lookup of heat queries that are matching a vertex or pattern is a straight forward operation using the indexes. Finding the heat queries that are matching a given triple pattern is performed in two steps. The first step is to perform a lookup using either a constant vertex from the pattern or a constant predicate. The second step is to scan the triple patterns of the matched heat queries looking for a match to the input triple pattern.

3.5.3 Generalized Rules

A heat query reflects the impact of the workload on specific parts of the data, which are the parts that have been processed in order to find the answer of the queries which are composing the heat query. However, in many cases, the workload is small and requires a considerable amount of queries to be collected over time. Thus, there is a big need to detect how the workload is composing over time, and generalize its trend to predict future behaviour. The concept of generalization is also implemented by [26, 37, 60]. However, the generalization in those works needs fixed thresholds and settings. Moreover, it is very vulnerable to the bad workload or the workload that changes its trends. Thus, it is very important to measure the effectiveness of the generalized rule in order to relieve the impact of bad generalization and amplify the effect of good generalization. We provide next, our method of generalization that

pays attention to the effectiveness of the generalization and avoids the drawback of the previous given works in this regard.

3.5.4 Heat Query Anonymization [4]

The triple pattern t is defined in Definition 3.1 as an ordered triple of items such that each item can be either constant or variable. If t contains variable at either the subject or object places, the triple pattern answer is tight to a certain vertex in the data-set that matched one of the t constants. However, if t contains one constant which is only on its predicate, and has variables elsewhere, the triple pattern can be projected to all the vertices that have that constant as a predicate on one of their edges. In this context, the anonymization process of t is the replacement of the constants on its subject and object places with a single variable x, while it keeps the constant at the predicate place. Moreover, the anonymization process ensures that it also replaces all the variables within the workload with the same replacement variable x, such that it allows the union of the matching heat queries according to the process explained earlier in the heat query generation process. If t contains a variable at its predicate place, it is excluded from the anonymization process.

The anonymization process generalizes a specific heat query and increases the number of triples it targets; however, we need now to measure the effectiveness of this generalization. The first thing to do in this regard, is to look again at the workload and observe how the anonymized heat query could have performed. For example, given a heat query h and its anonymized version $\grave{h}$. Due to the generalization, $\grave{h}$ naturally targets more vertices in the RDF graph than h. If the generalization is good, we would expect a future query to target those extra vertices, or we could instead rely on how the already received queries have been targeting those vertices. This can be achieved by looking at how the anonymized heat queries are getting unified. If $\grave{h}$ has been combined with other anonymized heat queries and its total heat has been greater than h, this indicates a more probable access rate for the extra vertices targeted by $\grave{h}$. On the other hand, if $\grave{h}$ still has the same heat and shape of h, this indicates that the generalization of h has not been seen yet in the workload. Thus, we can set the effect of anonymized heat query by the following formula:

$$effect(\grave{h}) = \frac{heat(\grave{h}) - heat(h)}{heat(h)} \tag{3.3}$$

Where, $heat(h)$ is a function that returns for the heat query h the summation of all of its heat values. The $effect(\grave{h})$ is a factor that is always taken into consideration

[4] Part of this subsection appeared in our publication [3].

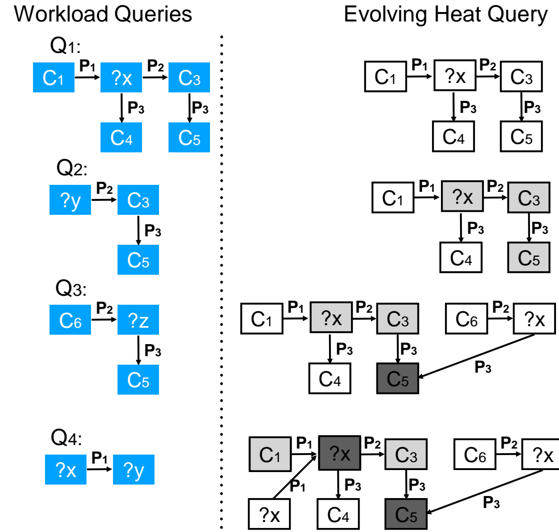

Figure 3.5: Heat query evolving from four queries

when using the anonymized heat map $\grave{h}$ in a related generalized workload rule.

If H represents a set of all anonymized heat queries in the system so far, then we can define $H_q(v)$ which returns for any $v \in V$, a list[5] of heat queries that v is associated with, or null if v does not belong to any heat query. The work of $H_q(v)$ can be further explained when we recall that the heat query h is a collection of one or more queries graph. Then h can be projected and executed on the data graph G, and the result of the execution according to Definition 3.4 is a set of sub-graphs where v can be checked whether it belongs to it or not.

3.5.5 Triples Access Rate By Heat Queries [6]

Given a query q that is to be executed on RDF graph G, the probability of a any vertex $v \in V$ to be part of the query answer q_a, under no previous workload assumption may be naively set to be uniform:

$$p(v) = \frac{|q_a|}{|V|}$$

[5]In most case each vertex is associated with one heat query due to the process of heat queries combination on the shared items.

[6]This derivation is also given in our publication [3].

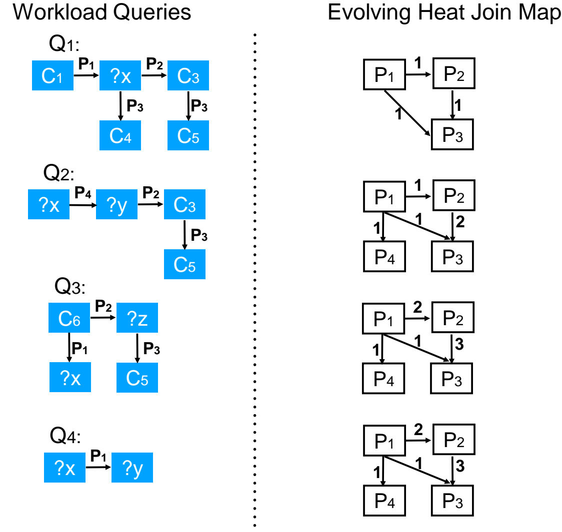

Figure 3.6: Heat join map evolving from four queries

But when we take previous workload Q into consideration, there is a frequency of appearance per vertex. Thus the above probability can be changed to:

$$p_w(v) = \frac{freq(v)}{\sum\limits_{\forall v_i \in V} freq(v_i)} \tag{3.4}$$

To keep the math compact, we assume a length of query answer equal to one. The value of $p_w(v)$ represents the usage factor or rate of access of v by its frequency in the heat query. The rate of access to v as expected by the anonymized heat queries set is then given by:

$$accessRatio(v) = \frac{\sum\limits_{\forall h_i \in H_q(v)} freq(h_i, v) \cdot effect(h_i)}{\sum\limits_{\forall v_i \in V} freq(v_i)} \tag{3.5}$$

where $freq(h, v)$ is the frequency of v as given by the anonymized heat query h.

$access(v)$ in Formula 3.5 can be separately specified for each certain index type, by specifying the index type in $freq(h, v, index)$:

$$accessRatio(v, index) = \frac{freq(H_q(v), v, index) \cdot effect(H_q(v))}{\sum\limits_{\forall v_i \in V} freq(v_i)} \tag{3.6}$$

A non-relative value of the access function is given by:

$$access(v, index) = freq(H_q(v), v, index) \cdot effect(H_q(v)) \tag{3.7}$$

Using the set of heat join maps, we can get an access function to any two predicates. This access value represents the probability of the two predicates being in a single query with respect to the workload. This can be written as:

$$accessRatio(p_1, p_2) = \frac{freq(H_j(p_1, p_2))}{\sum\limits_{\forall p_i \in P} freq(p_i)} \tag{3.8}$$

A non-relative value of the previous function is given by:

$$access(p_1, p_2) = freq(H_j(p_1, p_2)) \tag{3.9}$$

3.6 Heat Query Specific Rule

The main purpose of the heat query structure is to estimate the access rates of the resources based on the workload. However, those resources have also other access rates which are calculated based on the average behavior of the workload. Thus, the access rates by the heat queries are considered specific rules, and the average access rates are considered general rules. These rules can be then aggregated for each resource using the rules' properties explained earlier in Section 3.4. For this reason, we transfer the heat queries into specific access rule in the following definition:

Definition 3.8 (Heat Query Specific Rule) *The heat queries set (Section 3.5) defines a single rule for each index $\chi \in X$ as:*

$$\varpi_{he}(\chi) = (s_{he}, \hat{V}_{he}, a_{he})$$

where:
$s_{he} = \{s | s = q_G \forall (q_G, F, X) \in H\}$ *the set queries graphs of all heat queries in the system as by Section 3.5.*
$\hat{V}_{he}$ *is the answer of H, $a_{he} = \{(v, a) | \forall v \in \hat{V}_{he}, \forall \chi \in X, a = access(v, \chi)\}$, and $access(v, \chi)$ is given by Formula 3.7.*
We define further R^{he} as a set of all heat query specific rules:

$$R^{he} = \{\varpi_{he} | \forall \chi \in X, \varpi_{he} = \varpi_{he}(\chi)\}$$

The heat query represents an access rule for each index in the system. The source of this rule is the heat query itself, which has a set of connected patterns and their

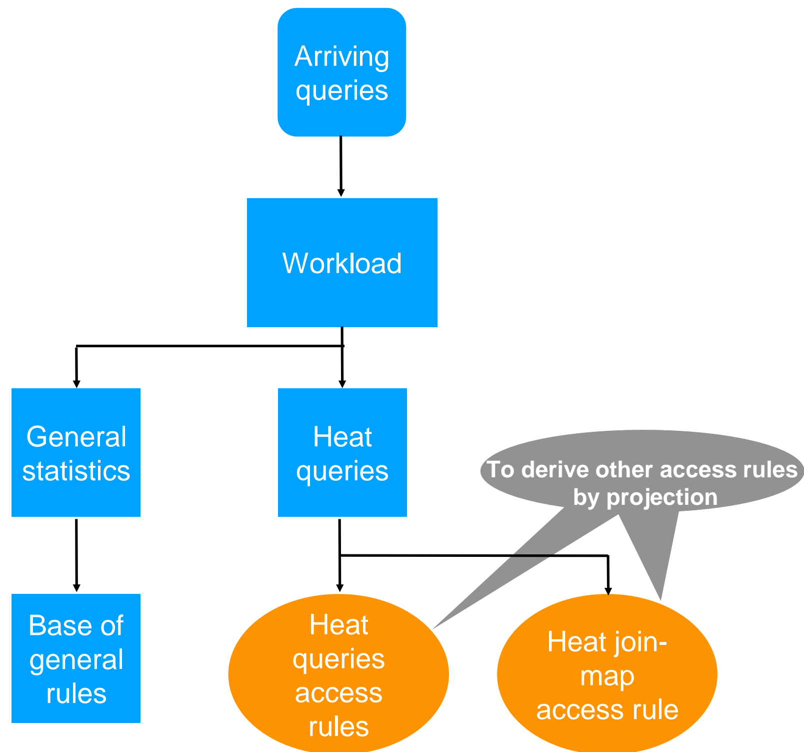

Figure 3.7: Workload rules' maps

frequency of access. The access formula assigns access value to each vertex in $\hat{V}_{he}$. However, the formula is a function of the heat query, and thus, the access function is a function of the source, which means that we can find the access value of each pattern in the heat query. This enables Property 4 (Source Ordering) of the rules property (Section 3.4), which allows ordering a rule by its source's elements without having to maintain its exact set of the vertices. This property enables important optimization in the universal space adaptation algorithm, as we explain later in Section 6.2.2.

3.7 Summary

This chapter presented the methods that we use to store and analyse the workload for the purpose of deriving the data-vertices access rates.

- The analysis of Real-world queries shows that the RDF workload often contains detectable trends and frequent patterns.

- We divide the storage space into *resources* unit. Each unit can be utilized by one *consumer* which is an equal size piece of data. The consumption can be done on one out of multiple indexes' *options*. A triangle of resource, consumer, options forms an assignment.

- For each assignment, we calculate the performance benefit and the access rate. The product of those two values gives the effective assignment's benefit.

- We use the workload to detect the access rates of vertices and indexes.

- The workload analysis is performed using access rules. Each rule is designed to look for certain trend in the workload. A rule can be projected on specific region of the data. Two rules targeting shared region of the data can be aggregated.

- The access rules that look for trends targeting specific vertices in the data graph are called specific rules. The general rules looks for average trends.

- A heat query is composed by combining multiple queries. It records the count of the frequent queries' items, the stats of the used indexes to evaluate the included queries.

- We use the heat queries set to create one specific access rule that assigns an access value to any vertex in the data set. That rule can be projected on any defined region in the data set.

- An access rule is transferred into an operational rule by providing a relative benefit function.

- The methodology of the rules allows the flexible plugging of futures rules.

Chapter 4

Local Storage

In typical key-value RDF stores, the triple data are stored into indexes. To achieve the required query execution performance, a triple store requires multiple data-wide indexes. Due to the high space impact, the stores choose to have only some of the indexes. The decision of choosing specific indexes is based on observation of the workload, and the store's storage-saving strategy. Instead of the fixed indexes strategy, we let the indexes dynamically adapts to the status of the workload and the storage space. Moreover, we integrate the indexes into the cost model and defines two access rules which are integrated into operational rules. Those rules are comparable to the join cache and replications allowing a universal storage adaption.

Contents

4.1 Storage Scarceness **70**

4.2 System Storage Hierarchy **72**

4.3 Indexes **72**

4.4 Problem of fixed Indexes **73**

4.5 Dynamic Indexes **74**

4.6 Indexes in The Cost Model **75**

4.7 Index Rules **78**

4.8 Index Rules Aggregation **79**

4.9 Cache Index **80**

4.10 Dynamic Indexes Evaluation **82**

4.11 Summary **88**

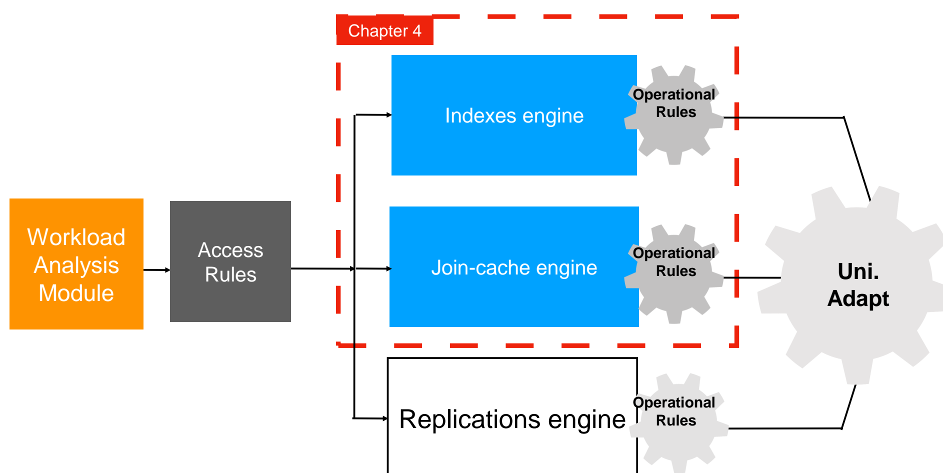

Figure 4.1: Chapter's scope

4.1 Storage Scarceness

In 1965 Moore has formulated a law that was then called Moore's law [54]. He predicted that the number of transistors per area unit will be doubled approximately every 2 years. This rate sets accurate trends for speed, size, and price in the digital world. The hard disk contains electromagnetic components, besides the normal electronic circuits. The average number of Gigabytes per price unit since 1982 followed similar exponential trends by being approximately doubled every 4 years. This behaviour is shown in Figure 4.2 which plots the average price of hard disks since 1982. The price data was collected by [52] from different sources. However, as it is also clear from the plot, the exponential trend has changed towards a linear trend since 2015. This is clearer in Figure 4.3 that has a linear y-axis.

On the other hand, the size of the data in the digital world doubles every two years, and is expected to be doubled every one year in the next decade. The ratio of data size growth to the disk size growth was, unfortunately, greater than one, and is getting much bigger.

The main memory or RAM is still small in size compared to the big-data scale. Although it showed better engagement to Moore's law, it has also deviated and failed to catch up starting from 2015. These given trends about the hard disk, main mem-

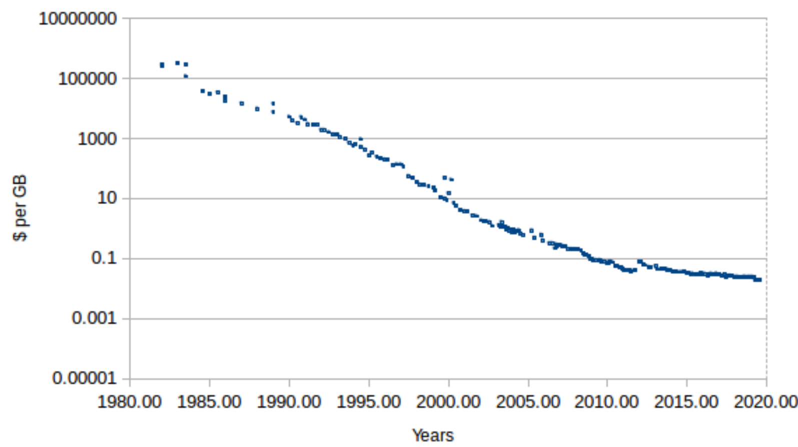

Figure 4.2: Average hard disk price through the years 1980-2019 as appeared in [52]

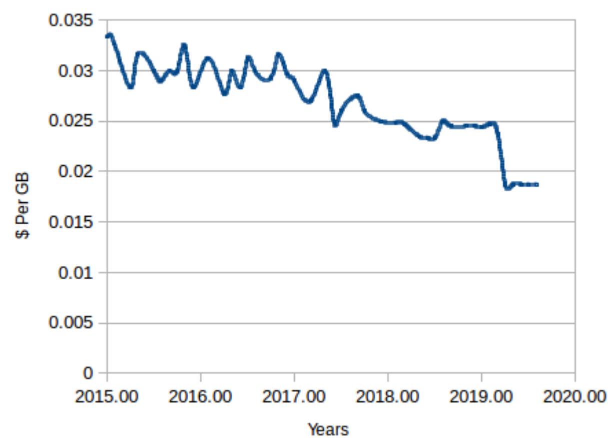

Figure 4.3: Average hard disk price through the years 2015-2019 as appeared in [52]

ory, as well as the trend of data size increase, highly motivate all the works that aim to the wise and optimized usage of the storage space. This should not however be interpreted as marking the works that always try to save storage space as winners. This is due to the following reasons:

- The ratio of RDF data-set to the available storage space could be small.
- The RDF system does not use the storage space only to store the raw data, but there are multiple storage employment levels for the sake of query performance. Increasing the space within RDF system does not only mean more space for new data, but it means better query execution time.

4.2 System Storage Hierarchy

In a typical computer system, the memory is available to the system in a hierarchical structure. The general trend is to have slower access time with a bigger amount whenever we move down in the hierarchy. That starts by the CPU cache which is the fastest and the smallest, then the random access memory referred to as main memory, and finally, the secondary storage which is usually a hard disk. Moreover, if the system is distributed there is another storage level accessed over the network. The access time of this level depends on the network used and on the access time of the certain storage unit on the remote node.
The access time of a typical storage unit shows general behaviour that can be described by the following formula:

$$accessTime(b^m) = randomDelay^m + \frac{f(b)}{transferRate^m} = \tag{4.1}$$

To access a serial block of storage of size b in a certain storage unit m, an initial cost of $randomDelay^m$ is paid given that the block b is located in a random place within the storage unit m. Afterward, the systems would pay a cost proportional to the size of the block and determined by the constant $transferRate^m$, given that the function $f(b)$ is linear. We will show in Sections A.3, A.4, and 5.2 how this formula is applicable on the level of a single storage unit.

4.3 Indexes

The classical model of the relational database is optimized for tables that represent entities with a fixed schema. The same schema is also reflected in the queries and in the tables indexing structure. As a contrast, the RDF data model falls in one big table with enormous embedded relations [1]. The indexing schema follows the triple structure as was explained in Section 2.4. Without an appropriate index, evaluating a single triple pattern may require scanning the whole data set, which is not feasible in the scale of the typical RDF data set size. Moreover, without an optimal index, a triple pattern may still require considerable scanning and filtering cost. We denote an index type by χ and define it as a function of its key. The key is best formulated as a triple pattern such that the index returns a set of all the triples in the data set that match this triple pattern. The implementation of the index follows the same basis given in Section 2.4. The index is always hashed on the first part of its key, and either sorted or hashed on its second part of the key. For instance, the SPo index

[1] Please refer to Section 2.3.1 for details about the triple table.

is hashed on the subject and sorted on the predicate. On the other hand the SP-o index is hashed on both the subject and predicate. The full list of indexes was given in Table 2.1.

Definition 4.1 (Index) *We define the following functions and properties for the index structure in the context of this thesis:*

- ***Index type.*** *Any index is associated with a type χ, which is a bit vector of length 3.*
- ***The index lookup*** *is defined as a function of its type and its key. The key is a triple pattern that has a constant corespondent to each 1 value in the associated type:*

  $$indexLookup(\chi, key) = \{d \in D \mid indexMatch(\chi, key, d) = 1\}$$

  where D is the data set of triples,

  $$indexMatch(\chi, key, d) = \begin{cases} 1, & \text{if } \forall x_i \in key, \forall y_i \in d, \chi_i = 1, i = \{0,1,2\} : \\ & (x_i = y_i) \text{ or } x_i \text{ is variable} \\ \\ 0, & \text{otherwise} . \end{cases}$$

  and $indexLookup(\chi)$ returns a set of all the triples in index χ.
- ***The index location.*** *The index type χ may exist in any physical storage level l, which is if specified, considered an independent index type labeled as χ^l.*
- ***Access Time.*** *Each index has an access time $\alpha(\chi)$, which is the average time required by an index to return the required set of triples.*
- ***Index Collection** X. The system keeps at any time a collection of indexes. $getOptimalIndex(key)$ is a function that returns the most optimal index type available in the index collection X for the given key, or $\emptyset$ if no index for this key is available.*

4.4 Problem of fixed Indexes

The main factor that limits the system's ability to build enough indexes is storage space. That case is mostly observed when the ratio of storage space to the data

set size is not high enough. Increasing this ratio means more free space which the system optimizer can use to index more data with more index types. That would lead to a decrease in query execution time. Having a fixed indexing structure could be either federated or conservative. Federated fixed indexing-structure like what is followed by RDF-3X [56] and Hexastore [81] who build a full-house set of indexes, is well known to require a higher level of storage space consumption. That may make the system not able to receive more or bigger data set, or not able to provide space for replication and caching. The problem is amplified when the system applies the given strategy on main memory that is more limited in size, or when the system receives a workload that rarely uses some types of already built indexes.
Following conservative indexing-structure like [64, 75, 58, 69, 46] would make the execution of the queries which require non-existing indexes more expensive. Moreover, the system could have an abundance of free storage space due to a small data set to storage space ratio; however, it still not able to employ this storage for performance. On the other hand, in the case of limited storage space, the workload received by the system can target only small parts of the data set. Those parts if had been recognized and fully indexed, it would highly optimize the total queries execution time.

4.5 Dynamic Indexes

Our solution to the problem of fixed indexes is to follow a dynamic indexing structure by allowing the system to have a flexible set of indexes that better suit its needs. In this context, the system decides the types of indexes to build upon an optimization process. That process is based on the analysis of the workload as well as the availability of the storage space. This optimization process integrates also the replication and queries cache. Within the dynamic indexing structure, each index does not necessarily cover the whole data set, but instead may cover any part of it, as long as it guarantees that any indexed data are completely and fully indexed. For instance, the SPo index can have any vertex in the data set to be indexed as subject-key, but once the system decides to index that vertex in the SPo index, it must assure that the full list of required triples is stored as a value. Thus, the saving of storage space is carried out on the level of the index-key but not on the level of the values. Failing to assure this would make the query engine unable to decide whether more values for a certain key are still available somewhere within the indexes set, and would substantially increase the query processing cost. Given that the number of edges per vertex in many RDF graphs is not uniform but shows a high variation,

we would have different storage overheads for the used index-keys. That overheads depend on the size of the lists of the triples that are associated with each key. This is translated into a difference in storage costs besides the difference in performance benefits.

Referring to our cost model that is defined in Section 3.2, we identify each index type χ as a single resource's option. Any vertex $v \in V$ in the data set can be employed by being indexed in any of the options. In order for the optimizer to make its decision about the proposed options, it needs to determine the cost and effective benefit of each option, given that according to Formula 3.1, the effective benefit is the product of the absolute benefit and the probability of access.

4.6 Indexes in The Cost Model

The dynamic index approach allows an index to grow or shrink in size based on the storage availability, workload as well as the integration with other storage consumers. That requires fitting the dynamic indexes into the cost model of Section 3.2.1. For that, we need the benefit of putting a data element in a certain index, its cost besides its access rate.

4.6.1 Index Cost

Any triple can be indexed in any index; however, to assure index consistency, assigning triples to indexes are performed on the level of the vertices. For instance, if a certain vertex v in the data graph is to be indexed in SPo index, then all of the triples that have v as subject must be indexed. Thus, the space cost of assigning v to SPo is the number of those indexed triples. This value can be easily measured by the system at the time of the indexing. We refer here for that measurement with the following function:

$$storageCost(v, \chi)$$

The function $indexLookup(\chi, key)$ given in Definition 4.1 requires the key to be a triple pattern consistent with the index type χ. For this reason, we need to transfer a vertex v to the triple pattern that can be used as key for a given index type. For that purpose we use the function called $key(v, \chi)$. For example if χ is SPo, the vertex the following triple pattern is returned: $(v, ?x, ?y)$, where $?x$ and $?y$ are variables. If χ is OPs, the returned triple pattern has v in the location of the object.

4.6.2 Index Benefit

The effective benefit according to the general cost model in Formula 3.1 is the result of multiplying the performance difference η by the probability of access ρ for the resource that is a single indexed item in this case.
The absolute benefit η of having a vertex $v \in V$ in index type χ can be recognized by the performance difference that the system gains when it makes the employment. From the perspective of the query execution, we look at the level of a single triple execution, and on the level of the whole query execution. We recall from Section 2.6.3 that to evaluate a triple pattern using the available indexes, there are three possibilities:

1. Use the optimal index if available, and directly provide the answer.
2. Use the sub-optimal index if available, plus an extra filter operation.
3. Non of the above is available, the case that requires a full data scan.

In the case of the second point, the benefit of having the optimal index available for vertex v instead of the sub-optimal is the removal of the extra filter cost (see Section 2.6.3 for details). In the case of the third point, the benefit will be avoiding a full data scan. We can formulate the triple lookup time $tripleLookupTime(v, \chi)$ for a vertex v in an index type χ_{l_2}, given that v is currently indexed by index χ_{l_1} :

$$tripleLookupTime(v, \chi_2^{l_2}) = \begin{cases} \text{full-data scan time}, & \text{if } getOptimalIndex(key(v)) = \emptyset \\ filterTime(\chi_1^{l_1}, key(v)) + \Delta, & \text{if } getOptimalIndex(key(v)) = \chi_1^{l_1} \\ 0, & \text{otherwise ,} \end{cases} \tag{4.2}$$

where $filterTime(\chi_1^{l_1}, key(v))$ is the time required to filter out the extra triples resulting from not using the optimal index[2], Δ is the difference in access time of v in $\chi_1^{l_1}$ and $\chi_2^{l_2}$, due to that l_1 and l_2 may refer to different physical media: $\Delta = \alpha(\chi_1^{l_1}) - \alpha(\chi_2^{l_2})$, $\Delta = 0$ for $l_1 = l_2$, and $\alpha(\chi)$ is the index access-time function that is given in Definition 4.1.

The above absolute benefit function is applicable in case of the query having a single triple pattern and thus requires only one data access path operation. However,

[2]See Section 2.6.3 for more details about the optimal index

in case of more than one triple pattern, the execution engine needs to perform a further join evaluation phase, and that results in multiple join-trees that differ with each other in the order of the join, the required indexes, and the performance (see Section 2.6.4 for details). The query optimizer works similar to RDF-3X [56] and selects the tree that is expected to show the best performance. However, in case of some indexes are not being available, the optimizer would choose the best performing tree which the system owns all of its required indexes. The performance difference between the best tree and the chosen tree is considered as points of benefits to the absent indexes for the given triple patterns. We label this tree performance difference as $treeTime(v, \chi)$. However, it is only feasible to be calculated on the level of the triple patterns and not on the level of single vertices. That is because the query trees where the performance difference is being found are composed of triple patterns. The calculated value can be generalized to all the vertices considered in the join operation. However, using the operation rules (Section 4.7) will remove the necessity of going down to the level of vertices, and stay in the level of patterns instead.

The index benefit function on the level of vertex can be then given by the following formula:

$$\eta_{idx}(v, \chi) = tripleLookupTime(v, \chi) + treeTime(v, \chi) \tag{4.3}$$

4.6.3 Index Access Rate

The last parameter to find in our cost model regarding the indexes is the access rate. According to the cost model 3.2.1, the benefit of assigning a vertex to be indexed in a certain index should be factorized by the access rate of that assignment. That access rate is base on the workload. The workload analysis methods were explained in Chapter 3. The methods resulted in two types of access rates: specific access rates given by the heat query and general access rates based on the average measurements. Moreover, the specific access rates are further generalized by the anonymization process.

The specific access rate of the heat query is expressed by an access rule (Section 3.6). We use that rule to derive the index specific rule using the projection property in the next section.

4.7 Index Rules

4.7.1 Index General Rules

The main idea behind having a general rule is to provide a more robust ground for the more specific rules. This is because the workload may not have detectable trends or may change these trends. The general rules are more resistant to the variation in workload quality. In this context, the storage optimizer collects basic statistic about each index in the Index Collection X, specifically the count of usage.

Definition 4.2 (Index General Access Rule) *For each index $\chi \in X$ we define a general rule as:*
$\varpi_{ge}(\chi) = (s_x, \hat{V}_x, a_x)$, *where* $s_x = \chi$, $\hat{V}_x = indexLookup(\chi)$ *as given by Definition 4.1 and* $a_x = indexAccess(\chi)$ *the access count of index χ in the workload.*
By Defining $\varpi_{ge}(\chi)$ for each $\chi \in X$, we result in a set of general rules R^{idx}_{ge}.

Applying Definition 4.2 on each index in system's set of indexes creates a corresponding set of general rules. A more specific rules are further generated in the next subsection, and both sets are aggregated into single index rules set so that we have a single access rule for each data element within an index.

4.7.2 Index Specific Rules

In most cases, the workload contains detectable trends that are more related to a specific part of the data set. We detect those trends using the heat query that is presented in Chapter 3. We have already given a rule in Definition 3.8 where a heat query h_e forms a rule $\varpi_{he}(\chi)$ that assigns an access value to each vertex in $\hat{V}$ the answer of h_e in the index χ. The index specific rule requires setting an access value to each vertex in a certain index that is located in a certain working node. Thus, The index specific rule $\varpi_{idx,sp}(\chi)$ can be derived from the heat query rule $\varpi_{he}(\chi)$ by projecting $\varpi_{he}(\chi)$ on a certain data partition.

Definition 4.3 (Indexes Specific Rules) *For each access rule in R^{he}, and for each destination index, the system defines a set of index access rules using the projection property:*

$$R^{idx}_{sp} = \{\varpi_{idx,sp}(\chi) | \forall \varpi_{he}(\chi) \in R^{he}, \varpi_{idx,sp}(\chi) = proj_{\varpi_{he}(\chi)}(r_i)\}$$

where $\hat{V}$ is the vertex set defined by each access rule ϖ in a certain partition, and r_i is the current partition for working node[3] i as given by Definition 5.1.

[3]The partitioning of the RDF graph will be given in details in next chapter, Section 5.3

4.8 Index Rules Aggregation

The vertices within each index have so far two access rules which are a general rule $\varpi_{ge}(\chi)$ and a specific rule $\varpi_{he}(\chi)$. In order to have a net access-value, the two rules need to be aggregated using Property 1 of rules properties (Section 3.4). By aggregating the two rules, a vertex that is targeted by both rules will get the average of both access values. The remaining vertices will have the access values given by the general rule.

Definition 4.4 (Index Aggregated Rules) *The system defines one set of index-aggregated rules given by:*

$$R_{as}^{idx} = \{\varpi_{idx}(\chi) | \forall \varpi_{idx,sp}(\chi) \in R_{sp}^{idx}, \forall \varpi_{idx,ge}(\chi) \in R_{ge}^{idx},$$
$$\varpi_{idx}(\chi) = aggregate(\varpi_{idx,ge}(\chi), \varpi_{idx,ge}(\chi), 0.5, 0.5)\}$$

The effectively of both rules in the aggregation process is set to 0.5 indicating equal weights. For instance, assume that at some point of time the SPo index has a general access value of 10 per vertex, and some vertex has specific access value of 50. Then that vertex will have a net access value of 30. If the workload quality drops and the heat queries are not able to detect specific frequent patterns, then the average index access will be the dominating value.

4.8.1 Finalizing Index Rules

The set of rules R_{as}^{idx} given by Definition 4.4, states one net access-rule to each index. However, the final optimization decision is made upon the performance benefits and not on the mere access rates. For that reason, we transfer each access rule in R_{as}^{idx} to an equivalent operation-rule by providing the benefit function that we have already given in Formula 4.3.

Definition 4.5 (Index Operational Rules) *The index access-rules set R_{as}^{idx} is converted into operational rule using Definition 3.6 as in the following:*

$$R_{op}^{idx} = \{(\varpi_{idx}(\chi), \chi, \Delta) | \forall \varpi_{idx}(\chi) \in R_{as}^{idx}\}$$

where Δ is a function that assigns a benefit value to each $v \in \hat{V}$ given by $\frac{\eta_{idx}(v,\chi)}{size(v)}$, η_{idx} is given by Formula 4.3 and $size(v)$ is the storage cost of indexing v in χ.

The benefit of each vertex is relative to its storage cost and multiplied by the access to produce the effective benefit of putting this vertex in that index. That

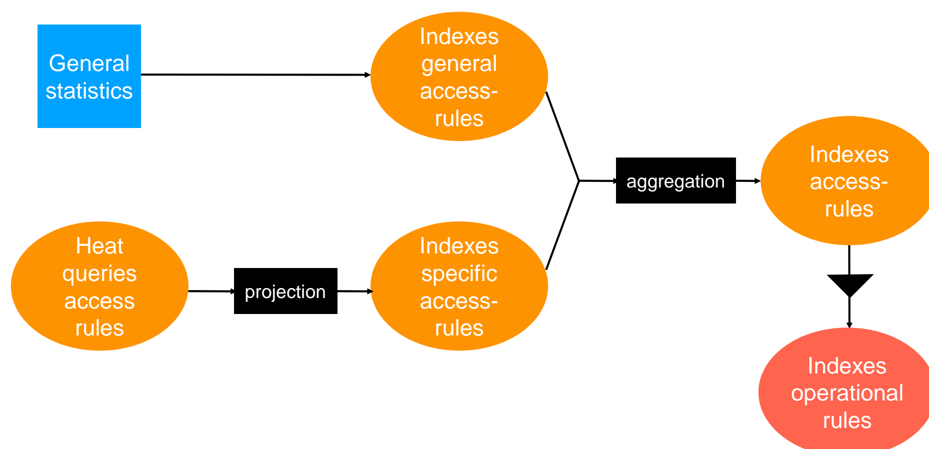

Figure 4.4: The map of indexes' rules

benefit is comparable with the other options' benefits. However, we show in Section 6.2.2 that we don't need to find the benefit on the level of each vertex. We set the level of calculation on the sources of each rule. The sources are a set of patterns which is much smaller in size than the set of the vertices.

Recalling the sources ordering property of Section 3.4, the indexes operational rules satisfy the ordering by benefits property given that the benefit function assigns the same benefit value to each the rule sources. The rule sources are the sources of the general rules aggregated with the sources of the specific rule.

4.9 Cache Index

A triple pattern evaluation is know as the data access path (Section 2.6.3) is performed by an index lookup. The evaluation of multiple triple patterns is then performed by joining the results of the data access paths (Section 2.6.4). This normal order of operations can be dramatically accelerated if the join result of two or more triple patterns is cached in what is called the cache index. In this context, a lookup in a cache index does not only perform a data access path operation, but return the result of the join evaluation.

The key in a normal data access path index is the constants of a triple pattern, and the index returns a set of single triples, in which every triple matches the key. In the cache Index, the key is the constants of two triples, and the returned output is

a set of triple-pairs, in which each pair is matching the given key. In the same way, the key can match more than two triple patterns.
Given that the most expensive join operation takes place when the query is unbounded (Section 4.10.2), we use in our system one type of cache index named PP-x. The key is a combination of two predicates p_1 and p_2 and the value is a set of triples pair, where each pair is in the form: $(s, p_1, x)(x, p_2, o)$. Thus, each pair is joined on x such that x is the object of the first triple and the subject of the second triple.
To be integrated into the index cost model, the storage cost of a cache index is found in the same way as a typical index. However, the performance benefit for the cache index is saving the joining time of the triple patterns that are indexed by the cache index. That can be written as the following:

$$\eta_{che}(T, \chi_2^{l_2}) = joinTime(T) \tag{4.4}$$

Where, T is a set of triple patterns, and $joinTime(T)$ is the time required to join them. The benefit is given as a function of the triple patterns, which directly fits with the heat query that is used to simulate the access of the workload to the data set (see Section 3.5). However, the benefit can also be given as a function of vertices, by mapping the triples patterns T to the vertices set that is resulting from the joining of T. The access rate of cache indexes within the access rules is calculated in the same way as the typical indexes (Section 4.6.3) except that we use the join heat maps instead of the heat queries.

4.9.1 Cache-index Specific Rules

The heat join map set H_j which is given in Definition 3.7 can be used to find the access values to the elements of the cache index PP-x. The vertices of a heat join map represent predicates, and each edge between two predicates mean that those predicates have been together in a single query. In this context, the source of the access rule is a set of all queries that can be composed out of all the heat join maps in H_j, such that each query contains two triple patterns formed as the following: $t_1 = (x_1, p_1, x)$ and $t_3 = (x_2, p_2, x)$, where both p_1 and p_2 are vertices connected with a direct edge in a heat join map. We officially state the access rule in the following definition:

Definition 4.6 (Cache-index Specific Rule) *We use the heat join map set H_j (Definition 3.7) to define the following access rule:*

$$\varpi_{che,sp} = (s_j, \hat{V}_j, a_j)$$

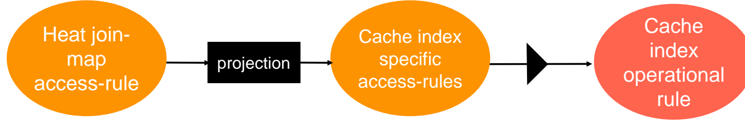

Figure 4.5: The map of cache index rule

where:
$s_j = \{(t_1, t_2) | \forall h_j \in H_j, \forall (p_1, p_2) \in E_j, t_1 = (x_1, p_1, x), t_2 = (x, p_2, x_2)\}$, *$H_j$ is the set of all heat join maps in the system, E_j is the edges set $\forall h_j \in H_j$ as given by Definition 3.7.*
$\hat{V}_j$ is the answer of s_j, a_j is a function that sets access value to each (t_1, t_2) ins_j using $access(p_1, p_2)$ given by Formula 3.9, and x,x_1, as well as x_2 are variables.

In Section 6.2.2, we show that we don't need to evaluate $\hat{V}_j$ during the optimization process but only during the assignment phase.

The access rule $\varpi_{che,sp}$ is then transformed into operational rule by setting the destination index to PP-x, and using the cache index benefit formula that was derived previously in Formula 4.4.

Definition 4.7 (Cache-index Operational Rule) *We define the following Cache-index operational rule:*

$$\varpi_{che,op} = (\varpi_{che,sp}, \chi, \Delta)$$

where:
$\varpi_{che,sp}$ is given by Definition 4.6, χ is the PP-x index, and $\Delta = \frac{\eta_{che}(\{t_1,t_2\},\chi)}{size(t_1,t_2)}$, $\eta_{che}(\{t_1, t_2\}, \chi)$ is given by Formula 4.4, and $size(t_1, t_2)$ is the storage cost of storing (t_1, t_2) in the cache index.
For integration with R^{idx}_{op}, we create the Cache-index Operational Rule set:

$$R^{che}_{op} = \{\varpi_{che,op}\}$$

4.10 Dynamic Indexes Evaluation

In this section, we provide practical evaluate that is limited to the dynamic indexes and cache approaches. In Chapter 7, we provide full evaluation to the indexes adaption as part of the universal adaption.

Since that we don't include the replication in this evaluation, we have setup one working node where the indexes structures are dynamic and adaptable with the workload according to their access and operational rules given earlier. We compare that adaptable approach to a fixed indexes approach, where indexes are initially fixed.

4.10.1 Detectable workload and High storage space availability

In the first part of the evaluation, we test the environment of high storage space availability. We refer to the latter term as system capacity which is the number of full indexes that the system can maintain. For example a system capacity of 6 means that the system has enough storage space to maintain 6 full indexes in its main storage unit. The RDF-3X always needs this capacity level to work. However, for a capacity level of 6, our system optimizes it towards both the indexes and join cache. The operational rule of the indexes is derived from two access rules (see Figure 4.4). The first is specific and based on heat queries. The heat queries work well in the case of the existence of detectable frequent patterns in the workload. The second access rule is general and based on the average access of each index. On the other hand, the operational rules of the cache are derived only from a specific access rule that is based on the heat queries (see Figure 4.5). As a result, in a capacity of 6, our system would assign the highly accessed data to the cache only in the case of the workload contains detectable trends by the heat queries, and always favours the indexes otherwise. In this context, we start with real-world workload trends, then we scale the workload quality down and measure the performance of the systems in comparison. The real-world workload is given by [60], where 90% of the queries target 160 frequent patterns. In this context, the first run in Figure4.6 has a workload quality of 90. The run is composed of two batches of queries, and each batch is of 1000 query. The objective of the first batch is to train the system, while we measure the performance for the second batch. All the generated queries are unbounded and of length three (tuning the bounding type and queries length is considered in more details in Chapter 7).
The RDF-3X has fixed six indexes residing in the secondary storage, and thus should show constant behaviour with the workload quality. However, there is some variation in its running time which is due to the operating system policy of managing the data between the main memory and secondary storage. The 3-indexes system resides totally in main memory and thus shows more stable behavior with the running time. The adaptable system made use of the high quality of the workload at the first run,

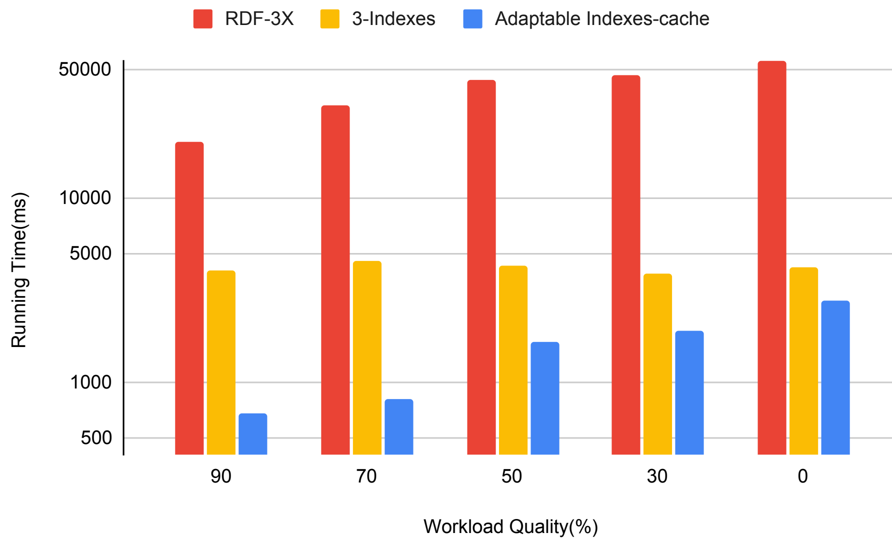

Figure 4.6: The Running times of adaptable indexes and join cache vs fixed approaches under storage capacity of 6

and employed its join cache. That helped answering most of the queries without paying the cost of joining. The effect of the cache decrease with the workload quality and reaches zero when workload quality drops to zero. At that stage, the specific rules are no longer in effect. However, the general rules still able to measure the average usage of the indexes and surpasses the fixed approach. Nevertheless, that is not very obvious in the capacity level of six, because the system has already the enough space to build most of the required indexes. To show the behaviour of the system in a more limited storage space environment, we set the capacity level to 3, and show the results in Figure 4.7. In the region of high workload quality, the adaptable system highly utilized its cache despite the limited storage space. However, the effect of the limited space was clearer as the workload quality drops to 50%. Nevertheless, the adaptable system continue to surpass the fixed 3-index system even when the workload quality drops to 0%.

4.10.2 Scalability of Queries Processing

In this part of the experiments, we follow the performance of the indexes for specific operations that are the data access path and the join evaluation, and the effect of increasing the data size. The data set was generated for this testing purpose, and the generation details are given in Chapter 7.

Recalling Section 2.6, a query q is composed of a triple patterns set, and its process-

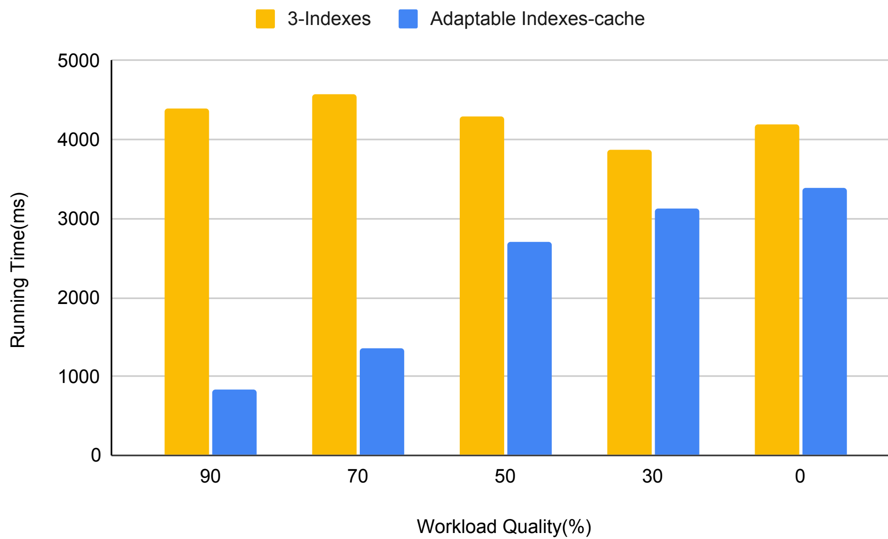

Figure 4.7: The Running times of adaptable indexes and join cache vs fixed approach under storage capacity of 3

ing is generally composed of two operations. The first is the process of finding the answers to the individual patterns using the available indexes (data access path), and the second is the process of joining these answers to find the answer of the query (join evaluation).

The performance of the **data access path** depends on the time required by the used index to return the requested data. We have two possibilities for the used index in this context: either the used index is an optimal or a sub-optimal (see Section 4.3). Moreover, and depending on the implementation, the optimal index is either hashed, sorted or hashed-sorted. For a general hashed index, we would expect that increasing the size results in a very limited impact on the performance, as the theoretical average behavior of such indexes is constant for the lookup operation. The lookup operation is done over a certain key that is composed of one, two, or three elements based on the index type and each element represents either the subject, predicate, or object. The lookup operation of the hashed index is performed directly by concatenation of the key's elements to create one key used for the index's internal hashed-table lookup[4]. For a sorted index, a lookup operation is composed of recursive three lookup operations for each of the index key's elements, and we would expect a logarithmic behavior with respect to each single lookup size. the

[4]In the graph terminology, the first key' element is used to get to a certain vertex, while the second element is used to get to one of the vertex's edges.

index size to get to any vertex and the same behavior to get to any of its edges. The hash-sorted index would require constant time to get the first hashed part of the key and a logarithmic time to get the sorted part which is usually composed of one element. On the other hand, a sub-optimal index requires a filter that has a cost that is linearly delimited by the size of the lookup operation.
Since the most used index type is the hashed-sorted index, the performance of the data access path, in this case, is linearly affected by the density of the graph (which is the average number of edges per vertex in the RDF graph) only in case the used index is sub-optimal. Otherwise, the effect of the graph density is affecting the performance of the data access path only logarithmically.
To evaluate this behavior of the data access path practically, we generated several single-triple queries, and run them on the system while changing the count of the total maintained triples by the system. The first query uses the SPo index and has a small result in terms of its size. Its performance is shown in table 4.1. In spite of that the data set has rapidly increased in size, the result size of the query is approximately constant as the constant subject of the query was not repeated anywhere in the newly added data until the final round where one more triple has appeared. Both of the index lookup time and query execution time showed approximately constant values despite the rapid increase of the data-size. The index lookup and the total query execution time is reasonably close as we don't count the dictionary and print time for this specific evaluation. Running the query using the sub-optimal index requires clearly more time; however, the time trend follows the same trend of the optimal index and does not rapidly change with changing the data size.
In Table 4.2, we considered running a query that requires the PSo index. The lookup time of the hash-based index is still the same, but we have a much bigger result size. This is a predictable measure since a predicate is expected to be much more frequent within the data-set than a single subject. However, we can easily observe from the query running time's values, the correlation of the time values with the result size values, and their stability with the data-set size. The bigger result size requires linear work to connect the output triples to each other and deliver the final result.

The **join evaluation** performs in a different way from the data access path. As was explained in Section 2.6.4, the total amount of required computations in each join step is related to the output of the preceding data access path stages in terms of their returned data size. The size of data from any index is again related to the selectivity of the items in the data set and to the data set size itself.

data size(M Triples)	lookup(ms)	query(ms)	result size	sub-optimal(ms)
1	0.3	0.33	2	1.05
10	0.3	0.34	2	1.14
100	0.32	0.35	2	1.15
1000	0.39	0.41	3	1.15

Table 4.1: The running time of a single-triple query that uses SPo index with respect to different data sizes

data size(M Triples)	lookup(ms)	query(ms)	result size
1	0.3	210	3127
10	0.3	688	11510
100	0.32	711	11510
1000	0.39	705	11510

Table 4.2: The running time of a single-triple query that uses PSo index with respect to different data sizes

To see the practical effect of data size on the join evaluation, we run a bounded chain query[5] of length 4 and list its response with respect to the data size in Table 4.3. The same response is listed for the unbounded version of the query in table 4.4. The unbounded query generated a higher number of triples during the processing and thus required more processing time. However, increasing the data size from 1 million to 10 million triples, increases the processing time of the unbounded query. This is due to the presence of more triples that have the same predicates of the query in the added data making the indexes providing more data for the join stage and increase its overall cost. The previous behavior is not noticed in the next increase of the data set size, due to the fact that the added data happens to not have the predicates presented in the query.

[5]The descriptions of query shapes are given in Section A.1.

data size(M Triples)	query(ms)	result size
1	0.93	2
10	0.96	2
100	1.1	2
1000	1.1	2

Table 4.3: Bounded chain query behavior with data set size

data size(M Triples)	query(ms)	result size
1	36	4
10	66	9
100	65	9
1000	88	9

Table 4.4: Unbounded chain query behavior with data set size

4.11 Summary

This chapter presented the a dynamic index approach that can replace the fixed indexes. We may summarize the chapter in the following points:

- Digital data increases in a faster rate than the storage space. RDF-triples stores have heavy storage consumption due to multiple levels of space requirements.
- In a typical key-value store, RDF data are stored into indexes.
- There are six types of indexes, and each index can be hashed or sorted.
- To avoid high storage consummation impact, a typical triple store chooses to implement some of the indexes based on hard observations to the workload.
- Dynamic indexes approach chooses the most beneficial indexes dynamically from the workload.
- To be fit in the cost model, indexes costs, benefits and access rates are derived.
- The workload access to the indexes is structured into two types: general and specific.

- The general workload access to indexes is transferred into general rules. Those rules simulate the the hard-observation in the fixed indexes approach.
- The specific workload access to the indexes is transferred into specific rules. They are derived from the heat queries rules by projection.
- Both of the rules are aggregated for each index and transferred into operational rules by providing the index benefit function. The indexes set of operational rules is comparable with the operational rules of the join cache and replications.
- While a normal index is used to index list of triples. A cache index is used to index pairs of triples. That saves the expensive cost of joining them, but costs more storage space
- We only defines specific rule to detect those triples that are highly beneficial to the system.
- A cache index operational rule allows it to be comparable with indexes and replications.

Chapter 5

Distributed Storage and Replication

The previous chapter considered the indexes as the local storage of the working nodes. This chapter gives the methods followed by the system to maintain a distributed storage of RDF. The main approach in this regard is to replicate certain data from the remote nodes' storage to the storage of a local node. We state the motivations, benefits, and types of such replication. We fit the replications in the cost model by stating their access and operational rules. That allows the replication to be compared against the indexes and cache, so that the system adapts its limited storage with the best options towards better performance.

Contents

5.1 Replication Motivations **92**
5.2 Distributed RDF Storage **93**
5.3 Initial Graph Partitioning **94**
5.4 Border Region **98**
5.5 Border Replication **99**
5.6 Load-balancing Replication **101**
5.7 Replication Aggregated Rules **104**
5.8 Summary **104**

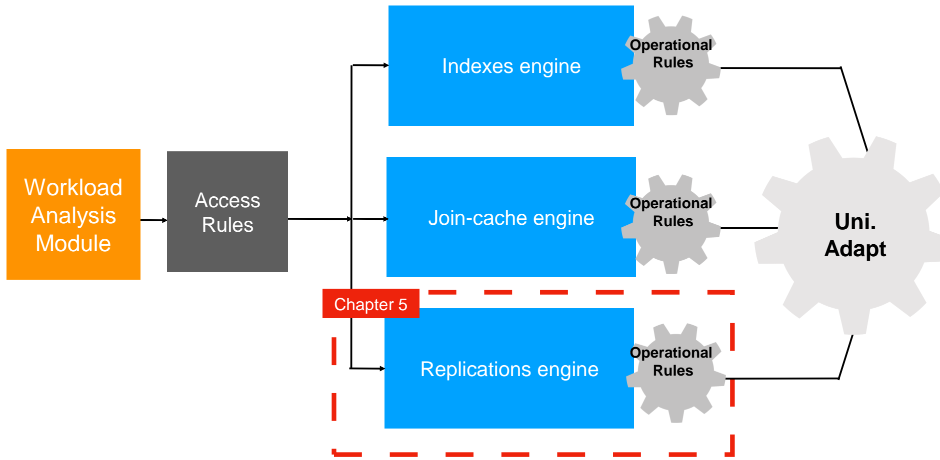

Figure 5.1: Chapter's scope

5.1 Replication Motivations

As we have been presented in Section 2.7, a distributed database is usually supported by replications for three main purposes: fault tolerance, workload balance, and the decrease of communication cost. The same motivations and objectives apply to a distributed RDF management system. However, the replication comes with the cost of consuming more storage space, and thus its benefit competes as one option among different options to utilize a storage resource.
Considering our cost model Section 3.2, the cost of the replication can be directly measured by the size of the data to be replicated. On the other hand, the benefit needs to be subdivided according to the required intent of the replication, which should be one of the three replication aims that we have mentioned above.

1. **Replicating to decrease communication cost**: During the query execution, some required data may not available locally due to the data partitioning. In order to continue the execution, the working node needs to get the data from another remote node where the data is available. However, if such data had been replicated from the remote node to the local node, the query execution time would save the network transfer time, which is given by the part $\delta \cdot b^{(m,j)}$ in Formula 5.1 given that the storage medium of the replicated data in the local node remains the same as the remote node; otherwise, the benefit calculation

needs to count for the difference in medium access time.

It is important to point out here, that in order for the replication in this category to work and provide the required benefit, the replicated data need to contribute partially in queries execution while some of the data should be available locally. The SPARQL query execution has obvious locality (i.e. neighboured graph-vertices are more probable to contribute in a single query, as was detailed in Section 2.6). If the used RDF data partitioning is a graph-based, then the replicated RDF data needs to be connected to the partitions border [38]. This is more precisely defined and detailed Section 5.4.

2. **Replicating to have better load balancing.** The system maintains a queue of received queries. If the queries arriving rate is lower than the system throughput, then the size of the queue is effectively zero, and the main focus of the system would be to serve each query as fast as possible. However, if at some point in time, a working node may receive queries more than its rate of queries execution, then this would lead to an increasing number of waiting queries in the queue. The problem of load unbalance between the working nodes might now emerge to the surface, if one or more of the working nodes has non utilized processing power. This happens if the working node can't execute any of the queries that are currently in the queue, because it doesn't have the required data locally. There are two options in this case which are either to move the data across the network to the remote node to help with processing the queued queries, or to perform replications for this purpose in advance.

3. **Replicating for better fault-tolerance.**: One of the basic motivation of having a distributed system is the increase in system availability by keeping multiple replicas of the same data in different hardware places within the working nodes. However, this type of replication is out of the scope of this thesis.

5.2 Distributed RDF Storage

For a system of distributed working nodes, each node has direct control over its storage resources, and has access to the other node's resources. However, such access has a network delay cost. In this context, we update the general storage access-time equation 4.1 to account for the existence of multiple working nodes. In this context, we have an accessing thread at working node i which wants to access a block of size

b and stored in storage unit m at working node j:

$$accessTime_i(b^{(m,j)}) = randomDelay^m + \frac{b^{(m,j)}}{transferRate^m} + \delta \cdot b^{(m,j)} \tag{5.1}$$

Or to make it more compact:

$$accessTime_i(b^{(m,j)}) = randomDelay^m + b^{(m,j)} \cdot (\frac{1}{transferRate^m} + \delta) \tag{5.2}$$

Where, δ is the network transfer rate between nodes i and j.
We can now use the same notation to denote the location of a certain index as well as its containing storage unit. Thus, the index $\chi^{(m,n)}$ is denoting an index of type χ within the storage unit m, and located in the working node n. Then, we can find the access time of an element within an index using Formula 4.1 by substituting for the block size as the following:

$$b^{(m,j)} = indexLookup(\chi^{(m,n)}, key) * B_t$$

where B_t is a constant represents the number of triples per block. From the perspective of a working node A, the remote storage units at some working node B are parts of the A's storage hierarchy. However, their exact levels depend on the network speed and the type of the storage unit on the remote node B. If the network speed is within the general limits of a local high-speed network, the working node A can generally access B's main memory faster than accessing its own hard disk. This situation has an important implication on the optimization decision, because what a node has decided to put in its memory affects also the performance of the remaining working nodes. As a results, this decision has to be taken collectively by consolidating with the other nodes. At the same time, the values of network transfer time and disk transfer time required to set this implication, are easy to be practically measured by the system.

5.3 Initial Graph Partitioning

We presented in Section 2.7.2 two directions of data partitioning which are: graph partitioning and hash partitioning. We compared their methods, outcomes, and effects on the performance of distributed query processing. In our system, we chose to perform graph partitioning aiming to reduce the communication costs while having steady ground for the workload-based adaption. In this section, we describe how the system performs this partitioning, and the approaches used to deal with its main issues which are the: partitions balance, the running time, and the border region.

We have seen in Section 2.1.4 that any RDF data set can be seen as a single graph. The RDF data partitioning is then directly reduced into a graph partitioning problem. Since that each partition is assigned to a single host, the number of partitions is known and equal to $|H|$, which is the number of hosts in our system. The objective of graph partitioning is related to how the system is going to employ and process the result of the partitioning process. Each host should use his share to execute the query it receives. The execution of a SPARQL query can be mapped into a subgraph matching problem (Section 2.6). While we are looking for a certain sub-graph q in a certain graph G_i, it may happen that we have reached the G_i border, and having a sub-graph $\hat{g}_i$ matches the sub-query graph $\hat{q}$, where $\hat{q} \subseteq q$ and $\hat{g} \subseteq G_i$. We would then be in the problem of answering the following question:
Does a data graph G_j (hosted by one of the neighbour nodes) have a sub-graph $\hat{g}_j$ such that both $\hat{g}_i$ and $\hat{g}_j$ form a single sub-graph $g = \hat{g}_i \cup \hat{g}_j$, and g matches the query graph q?
The answer to the above question requires moving data across the network, which is marked as a costly operation that we should avoid by the partitioning process. At the initial step (i.e. at system start-up), we don't have a certain assumption about the workload or the expected shape of system queries. Thus, the initial step should partition the input RDF graph while trying to achieve the minimum probability of queries to require data from neighbours, and increase the probability of local execution within hosts available data. One of the best methods to achieve this is by using a min-cut algorithm such that we have the partitions' borders with the minimum number of edges, thus with minimum probability of a query requiring sub-graphs data from neighbours. However, the process requires also some assumption of balancing the size of the resulting partitions. For this purpose, we use METIS and discuss its constraints and output in the next subsections.

5.3.1 METIS based Partitioning

METIS [45] is a set of tools developed by Karypis Lab [1] that serve the purpose of graph partitioning based on the multilevel recursive-bisection, multilevel k-way, and multi-constraint partitioning schemes. For the context of this thesis, we define the partitioning process of METIS as in the following:

Definition 5.1 (METIS Partitioning) *We refer to METIS as a function $metis(v)$ which for any $v \in V$ returns the static partition number which v belongs to. We could*

[1] http://glaros.dtc.umn.edu/

then define the partition $r_i = \{v \in V \mid metis(v) = i\}$. *V is the set of RDF vertices as given by definition 2.1.*

The general objective of such partitioning is to have a minimum number of edges (or sum of edges' weights) that straddle different partitions, while at the same time, it tries to produce balanced partitions. This balance is defined by METIS as a set of constraints. The set of constraints is mapped into a vector of weights for each vertex in the graph. All the vectors of all the vertices within a certain METIS partition can be summed into one total vector. The partitioning process will try to produce partitions that have equal total vectors. The default vector contains one element for each vertex having a value of 1. This vector instructs METIS to produce partitions that are of equal size in terms of the number of vertices. However, setting strict balancing constraints would reduce the accuracy of the general objective of having a minimum number of edges between partitions besides increasing the complexity of the optimization problem such that it costs more computation steps. For these reasons, METIS allowed the user to specify the maximum degree of load imbalance allowance, by using the option: *options[METIS OPTION UFACTOR]*. This option is defined by METIS [43] as the maximum allowed load imbalance among the partitions for each element of the constraints vector. We denote it as $imbalance(j)$ for constraint j in the constraints vector. The formula that defines $imbalance(j)$ for partition i according to [43] is:

$$\max_i\left(\frac{w(j,i)}{t(j,i)}\right) \leqslant \frac{imbalance(j)}{1000} + 1 \tag{5.3}$$

Where, $w(j,i)$ is the fraction of the overall weight of the jth constraint that is assigned to partition i, and $t(i,j)$ is the desired weight of the jth constraint for partition i.

Since our current objective of initially partitioning an RDF graph has no assumption about the workload, the interesting constraint is partitions sizes in terms of their number of vertices. In this context, our constraints vector is the default vector. However, we need to set a suitable value for the maximum acceptable *imbalance* to be set in *options[METIS OPTION UFACTOR]*.

Adaption of Partitions Size Balance

Setting a proper value to the METIS imbalance option (options[METIS OPTION UFACTOR]) is not handled or formulated in all the known related works that used METIS to partition the RDF graph despite its effect on the outcomes of the METIS.

Relaxing this value would speed up the partitioning process as well as increase the accuracy of its main objective that is having minimum cut; although it could produce variation in partitions' sizes.
Since we want to avoid any fixed parameters in our adaptable system, we would like to formalize a method that allows the system to find a proper value of *imbalance*. Relaxing the *imbalance* value is a more favourable choice. This coming from the idea that the host with smaller data size can fill its extra space by replications from the neighbours. Thus, we focus on the first place on performing more optimal partitioning, then solve the problem of extra storage space, by also choosing more optimal parts to replicate.

We let our system adapt itself to the best possible partitions balancing situation that is suitable to the available storage size which could be allocated by each working node for hosting its main data share. The value of this storage space which we denote S_m is dynamically allocated by our storage space optimizer (Section 6.2).
If the total data size is S_d in a system of $|H|$ hosts, the balanced host share would be:

$$\frac{S_d}{|H|}$$

At host i, the difference between the available storage S_{mi} and the host share of the data $\frac{S_d}{|H|}$ is the amount of extra data that the host i can tolerate. Thus, we can tolerate a maximum imbalance per partition P_o as given by:

$$P_o = \min_i (S_{mi} - \frac{S_d}{|H|})$$

However, to avoid creating an extreme case of partitions size variation, we limit the maximum accepted imbalance per partition to half the initial proposed share. In this context, P_o is rewritten as:

$$P_o = \min[\min_i (S_{mi} - \frac{S_d}{|H|}), \frac{S_d}{2 \cdot |H|}] \tag{5.4}$$

The above value of P_o adapts dynamically with the available storage providing the enough flexibility to METIS to produce well partitioned graph while still avoiding the case of extreme size variation.
To map P_o into the METIS imbalance factor and from Equation 5.3, we have:

$$\frac{\frac{S_d}{|H|} + P_o}{\frac{S_d}{|H|}} = \frac{imbalance}{1000} + 1$$

When we solve for *imbalance*, we get:

$$imbalance = 1000 \cdot \frac{P_o \cdot |H|}{S_d} \tag{5.5}$$

Our partitioning system uses Formula 5.5 to find the *imbalance* value which is set to the imbalance input option of the METIS. It should be noted that this value is related to the maximum allowed imbalance in the partitioning; however, the METIS tries to produce more balanced partitioning as long as that doesn't affect the objective of the keeping min-cut across the resulted partitions as been reflected by the inequality in 5.3. The more storage space available at the working nodes, the more flexibility we give to the METIS to favour producing more connected rather than more balanced partitions, although, this could leave some hosts with only small size of data, which can be next utilized by replication as we have mentioned earlier in this subsection.

5.4 Border Region

As described in Section 2.7.3, the graph partitioning process aims to decrease the probability of a query to require data from multiple partitions. As that objective was linked to minimize the number of edges that go between partitions, the border region which contains the vertices where edges come and leave to other partitions, requires special attention [38, 37]. We have given in Section 2.7.3 a description of the methodologies used by related work to deal with this border region and its main drawbacks. We described as well, the main points of our solution to deal with those issues. We go here into more details starting by stating our definition to this border region:

Definition 5.2 (Border Region) *For a partition r_i, we define its border region as $border(i) = \{v \in r_i \mid \exists(v, v_m) \in E : v_m \notin r_i\}$. The border region at partition r_i, with depth δ is defined as follows:*
$border(i, \delta) = \{v \in V \mid v \notin r_i, outdepth(v, i) \leq \delta\}$, where the $outdepth(v, i)$ is the distance between any vertex $v \notin r_i$ and the partition border $border(i)$. V and E are sets of the RDF graph vertices and edges as given by Definition 2.1, and r_i is a partition defined by Definition 5.1.

In the above definition, we stated $border(i)$ as the sharp line of the partition where the edges are leaving from or coming to the partition. However, the more general border region is a function of the depth inside the neighbours' partitions which we denoted as *outdepth*. The sharp border region $border(i)$ is also given by $border(i, 0)$. Our motivation to consider the depth from the border region was explained earlier, and comes from that a SPARQL query has an effective *length* which we defined in 3.3, and if a query has touched the sharp partition border ($border(i, 0)$), part of that

query is already at distance ≥ 0 from this partition and might have a remainder in the neighbour partitions which are also at distance ≥ 0 from those partitions' border.

5.5 Border Replication

In order to decrease the communication cost between graph-based partitions, border replication is used. The benefit value of this replication is given in Section 5.1 as a gain in the access time to replicated data. This gain is related to the network access time as was given in Formula 5.1. In this context, each time a border-replication triple is used, it delivers the same benefit; however, how often this triple is going to be used is shaping the effective benefit of replicating this triple (Formula 3.1). This is related to its access rate that we detect and estimate using two types of rules: general and specific. In the following, we state and define those rules, which enable aggregating them into a single replication rule ready to contribute in the storage universal adaption.

5.5.1 General Border Access Rule

Finding a border general access rule depends on deriving an access formula to the border region, where the border replications are taken. Consider a query q that has length l, the query answer q_a according to Definition 3.2 is the set of all the sub-graphs in the RDF graph G that match the query graph and substitute its variables. This also means that each of those sub-graphs has the same length as q. Assume that some $a \in q_a$ has at least one vertex $v \in border(i)$, where $border(i)$ is the border region of a partition r_i. The worst case for that partition happens when v is a source or sink vertex in q [2], such that we have only $v \in r_i$ and could have all other vertices of a in the other partitions. Assuming uniform access probability, the average location of v would be on the middle of the length path of q. Thus, we could have half of a on the other partition. The effective length of this part is $\frac{l}{2}$. Given that the average queries length is L, we can write the uniform probability of a vertex $v_m \notin r_i$ to contribute in queries answers at partition i as the following:

$$p_{rem}(v,i) = \frac{1}{outdepth(v,i)} \cdot p_{border} \tag{5.6}$$

Where p_{border} is the probability of a query at partition i to access its border region, which is set to 1 at system startup.

[2] A source vertex in DAG is a vertex with no incoming edge, while sink vertex has no outgoing edge

Equation 5.6 represents the general rule of access to the vertices that are located in a remote node hosting another partition and at some given distance. The value of the p_{border} is initially set to 1, but is going to be further updated depending on the workload by counting the rate of accessing the border region by all the executed queries in the system so far. The same method is used to set the average query length.

By having the border region access formula, we are ready to define the general border replication rule in the following definition:

Definition 5.3 (General Border Replication Rule) *For each working node* i*, a general border replication access rule is defined for each outdepth* δ *as:*
$\varpi_{br,ge}(\delta) = (s^{\delta}_{br}, \hat{V}^{\delta}_{br}, a^{\delta}_{br})$*, where* s^{δ}_{br} *is a function that when applied on partitions* r_i *at outdepth* δ *it returns* $\hat{V}^{\delta}_{br}$ *:*
$s_{br}(\delta, r_i) = \hat{V}^{\delta}_{br} = border(i, \delta) - border(i, \delta - 1)$*, and*
$a^{\delta}_{br} = \{(v, a) | v \in \hat{V}^{\delta}_{br}, a = p_{rem}(v, i)\}$*.*

5.5.2 Specific Access Rule

In the previous subsection, we derived a general access formula that is based on the average behaviour of the collected queries. The formula states that for some partition i, the access of some vertex v at some outdepth δ decreases rapidly with the increase of δ. This means that the nearest vertices are more beneficial as border replication, and all the vertices at the same distance have the same importance value. Moreover, the total number of vertices at outdepth δ from partition i increases exponentially with δ, given that on average, each vertex at depth δ is connected to more than one vertex at depth $\delta + 1$. This makes the storage cost of replicating border vertices show the same exponential increase. The rapid-decreasing importance, as well as the exponentially increasing cost will make the fixed replication from the border a weak choice with respect to other choices of our universal adaption model that consider the replication as one choice out of multiple choices. As a result, this could lead to a high increase in communication costs. This problem highly motivates the existence of more specific rules that limit the selection domain to a smaller number of vertices with higher importance. For these specific rules, we look to the workload and detect the border vertices that have more probabilities of access than the probabilities that are only reflected by their *outdepth* value. For that purpose, we use the heat query of Section 3.5, which provides the access probability on the level of a single vertex as was given by Formula 3.5. The heat query access rule can be projected on any part of the data set. We use this property to project it on the border region so that we

can define the specific rule of the border replication as in the following:

Definition 5.4 (Border Replication Specific Rule) *By the projection property, the border replication specific rule is the projection of heat-query specific rule ϖ_{he} on the border region of partition r_i:*
$\varpi_{br,sp}(\chi) = proj_{\varpi_{he}(\chi)}(border(i, L_{max}))$*, where L_{max} is maximum query length recorded in the workload.*

This rule is similar to the index specific rule (Section 4.7.2), where the heat queries rules are projected on the local data in a working node.

5.5.3 Aggregating Border Replication Rules

The access functions of general and specific rules are targeting the vertices in the neighbour nodes. In this context, some of the vertices are targeted by both rules. To deal with this issue, we aggregate both rules into one border replication access rule.

Definition 5.5 (Border-replication Aggregated Rule) *The general and specific access rules are aggregated according to the aggregation property (Section 3.4):*

$$\varpi_{br}(\chi) = aggregate(\varpi_{br,sp}(\chi), \varpi_{br,ge})$$

The access rule is transferred into operation rule for each index in the system:

$$R^{bo}_{op} = \{\varpi_{op} = (\varpi_{br}(\chi), \chi, \Delta) | \forall \chi \in X\}$$

where Δ is the difference in access time between local and remote storage divided by the storage cost for each replicated vertex to the destination index χ.

The above set of operational rules represents border replication. We still have another type of replication that we consider in the next section. Both types will be then aggregated to create one set of rules representing the replications.

5.6 Load-balancing Replication

Besides overcoming the border region problem, replication is also used to enhance the load balancing between the working node to increase the system throughput.

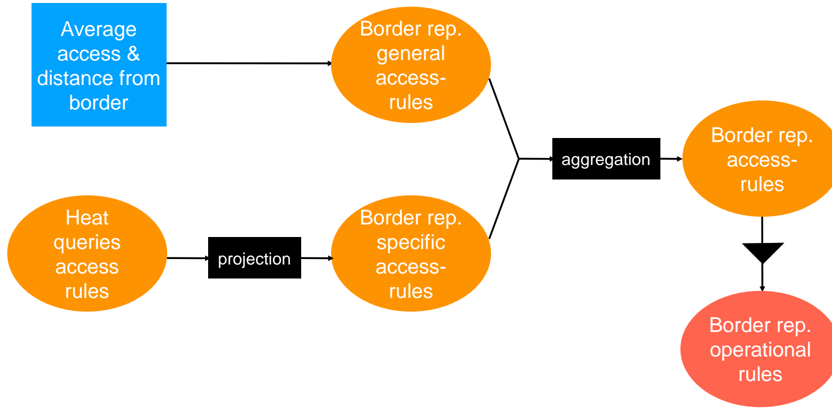

Figure 5.2: The map of replications' rules

5.6.1 Load-balancing Replication in The Cost Model

Recalling our general cost model Section 3.2, the cost of the replication for the sake of load balancing is still the size of the replicated data. The benefit of having such replication is to increase system throughput. For a block of replicated data with a given size, its benefit to a working node is related to the count of queries that it has contributed to in a period where that working node would be idle. More precisely, that performance benefit can be written as:

$$\frac{\textit{size of replicated data}}{\textit{average of query processing size}} \cdot \textit{average query execution time}$$

From Section 4.6.1, the size of replicated data per vertex v is given by:

$$storageCost(v, \chi)$$

From Definition 3.2, the average query size is given by:

$$\frac{\sum_{(q,f)\in Q} q_p}{\sum_{(q,f)\in Q} f}$$

Then we can rewrite the load balancing replication benefit as:

$$\eta_{LB}(v, \chi^{(m,n)}) = \frac{storageCost(v, \chi) \cdot \sum_{(q,f)\in Q} f}{\sum_{(q,f)\ inQ} q_p} \cdot q_e^{avg} \tag{5.7}$$

Where n is the working node where this benefit function is evaluated, v is located at some remote node at the time of the calculation, and q_e^{avg} is the average query execution time.

That benefit is only applicable in the time when this working node would be idle if it does not have this replication data. At such time, the queries queue of that working node is either empty or contains only queries that require the replicated data. That time is the accumulative time in which a working node is idle while other nodes are working. The ratio of that time to the total running time (named as τ) represents the factor of accessing the replicated data. Thus, if we replicate a vertex from node n where it has there an access value of f, then it would have access in this node equal to f multiplied by τ.

$$access_{ba}(v, \chi) = a_{re}(v, \chi) \cdot \tau \tag{5.8}$$

where $a_{re}(v, \chi)$ is the access of the vertex in a remote node as given by the $\varpi_{idx}(\chi) \in R_{as}^{idx}$ the index aggregated rules set given by Definition 4.4.

5.6.2 Load-balancing Replication Rules

In the previous subsection, we formulated the benefit of such replication $\eta_{LB}(v, \chi)$ in Formula 5.7, which is given for each vertex v and destination index χ. We also described the access function as given by Formula 5.8. The access rate is factorized by τ which down to zero at perfect load balancing.

Definition 5.6 (Load-balancing Replication Rules) *We define in the following access and operational rules for the load-balancing replication data:*

- *The load-balancing replication access rule:*

  $R_{as}^{ba} = \{(a_{ba}, \hat{V}_{ba}, s_{ba}) | \forall (a_{idx}, \hat{V}_{idx}, s_{idx}) \in R_{as}^{idx}, a_{ba} = a_{idx} \cdot \tau, s_{ba} = remote(s_{idx})\}$

  where $remote(s_{idx}$ returns only the sources in the remote nodes.

- *The load-balancing replication operational rule is defined as the following:*

  $$R_{op}^{ba} = \{r | \forall \varpi(\chi) \in R_{as}^{ba}, r = (\varpi(\chi), \chi, \Delta), \Delta = \frac{\eta_{LB}(v, \chi)}{size(v, \chi)}, v \in \hat{V}_{ba}\}$$

  where $\eta_{LB}(v, \chi)$ is given by Formula 5.7 and $size(v, \chi)$ is the storage cost of replication v in the local index χ.

5.7 Replication Aggregated Rules

The rules of border replication and load-balancing replication are both targeting vertices that are located in the neighbour working nodes. Since that their vertices intersections are not an empty set, we need to aggregate both of them. However, the aggregation is performed on the operational levels and not on the access level. This is because the benefit function of the border replications differs from the benefit function of the load-balancing rule.

Definition 5.7 (Replication Aggregated Rules) *The operational rules of both border replication and load-balancing replication are aggregated as the following:*

$$\begin{aligned} R_{op}^{rep} =&\{\varpi_{op} | \varpi_{op} = aggregate_{op}(\varpi_{op1}(\chi), \varpi_{op2}(\chi)), \forall \chi \in X, \varpi_{op1}(\chi) \in R_{op}^{bo}, \\ &\varpi_{op2}(\chi) \in R_{op}^{ba}\} \end{aligned}$$

The R_{op}^{rep} represents a set of operational rules the represent the replications or the vertices in the remote nodes from the perspective of a certain working node. That set is comparable with the set of index operational rules and join-cache operational rules that were given in the previous chapter.

5.8 Summary

We summarize the chapter in the following points:

- In a system of distributed working nodes, the RDF graph is partitioned and assigned to the nodes.
- The border regions create a performance problem because the queries in that region could require synchronization across the working nodes.
- That border region problem is overcome with border replication.
- Putting the border replication in the cost model answers the questions: what data to replicate? and how much?
- The border replication has a general access function related to the vertices' distance from the border and defines a general access rule. On the other hand, its specific access function is derived from the heat queries by projection and defines the specific access rule.
- Both of the rules are aggregated into one border access rule. A benefit function is attached to that rule to create the set of border-replication operational rules.

- Another purpose for replication is to perform load balancing aiming to increase the system throughput.
- The access rate to these replications is related to the nodes' load-imbalance factor, and to the access of the data in their source nodes. We derive that access to define an access rule for each index in the system. The operational rule is defined by adding the benefit function which is related to the system throughout.
- The operational rules of the border replication and load-balancing replication are aggregated into one set of operational rules. Those rules are comparable with indexes and join cache operational rules.

Chapter 6

Universal Adaption

The previous two chapters presented separated approaches for adapting the indexes, join cache, and replications with the workload and storage space. The adaption process for each of the three concluded by a set of operational rules that are comparable with each other. This chapter deals with performing the universal adaption of the indexes, join cache, and replications using their derived operational rules.

Contents

6.1 System Architecture 108

6.2 Storage Space Optimizer 109

6.3 Creating The Proposed and Assigned Rules 116

6.4 Summary 117

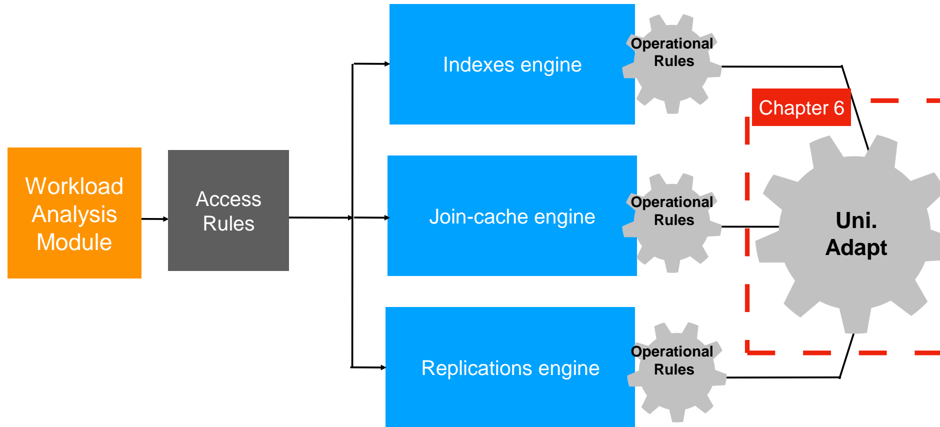

Figure 6.1: Chapter's scope

6.1 System Architecture

We have presented in chapter 2 the design options of a distributed triple store while reviewing the related works of RDF triple stores. We considered the federated shared-nothing nodes, where each node hosts an adopted version of a central triple store. This approach is also followed by [37, 26, 38, 83, 31]. This provides the system with enough flexibility to adapt the indexes and replication layers with the adaption parameters that were previously explained in Chapter 3. The main components and architecture of our adaptable RDF triple store (UniAdapt) is shown in Figure 6.2. Our distributed system H is a set of n hosts. A host h_i can directly send any message to any other host h_j using the underlying network. We refer to the delay accounted by sending a message in the network as:

$$delay = size(msg) * Z * \zeta \tag{6.1}$$

where Z is the network transfer rate, and ζ is a random function representing the size of the current traffic in the network at the time of message sending. Since we assume that the network is dedicated to the purpose of connecting the working hosts, the traffic that is being transferred in the network is essentially the traffic coming from system messages and the data synchronization between the hosts, i.e

moving the intermediate results of running queries across hosts. Each node h works independently from other nodes, such that it receives its own share of the RDF graph. The node has its storage optimizer that builds the node's replication layer by looking into the neighbours. The node runs queries on the available local data (main share plus replications), and returns the result to a selected node that assembles the final query result. Each node also makes its own decision about the type and quantity of replications which are to be built from neighbours' main shares and it has its own optimizer for this purpose. The initial partitioning is made by a single node (node 1 in Figure 6.2), and the results are distributed to other nodes.

Within each working node, there is a main memory query engine, which is supported by a hard-disk query processing engine based on RDF-3X [56]. The storage layer is composed of a dictionary and indexes. The dictionary is a dual hash tables structure that maps each string in the raw data-set to a compressed integer code, and performs a reverse mapping of any integer code back to its original textual representation. By using this dictionary, each textual triple in the data set is converted to an integer triple and stored in the appropriate indexes. The dictionary concept has been used by many works, but [17] was the first to apply it to RDF systems.
The system has a collection of indexes where the RDF data actually reside as was explained earlier in Section 2.4. Each index may contain local or replicated data. The system keeps a record that helps to distinguish the local data from the replication, where this record costs a single bit per triple.

The storage optimizer is responsible of managing the storage layer by collecting and analysing the workload in order to make decisions about what type of data to assign to each index including the join cache. A decision also is made for the size and type of the replications, allowing UniAdapt to show universal adaption behaviour. The storage optimizer and the universal adaption is explained in more detail in the next section.

6.2 Storage Space Optimizer

For a given storage space unit s at a certain working node, the optimizer aims to employ it for lower queries execution time, by either building more indexes, join cache or by building more replications. The choice of building more indexes needs further decisions about the certain vertices to index and the type of such indexes. In the same context, building more replications needs further decision about the certain vertices to replicate and the type of index to maintain them. Chapter 4 considered optimizing the indexes and join cache by formulating their benefits, costs, and access

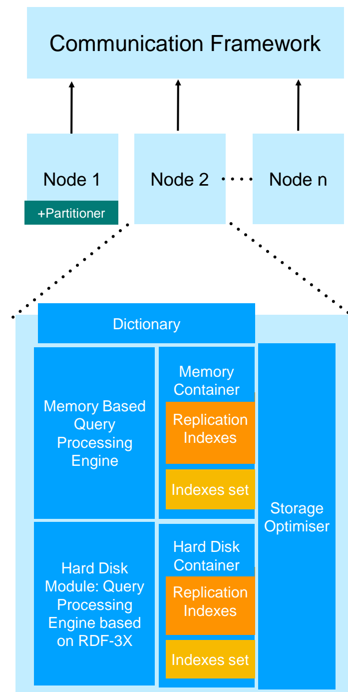

Figure 6.2: Abstract system architecture

rates. Then transferred the formulas into sets of operational rules. The same is done in Chapter 5 which concludes by providing a single set of replication operational rules. The storage optimizer puts the three optimization modules together and unifies the optimization process. The abstract storage optimizer components are shown in Figure 5.1.

6.2.1 Universal Adaption

The universal adaption was already presented in Section 3.2 as the ability of the system to make an optimized decision to employ any of its resources with the best-expected option out of a set of multiple options. The basic of the adaption algorithm is stated in Section A.2.1 and projected on the storage adaption in Figure 6.3 which sketches the process architecture. The workload is collected and analysed into Heat Query graph(s). The system has predefined rules which have formulas to calculate the benefit of the vertices in the RDF graph based on the cost model. A raw data unit in the RDF graph is colored with blue in Figure 6.3. In the storage system, we have several indexes as well as the replication, and each structure is differentiated with its own unique color in Figure 6.3. The cost model of the optimization process

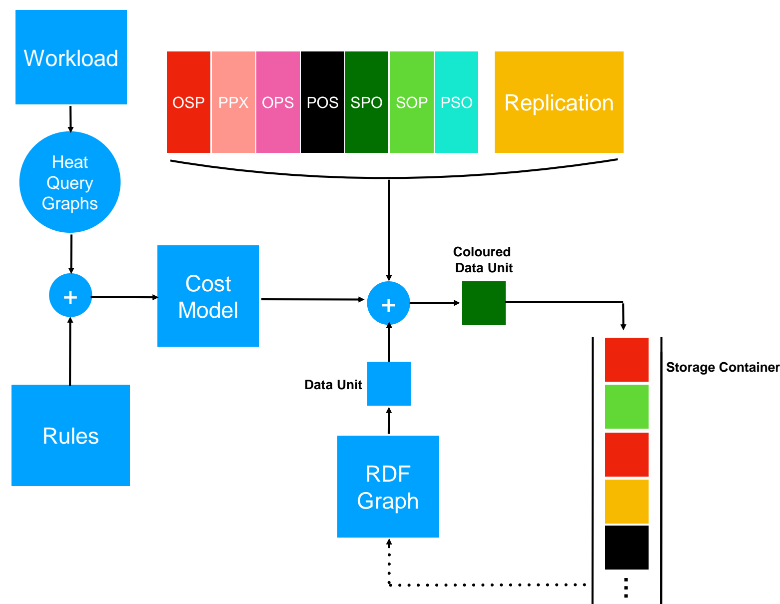

Figure 6.3: The process of storage space adaption

assigns to the blue raw data unit, the best-known structure, and labels it with the selected structure colour to produce a coloured data unit (the dark green in Figure 6.3 is an example indicates that the data unit was assigned to the SPO index). The coloured data unit is stored in the storage container and structured as its selected structure. The storage container is the physical representation of the storage where assigned data units are stored and the system pays its storage cost. This also means that the RDF graph is not presented physically, but it conceptually represents the storage container. This representation is modeled with the dashed line that connects the storage container with the RDF graph. This relation also means that, at some point, a coloured data unit in the storage container is going to be treated as raw data unit, such that it can be evaluated again by the cost model and possibly assigned to another place.

The Basic Storage Adaption Algorithm

We consider now transforming the basic adaption algorithm (given in Appendix A, Section A.2.1) into a more applicable storage adaption algorithm.
The basic work of the initial algorithm is to calculate for each vertex in the data graph the benefit of assigning it to any of the storage options (which are the memory

and hard disk indexes). The considered data graph is the local part of the data within a working node and its neighbours. The above benefit is further factorized by the access rate deduced from the workload. The next step is to sort the vertexes up on their benefits and select the most beneficial triples to be assigned to the best-fit index.

The methodology used to analyze the workload to find the vertices' access rate is given in the previous chapter, Section 3.4 where we derived Formula 3.7 for vertex access rate.

The running time of the above algorithm is expensive since it needs to monitor each vertex, besides the storage cost required to keep track of vertexes' benefits. To overcome these costs, we could reduce the accuracy of the benefits values and aggregate them into limited number of levels(e.g.five levels). This would result typically in a pyramid-shape of values. The lower big base contains the less important vertices, while the top has a small amount of more important vertices at its higher levels. The optimizer may now keep track of only the most important vertices by maintaining a priority queue sorted by the vertices' importance. Such a priority queue may be updated at some point in time when the system has detected obvious changes in the workload. A second priority queue is kept by the optimizer in order to keep track of the vertices that have been already assigned in memory. However, the vertices in this case are the less important vertices out of what exists in memory in terms of their benefit. As a result, the optimizer checks the two priority queues and performs a swap whenever a vertex at the top of the first queue is more important than the vertex at the second priority queue's top. This algorithm is already followed by a previous version of our system [3]. However, the method still requires a considerable amount of storage space besides the difficulty to maintain the queues of vertices. In the next subsection, we explain a more optimal algorithm to handle the storage

space adaption that we have followed in for this version of UniAdapt.

Algorithm 1: Basic space adaption algorithm

```
   input : RDF graph G = {V, E}, the partition number i, indexes set X, and the set of heat
           queries H
 1 G_h ← apply(H, G);
 2 G_h = {V_h, E_h};
 3 for each v ∈ V_h do
 4     for each χ ∈ X do
 5         if v is local then
 6             access ← access(v, χ);
 7             baseBenefit ← η_idx(v, χ);
 8         else
 9             access ← access(v, χ) · p_rem(v, i);
10             baseBenefit ← Δ;
11         end
12         benefit ← baseBenefit · access;
13         U ← U ∪ {(v, χ, benefit)};
14     end
15 end
16 updateQueue(assignedVerticesQueue, U);
17 updateQueue(proposedVerticesQueue, U);
18 while size(proposedVerticesQueue) > 0 do
19     {(v_p, index_p, benefit_p)} ← pop(proposedVerticesQueue);
20     {(v_a, index_a, benefit_a)} ← pop(assignedVerticesQueue);
21     if benefit_p > (benefit_a + BIAS) then
22         swapAssignment((v_a, index_a, benefit_a), (v_p, index_p, benefit_p));
23     else
24         break
25     end
26 end
```

The algorithm runs at each working node i with the RDF graph G, the set of indexes χ, and the set of heat queries H as inputs. The first line applies the heat query to the RDF graph G to get the sub-graph G_h which has the set of vertices V_h with an access rate of more than zero.

The first loop iterates over all the vertices in V_h, then for each index χ, it retrieves the access and benefit of each vertex v. However, these values are calculated differently depending on whether v is a local or remote vertex. In the case of local, the *access* value is set in Line 6 using the heat query access formula given by Formula 3.7. The base benefit is the $\eta_{idx}(v, \chi)$ which is already given by Formula 4.3. In case of that v is a remote vertex, the access returned by the heat query is multiplied by the p_{prem} which is the general access rate to the border replication looking from working node i. We find p_{prem} by recalling Formula 5.6. In Line 10, we set the base benefit of the remote vertex to Δ which is the difference in access time between a local and remote vertex which basically depends on the network access time. The benefit is then calculated according to Formula 3.1 by multiplying the access by the

base benefit. A triple element of vertex, index, and benefit is created and assigned to U at the end of the loop vertices loop's iteration (Line 13).

The next section of the algorithm deals with the assignment and swapping between indexes in the storage element. First, the priority queues are updated with U. As was earlier mentioned, we have two priority queues. The first is $assignedVerticesQueue$ which holds the vertices that have already been assigned in the previous runs of the algorithm, ordered ascendingly by the vertices' benefit. The second queue is $proposedVerticesQueue$ which contains the vertices that are proposed for assignment and ordered descendingly by the benefit. Once the benefit values in each of the queues are updated, the loop of Line 18 may begin to perform the swap operation between the top of each queue. The loop halts when the top of the proposed queue is no longer more beneficial than the top of the assigned queue, or when the proposed queue is empty. The swap procedure also updates the queues accordingly to keep them consistent in terms of storing the assigned and proposed elements.

6.2.2 Better Algorithm: Rules-based Space Adaption Algorithm

The main issue with Algorithm 1 is that it has to scale on the level of all the vertices. To avoid this problem, we make use of the operational rules derived for the indexes, join cache, and replications. Each operational rule includes an access rule, a destination index, and a benefit function. The included access rule has a set of sources and an access function. That are all the needed to calculate the benefits of each vertex represented by the sources set. However, we don't have to project the benefits on the vertices. Instead, we project the benefit function on the sources, and compare the sources instead of comparing the vertices. Since that the number of the rules are limited and so that their sources, we can perform dramatical performance optimization to the adaption algorithm given by Algorithm 1, and use Algorithm 2 instead.

The algorithm works whenever enough workload change has been detected. The new workload Q besides the data set graph G, the index types set X and two sets of rules are the inputs to the algorithm. The first set of rules is called the proposed rules R_p, which are the rules that have their sources representing the data that are ready to be assigned to the working nodes' main memory. In the same context, the assigned rules are representing the data that were already assigned to memory.

The first loop updates both of the two rules sets with the new workload Q using the procedure $updateRulesAccess(r, Q)$. This requires updating the inline heat queries, statistics, and recalculating the benefit formulas. In Line 3, each proposed rule is

going to have its sources sorted descendingly by their benefits, such that we have the most beneficial source element in the head of the rule sources. In the contrast, each assigned rule will have its sources ascendingly sorted by their benefits in order to bring the least beneficial source element to the head of the rule sources. The sources sorting process is carried out using Property 4 of the rules' properties given in Section 3.4. The second loop takes one rule r_p from the proposed rules set, and another rule r_a from the assigned rules set. r_p is the rule that has the best source, while r_a has the worst source in terms of the benefits. These assignments of r_p and r_a are taken place in Line 6 and Line 7 respectively. This operation requires scanning the rules set and comparing the rules by their benefits. Each rule ϖ has its set of sources s. However, since ϖ is sorted, only the source head $\bar{s}$ from each rule is considered in the scan process.

The algorithm halts normally when the benefit of the worst assigned rule is greater or equal than the benefit of the best proposed-rule. However, there are other practical reasons to halt the process not mentioned here, for instance, in case of no more proposed rule is available.

At Line 11, we evaluate the head source of rule r_p to produce the vertices set $\hat{V}_p$. The same is done in Line 12 to evaluate the head source of rule r_a, and produce $\hat{V}_a$. The final step is to swap $\hat{V}_p$ with $\hat{V}_a$ using the procedure $swapAssignment(r_p, r_a)$. The procedure needs access to the full rules in order to get the destination index. Moreover, the procedure marks in r_p the source that has been assigned, and in r_a the source that has been unassigned. This allows the algorithm to work again by keeping track of what has been assigned and what is still proposed.

In order for the algorithm to work as expected, any two rules in R_p need to be aggregated whenever there is an intersection between their set of vertices. This is because the sources of the rules represent the vertices, and we need to have one benefit value per vertex in order to correctly compare them. This is already performed during the build of the operational rules in the previous chapter, and We further discuss this point in Section 6.3.

Running Time

To analyse the running time of the algorithm, we look at its two loops. The first loop has a size of $O(|R|)$. In each iteration, we sort one rule by its sources set s. This costs $O(|s| \log |s|)$ for each rule, which makes the total loop cost within $O(|R| \cdot |s| \log |s|)$. In the second loop, we also iterate over the rules sets, and preform constant work on the head source of each rule. However, the costs of the evaluation parts are

dynamic and depend on the complexity of the source patterns and the availability of the indexes. The cost of the swap procedure is linear with the number of vertices to be swapped by the algorithm, which also depends on the detected changes in the workload which are translated into changes in rules benefits.

Compared to Algorithm 1, we evaluate the vertices in Algorithm 2 at runtime, so that we substantially reduce the high cost of maintaining the benefits on the vertices levels by maintaining the benefits on the rules' sources level. Moreover, the running time of Algorithm 2 is delimited by $|R| \cdot |s|$ which is small number compared to the data graph vertices.

Algorithm 2: Rules-based space adaption algorithm

```
   input : RDF graph G = {V, E}, and two sets of the system operational rules: proposed rules R_p
           and assigned rules R_a
 1 for each r ∈ R_p ∪ R_a do
 2     r ← updateRulesAccess(r, Q);
 3     r ← sortRuleBySource(r);
 4 end
 5 while true do
 6     r_p ← ϖ_op | ϖ_op = (ϖ, χ, Δ), ϖ = (s, V̂, a) : ∀r_i ∈ R_p, [a(s̄) · Δ(s̄)] ≥ [a_i(s̄_i) · Δ_i(s̄_i)];
 7     r_a ← ϖ_op | ϖ_op = (ϖ, χ, Δ), ϖ = (s, V̂, a) : ∀r_i ∈ R_a, [a(s̄) · Δ(s̄)] < [a_i(s̄_i) · Δ_i(s̄_i)];
 8     if b_a ≥ b_p then
 9         break
10     end
11     V̂_p ← evaluate(s̄_p);
12     V̂_a ← evaluate(s̄_a);
13     swapAssignment(r_p, r_a);
14 end
```

6.3 Creating The Proposed and Assigned Rules

Recalling the cost model of Section 3.2.1, the optimization process needs to find the best option out of several options to employ some resource. In the case of a storage optimization process, a working node is optimizing each unit of storage with the best piece of data structured in the optimal index. We presented in Algorithm 2, a rule-based optimization algorithm. However, in order for it to work, we need to provide a set of operational rules that represent the access and benefit distributions of the data set with respect to the workload. This requires that these rules to be sortable by their sources so that we compare and differentiate between the rules without having to store the stats on the level of vertices. For an access rule to be sortable on its sources, the result of applying the access and benefits functions on the source patterns must be equivalent to applying the functions on the individual vertices of the rule given by $\hat{V}$. This requires that each vertex in $\hat{V}$ is identified by only one pattern in the source set of the rule. Assuring that each rule in R_p of Algorithm 2

is sortable will enable determining the maximum source per rule which is called the head source $\hat{s}$. However, we need next to compare $\hat{s}$ of each rule and find the rule with the maximum benefit so that we can evaluate and assign its vertices. This again requires that the same condition which was applied on the single rule's sources, to be applied on the heads of all rules. This means that each vertex in the RDF graph must not be targeted by more than one rule within one working node. To achieve this, the rules need to be netted and aggregated according to the rules' properties given by Section 3.4, such that the intersection of the vertices sets of all the rules in R_p is equal to $\emptyset$. Fortunately, we have already performed these aggregations when building the three sets of operational rules: R_{op}^{idx}, R_{op}^{che}, and R_{op}^{rep}. Those represent respectively: indexes operational rules, join cache operational rules (Chapter 4), and replications operational rules (Chapter 5). Those rules' sets are ready to be added to the proposed rules set R_p of Algorithm 2:

$$R_p \leftarrow R_{op}^{idx} \cup R_{op}^{che} \cup R_{op}^{rep}$$

Besides the set of proposed rules, Algorithm 2 has a set of assigned rules R_a. Those are the rules that track what has been already assigned to the local storage. Thus, the replication operational rules in R_a do not represent the data in the neighbour nodes, but the data that have already been replicated locally. The benefits of those rules are still the same as the proposed rules; however, the access functions differ. Since we already have the data, the assigned rule now measures the real access value instead of the potential access values that are used in deriving the access rules so far. As a result, the assigned rules are copies of the proposed rules except for the access calculation method.

6.4 Summary

This chapter detailed the method of performing a universal adaption of the storage resources on the levels of the indexes, join cache, and replications. That is achieved by comparing their operational rules that were derived in Chapter 4 and Chapter 5. Each operational rule has a set of sources and a benefit function that assigns relative benefits values to the vertices coming from each of the rule's sources. Since these benefits are globally comparable, the adaption algorithm sorts the sources of each rule and always picks the source with the highest benefit for assignment. The storage indexes will always be filled with the best known performing options allowing the universal adaption with respect to the workload and the storage space.

Chapter 7

Universal Adaption Evaluation

In this chapter, we perform systematic practical experiments to evaluate the effects of various adaptation sources on the performance of selected RDF-triples stores. We start by describing the used benchmarks and queries, then we test the scalability of the systems with respect to the data-sets sizes. The core practical evaluation is performed in the Universal Adaption section. We conclude the chapter by presenting our summary and conclusions in the final section.

Contents

7.1 Generation of Data-sets and Queries 120

7.2 Data-set size 120

7.3 Universal Adaption 121

7.4 Summary 131

7.1 Generation of Data-sets and Queries

In order to test and evaluate any RDF triple store, we need to have both of a data-set and a query-set. Different real-world RDF data sets are available like YAGO [36], DBpedia [19], and BTC [35]. BTC contains a collection of data sets like BIO2RDF[5, 21]. On the other hand, there exist also generated data sets like WatDiv [6] and LUBM [62]. While a real data set has usually better acceptance in term of results validation, a generated data set has the privilege of properties tuning flexibility. This flexibility allows better sketching of system behavior with respect to the data-set properties change.
Recalling the modeling of RDF data from Section 2.1.4, any RDF data set is eventually modeled as a graph that has exactly three elements: Vertex, Edge, and Edge's label. As a result, the properties of a data-set is reduced to the property of a graph. The main analysis metric for such a graph is the distribution of the edges density within the regions of the graph. This distribution typically follows a normal distribution [88]. The two factors that draw the normal distribution shape are the mean and standard deviation. We are going to consider those metrics when classifying and analysing a data set.
As a contrast to the data-sets, real-world queries are not publicly available expect for a limited number of queries. However, as we have already pointed out in Section 3.1, different works analysed some existing real-world queries logs and produced its specifications in workbench studies [12, 30, 68]. These specifications can be practically used to produce a workload that is simulating a real-world stream of queries; moreover, such workload generation method provides the flexibility of evaluating the behavior of the query processing system versus tuned parameters of the workload. In this context, to generate testing query set in our evaluation, we implemented our own query streams generator following the properties mentioned in Section 3.3 besides using the standard generator of the WatDiv data-set, and used both to show the performance of UniAdapt as well as the related systems that used for comparison.

7.2 Data-set size

In this section, we focus on evaluating the scalability of the systems with respect to increasing the size of the data-set in terms of the number of triples. The scalability of the system is evaluated from two perspectives: the first is the ability of the system to store and maintain an increasing number of triples, and the second is related to the effect of the increasing size on the queries performance.

7.2.1 System Capacity

Evaluating the adaption of the system with the storage space is an essential objective of this chapter. In this context, we would like to have a numerical measure of the system storage availability. As we have seen above in this section, the performance of a single index is related to the density of the RDF graph or the number of edges per vertex. However, the ability of the system to maintain more indexes and more replication is relative to both the data size and the storage capacity of the system. In this context, we define the storage capacity for a certain storage unit m as:

$$cap(m) = \frac{size(m)}{size(\chi_{full})} \tag{7.1}$$

where $size(\chi_{full})$ is the size in byte of a full index that contains the whole triples in the data set.
Formula 7.1 returns a value indicating how many indexes the system can fully maintain, and this metric is going to be the basic of measuring the system adaptation with space as will be shown latter in Section 7.3.

7.3 Universal Adaption

In this Section, we perform the core evaluation of our adaption system with respect to other systems which implement some level of workload adaption. We first give a real-world starting point, then consider multiple workload scenarios. We then consider the extreme cases of poor workload quality and measure the systems' responses in different levels of storage space.

7.3.1 Starting point

We consider here testing the system with a workload that has real-world parameters, and evaluate the adaption of the systems after the first batch of queries. This first batch that has the size of 1000 queries serves the purpose of training the adaptable system. The evaluation of the system performance is carried out next with several batches.
The first test is performed on the DBpedia data-set. According to [78] and [60], we have the following given points:

- 90% of the queries target 160 frequent patterns.
- 80% of the queries have a length of less than or equal to 2.

- A high access rate is expected at most of the times.
- The number of working nodes is 2 simulating the same distributed environment reported by [78].

Our tests are divided into runs such that each run has its specific parameters. For each run, we execute two batches of queries with the given parameters and evaluate the system adaption for the second batch, while the first batch serves the purpose of training the adaption layer. The adaption is projected on the storage layer and reflected on the query performance. The performance is measured as the total running time of the query batch subtracting the system idle time, which is the time when the system is totally idle. We change the parameters of the starting point to create more workload scenarios. We switch the data set to WatDiv, increase the length of the query up to 4, and increase the number of working nodes to 4.

7.3.2 Adaption Parameters

Form the starting point given above, we change the workload and space parameters and evaluate the adaption of the system again in the dual batch method described in the previous sub-section. However, given that we have a big number of workload parameters besides the hardware parameters, and since considering changing all the parameters would end with a non-feasible exponential number of experimenters, we select a path of change that aims to reflect the adaptation behavior of the system with an average number of experiments. In this context, we give an abstract introduction of those parameters in the following:

- Storage capacity which has been introduced in Section 7.2.1, is the relative ability of a storage unit to maintain RDF triples.
- The indexes used by the system is abbreviated according to the indexes notation Section 2.5.
- The replication is subdivided into two parts: border replication (Rep. B) and load-balancing replication (Rep. L). More details are already given in Section 5.1.
- A query is either unbounded (has no constant in any of its subjects or objects) or bounded on either of its subject or object. More details are given in Section 3.3.2.
- The quality of the workload is labeled α and given according to Section A.2.

Given the above points, we used three levels of storage capacity to measure the behavior of the system in different storage levels. For the first capacity level which is 2.2 we have two groups of runs, the first is against the DBpedia data set where the length of the query was averaged to 2, and the second is against the WatDiv data set where we able to get longer queries averaged around 4.

Run	Capacity	Length	α	Bounded	S-bounded	UniAdapt	Adpart	WARP
0	2.2	2	0.1	0.9	0.8	946	1009	24862
1	2.2	2	0.1	0.9	0.8	471	893	21554
2	2.2	2	0.01	0.9	0.8	411	887	14796
3	2.2	2	0.1	0.9	0.3	480	995	19990
4	2.2	2	0.01	0.9	0.3	381	721	13903

Table 7.1: Parameters of runs 1-4 with systems' running times

Run	SPo	PSo	POs	OPs	SP-o	OP-s	Rep B	Rep L
0	324	0	324	0	0	0	0	0
1	0	0	324	47.2	322.8	1.2	10.7	17.1
2	0	0	324	26.4	304	10.3	20.1	27.4
3	57	324	0	0	2	324	11.2	15.3
4	26.4	324	0	0	9	324	18	20.6

Table 7.2: storage distribution of runs 1-4 (in millions of triples)

Table 7.2 shows the storage distribution in terms of the sizes of the relevant indexes (i.e. the indexes which have size more than zero at some point of time during the runs). At Run zero the system is basically building its knowledge about the workload, and starts with the indexes SPo and POs. This start allows the system to answer any triple pattern that is unbounded or bounded on the subject. The type of workload received by the three systems is seen in Table 7.1; the average of the queries length is 2 and 90% of the queries are bounded, and 80% out of that 90% are bounded on the subject, while the rest are bounded on object. The workload quality (described in Section A.2) of the generated workload is set to 0.1. The system capacity is limited to 2.2, which means that the storage unit can maintain 2 full indexes, besides free size which can be employed to maintain a size of data equivalent to 0.2

of the full index.

In the second batch of the run numbered 1, the optimizer of UniAdapt detects that most of the triple patterns are bounded on the subject and also have the predicate as constant. This led to changing the size of the index SPo to zero and the size of SP-o to 323.8 million triples. The hashed index is faster and covers all the requests of the 80% queries that have subject-bounded patterns in the batch of queries. The remaining object-bounded queries may use either the OPs or the POs for some extra cost. However, the POs can cover also the request from unbounded patterns (have constants only in predicates). Thus, the optimizer decided to set full-size POs, set the size of PSo to zero, and assign 47.2 M triples to OPs. Those triples have been selected by the optimizer as the most relevant triples to OPs. Any other triples that need to be queried by OPs may use the POs index. The optimizer decided at these storage and workload parameters to have 27.8 M triples divided as 10.7 to support the border and 17.1 to support the load-balance between the working nodes. This is also consistent with the relatively small average query length recorder at this run.

In Run 2, we have better workload given that α has a smaller value. The effect is reflected in the storage layer by a decrease in the SP-o indexes, as the optimizer has now a better ability to detect the S-bounded patterns making use of the excellent workload. For the same reason OPs was decreased allowing more space to the OP-s index, since the better workload has enabled the system to better detect the O-bounded patterns.

Run 3 has the same parameters as Run 1 except that the ratio of subject-bounded queries is now 30%. That also means that the object-bounded queries are now representing 70% of the total bounded queries which are still representing 90% of the total queries. With respect to Run 1, the optimizer replaced POs with PSo which have now full index size instead of zero in the previous run. The size of OP-s is turned to full instead of SP-o giving that most of the queries are now object bounded and have the predicate as constant. Run 4 goes with better workload quality with respect to Run 3, and the optimizer was more capable of detecting the 30% subject-bounded queries allowing a more replication space.

Increasing Queries Length

The three runs numbered from 5 to 7 are directed to see the effect of increasing the average length of the query from 2 in the previous group of runs to 4. The storage behavior of the system is shown in Table 7.3, and the workload parameters are given

in Table 7.4.
Despite that, the queries length is doubled in Run 5 with respect to Run 3, the optimizer preferred to assign less space to replication because the increase in length led to more local index processing, and more space is given to the SPo and SP-o indexes. On the other hand, the system uses the good quality of the workload to replicate smaller but more beneficial replication.
Run 6 has basically the same parameters as Run 5 except for a lower workload quality. That made the optimizer assign more space to the border replication, and less space to the SPo and SP-o indexes. Run 7 is also similar to Run 5 except for the distribution of queries bounding type, such that Run 7 is mostly Subject-bounded. The replication, in this case, recorded approximately the same values, while the SPo, PSo and SP-o have changed roles with OPs, POs, and OP-s respectively.

Run	SPo	PSo	POs	OPs	SP-o	OP-s	Rep B	Rep L
5	69.5	324	0	0	24	311	5.49	0
6	57.1	324	0	0	4	324	17.9	0
7	0	0	324	70.1	324	2.7	4.9	7

Table 7.3: Storage distribution of runs 5-7

Run	Capacity	Length	α	Bounded	S-bounded	UniAdapt	Adpart	WARP
5	2.2	4	0.01	0.9	0.3	2741	6414	80047
6	2.2	4	0.1	0.9	0.3	3084	9694	149783
7	2.2	4	0.01	0.9	0.9	2820	9428	107540

Table 7.4: Parameters of runs 5-7 with systems' running times

Increasing the Capacity

In all of the previous runs, the system was put under strict storage space availability. Runs 8 to 11 are meant to measure the response of the system in case of more storage space is available. In this context, the capacity in Table 7.5 is now 4.5 which means that the storage has enough size to maintain more than 4 full indexes. Another workload parameter that has been changed in this group of runs, is the presence of more unbounded queries. An unbound query contains constants only at the levels

of the predicate, thus its processing involves a relatively big number of triples that require multiple rounds of joining. In all of the four runs in Table 7.5, both of the average queries length and the ratio of subject-bounded queries are fixed on 90% and 4 respectively. This is seen by the optimizer as more needs to the SP-o index. Despite that, the overall ratios of the bounded queries are varied between 40% to 70%, the given capacity ratio hints the optimizer to favor maintaining full indexes for POs, OPs, and SP-o during all the runs from 8 to 11.

Run	SPo	PSo	POs	OPs	SP-o	OP-s	PP-x	Rep B	Rep L
8	0	0	160	160	160	56	73	91	24
9	0	0	160	160	160	25	104	101	13
10	0	0	160	160	160	49	160	22	10
11	0	0	160	160	160	22	197	20	4

Table 7.5: Storage distribution of runs 8-11

Run	Capacity	Length	α	Bounded	S-bounded	UniAdapt	AdPart	WARP
8	4.5	4	0.1	0.7	0.9	19112	143531	881370
9	4.5	4	0.01	0.4	0.9	9542	85133	352238
10	4.5	2	0.1	0.4	0.9	3359	4871	48119
11	4.5	2	0.01	0.7	0.9	1012	2436	13533

Table 7.6: Parameters of runs 8-11 with the systems' running times

Performance Evaluation

The final objective of the adaption process is to have better queries execution time. In order to have an overall image regarding the performance of the three approaches in comparison (UniAdapt, AdPart, and WARP), we sketched the execution times of the runs from 0 to 11 in Figure 7.1. The UniAdapt was superior to the other systems in all of the runs. However, there is a general increase in execution times in all the systems in the range Run 5 to Run 9 because of the increase in queries length. Run 8 to Run 11 showed additional higher execution time due to the increase in the ratio of the unbounded queries.

The best relative performance of UniAdapt is seen in Run 8 and Run 9. This is

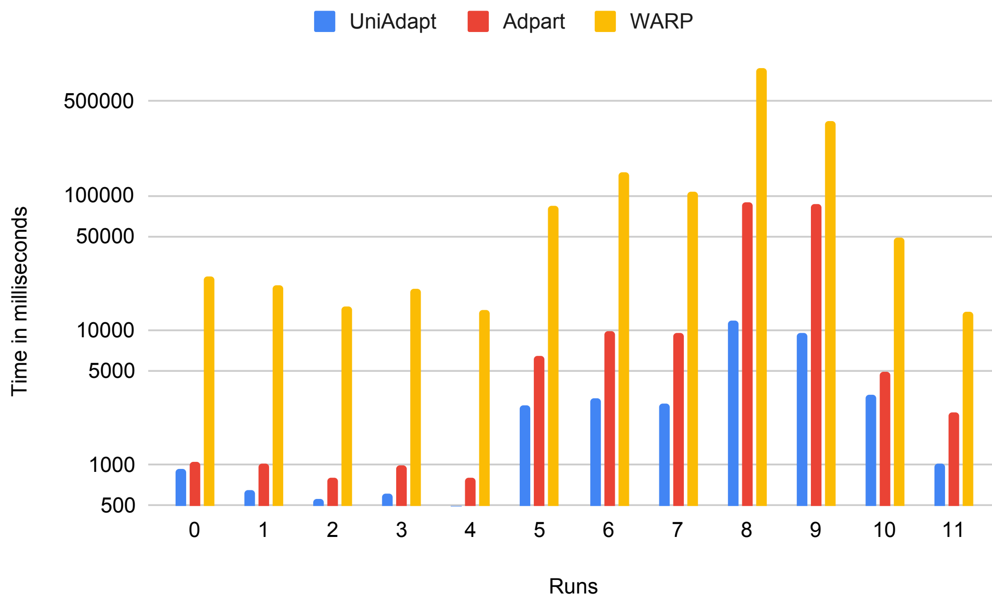

Figure 7.1: The systems' performance comparison of the 12 runs

mainly due to the relative abundance of storage space and the bigger length of the queries. In these circumstances, UniAdapt was able to employ the available storage space to maintain the proper full indexes, relevant border replication beside caching the most important parts in the PP-x index which save expensive joining time. That cache effect was obvious in Run 9 where the workload was of better quality and led to the shown decrease in the execution time. On the other hand, the longer queries length required more expensive communication cost for AdPart, since it partitions the data to the working node by hashing the subject of each triple.
The WARP which is based on RDF-3X [56] is not a native main memory system, thus its execution time pays the cost of hard disk latency. However, WARP performed relatively well in Run 10 and better in Run 11. This due to that these runs have more storage space, which allows the operating system policy that handles the main memory to become more effective. This is more obvious in Run 11 where the workload quality is high, which means that small parts of the data are more frequently accessed.

7.3.3 Non-frequent Workload

In this part of the evaluation, we test the behavior of the system under extreme workload circumstances in which the workload targets the WatDiv data set with

uniform distribution and with no chance of repeating. The heat queries set cannot detect any specific behavior, and thus the specific-rules set has low effectiveness. However, the general rules can still detect the general usage statistics of the indexes, replication, and join cache. The objective of this part is to evaluate the effect of the general rules under these circumstances.
Under the above assumptions, the specific rules of the replication are not active anymore, and the general rule is mainly related to the average usage of the replicated data and its distance from the border. All the data at a certain distance from the border is treated equally. However, the indexes average usages are going to be variant depending on the shape of the queries. The cache indexes are also badly affected by the workload. However, there is still an effect of the storage availability on the indexes cache.
We divide this test into sub-tests where each sub-test is composed of several runs.

Short Heterogeneous Queries

The first subtest has 100% unbounded short queries with length equal to 2, and with full uniform access that has no repetition. Figure 7.2 is showing the behavior of the system with 5 runs of the given workload properties. Each run is composed of 5 batches with 200 queries each. Each run is performed in a specific level of system storage capacity. The first run has a strict capacity of 2. At this capacity level, the system is not going to have enough space or workload knowledge to perform join cache besides extra replication and indexes. However, the short length of the queries decreases the need for the replication besides the heterogeneity of the queries which mainly requires the PSo and OPs indexes. Thus, both UniAdapt and AdPart performed closely in this run. The next two runs increased the storage space and thus enabled the system of having more replication; however, this caused little change on the running time. The system starts to show a considerable decrease in running time starting from a capacity level of 8. That is where the cached join index starts to have enough elements to affect the performance. Although the cache required a lot of storage space, a capacity level of 8 is quite possible in the case of the system is maintaining a small data set. The performance of the AdPart shows the expected steady behavior with the system capacity. We changed the heterogeneity of the queries from pure unbounded to mixed of bounded and unbounded and sketched the results in Figure 7.3. Although this change increases the indexes' needs in query processing, UniAdapt adapts itself with the available storage space with respect to AdPart. At the highest level of capacity, the system achieved all the required

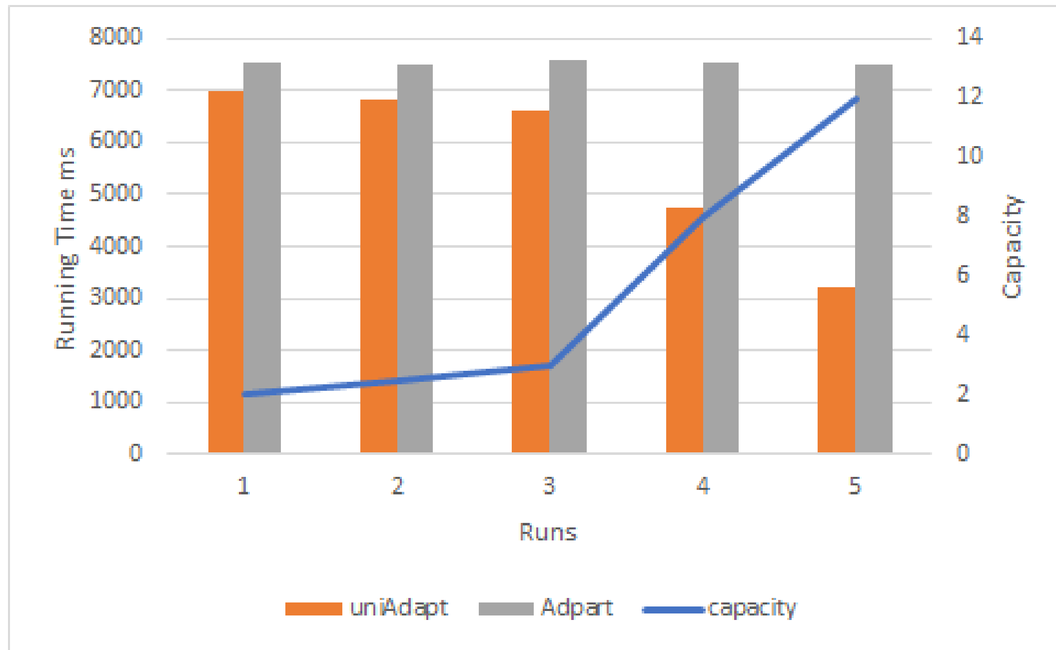

Figure 7.2: Short heterogeneous queries vs capacity

replication and substantially decreases its response time.

Long Heterogeneous Queries

We use the same workload properties of the previous sub-test but with queries length of 5. At this length, the role of the replication is very clear in on the query execution time. The behaviors of the systems are sketched in Figure 7.4 with respect to the system capacity. Both systems show similar performance at the lowest level of system capacity. However, UniAdapt employs the increasing space for more replications to overcome the bad impact of the workload and enhance the performance. At the capacity level of 3.5 and 4, the impact of the cache and the sufficient level of replications appears into a considerable decrease in the queries execution time.

7.3.4 Non-uniform Workload to Partitions Access

We see here another scenario of bad workload trend, when the workload tends to access some parts of the data sets that happen to be in only one partition. By this scenario, we generate 4 partitions using METIS then generate a workload that targets only one of them. The initial effect of this extreme case on the system is a bad load balance state such that 75% of the distributed system resources are not utilized.

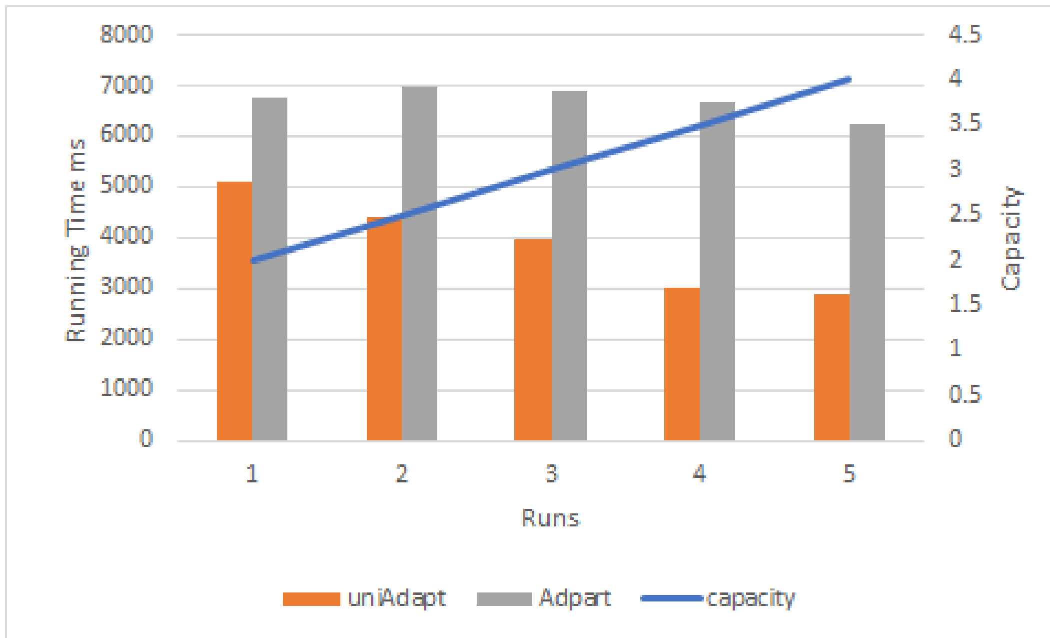

Figure 7.3: Short non-heterogeneous queries vs capacity

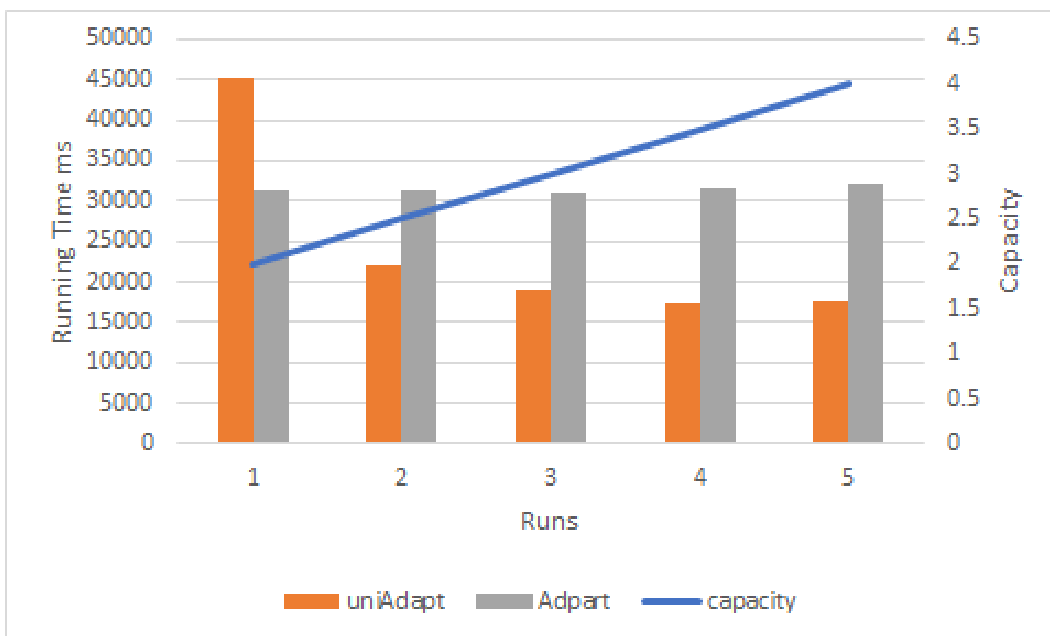

Figure 7.4: Long heterogeneous queries vs capacity

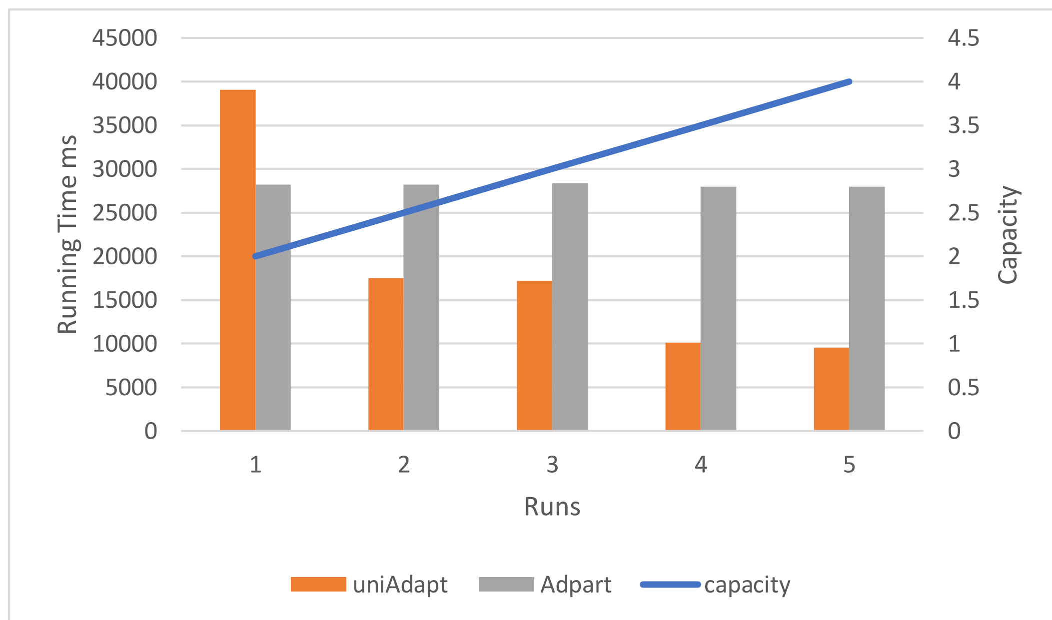

Figure 7.5: Long non-heterogeneous queries vs capacity

The effect on AdPart is limited since it uses a hash-based partitioning strategy. The reaction of UniAdapt to this scenario is to perform replication for the purpose of load balancing. This can be very effective when the locality of the workload is high such that the size of the targeted data is small and thus can be easily replicated. This is seen in Figure 7.6 in runs 4 and 5. In Run 0 the system has not yet made any adaption step towards the non-balanced workload access. The first step is taken in Run 1 and reflected in a decrease in the performance despite the limited storage space (see Table 7.7), because the replication for the purpose of load balancing had been marked as the highest priority in the idle working nodes with respect to other storage consumers, besides the short-queries workload. In Run 2 and Run 3, the high storage space capacity solved the problem by enabling the replication of all the highly accessed data. In Run 4, the small locality ratio is translated into a smaller range of the highly accessed data, which enables the system to easily replicated them even within a limited storage space capacity.

7.4 Summary

This chapter presented a practical evaluation of our universal adaption approach. We provided different types and quality of workload and storage space availability

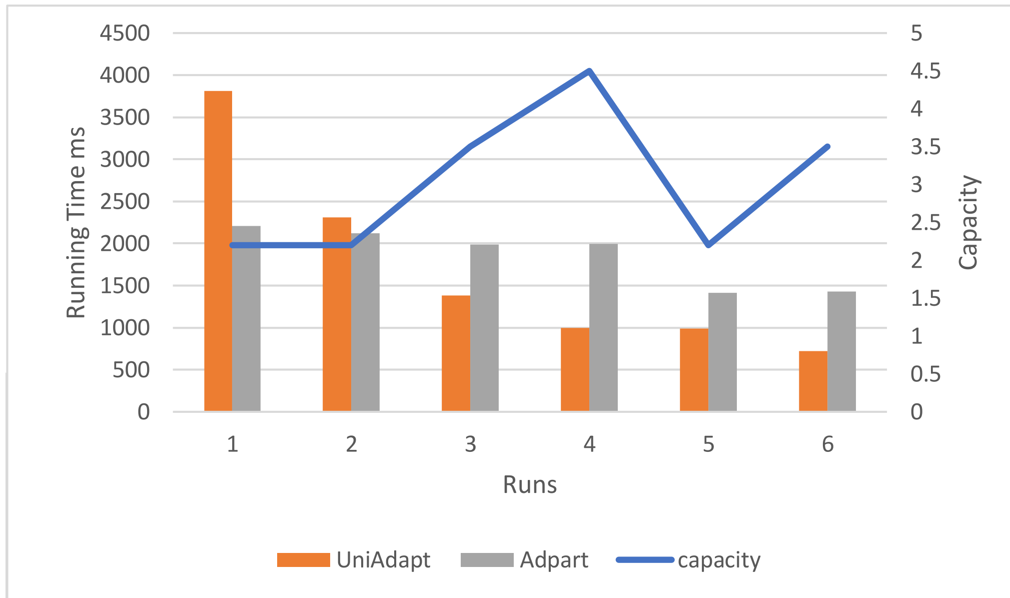

Figure 7.6: The response of the systems towards non-uniform workload access with respect to capacity

Run	Locality Ratio	Capacity	Queries Length
0	0.1	2.2	2
1	0.1	2.2	2
2	0.1	3.5	2
3	0.1	4.5	2
4	0.01	2.2	2
5	0.01	3.5	2

Table 7.7: Workload Properties of The Non-uniform Workload

including real-world scenarios. The UniAdapt was able to utilize its storage resources making use of the workload. This utilization achieved high performance in the cases of high workload quality, high ratio of storage space availability, short queries, and bounded long queries. During the different scenarios, UniAdapt tuned its structures of indexes, replication, and join cache with the detected state of workload and storage space.

To measure the system performance at its limits, we considered the extreme non-frequent workload scenarios, in which the workload is not repeating any of its parts. However, the adaption system detected this low level of workload quality and made

use of its general rules as well as the availability of storage space to boost the performances in most of the cases. The system also responded to the extreme case in which the stream of queries is targeting only one working node. It detected and replicated the hot parts of the data to other nodes allowing them to contribute to the queries execution and released the bottleneck caused by the extreme load-unbalance state.

Chapter 8

Threading

The previous chapters considered mainly the adaption of the storage layer with the workload. The workload contains other types of trends that can be used to reach better utilization of the system processing resources. This chapter presents the methods used by UniAdapt to adapt the processing resources with the queries' arrival rate.

Contents

8.1 Adaption to Queries Arrival Rate 136

8.2 Queries Queuing Model 136

8.3 Adaption of The Processing Resources 139

8.4 Evaluation 139

8.5 Distributed Working Nodes 142

8.6 Summary and Conclusion 146

8.1 Adaption to Queries Arrival Rate

Any modern computer system has a certain level of parallelization capabilities, which is seen by the application as multiple threads. When a query processing system receives a single query, the basic objective is to process this query as fast as possible. Exploiting the threading capabilities of the system speeds up the execution. However, this speedup is usually not linear with the number of the used threads due to threads synchronization costs. If the system receives a stream of queries at a rate that is bigger than its throughput rate, the queries start to build up in the queue. The system would have to choose between assigning the threads to single queries (intra queries parallelism), assign each query to a single thread (inter queries parallelism), or have a combination of the two approaches. This is related to the queries arrival rate and the queuing model of the system,
We have presented in Section 3.1 a real-world example of how rapidly may a queries arrival rate changes over time. In order to show the technical effect of this change on the RDF management system, we will first provide a model of the queries queuing method followed by the system.

8.2 Queries Queuing Model

Our distributed system is composed of n working nodes connected by a network. Each node has the control of its resources, its share of the RDF data, and it has access to other nodes' data. Each node has its own queries queue as well as to the remote nodes' queue; however, we assume that each node has access to any query with the same access time. This makes the system model have one query queue. The query's turnaround time q_{tr} is the time of the waiting in the queue q_w plus its execution time q_e.

$$q_{tr} = q_w + q_e \tag{8.1}$$

Assume that at some point of time, there are k queries in the queue. The average waiting time of the $i'th$ query is given by:

$$i \cdot \frac{q_e^{avg}}{m_a \cdot m_e}$$

where m_a, m_e are respectively the intra and inter query parallelization factors provided by the distributed system. The average waiting time of the k queries is going to be:

$$q_w^{avg} = \frac{\sum_{i=0}^{k}(q_e^{avg} \cdot \frac{i}{m_a \cdot m_e})}{k}$$

$$= \frac{q_e^{avg}}{m_a \cdot m_e \cdot k} \cdot \sum_{i=0}^{k} i$$

$$= q_e^{avg} \cdot \frac{(k+1)}{2m_a \cdot m_2}$$

Then from Equation 8.1, the average turnaround time of the query is given by the summation of the q_e^{avg} and the q_w^{avg} and thus can be stated as:

$$q_{tr}^{avg} = \frac{q_e^{avg}}{m_a} + q_e^{avg} \cdot \frac{(k+1)}{2m_a \cdot m_e}$$

and in a more compact form:

$$q_{tr}^{avg} = q_e^{avg} \cdot (\frac{1}{m_a} + \frac{k+1}{2m_a \cdot m_e}) \tag{8.2}$$

and by letting:

$$f(m_a, m_e) = (\frac{1}{m_a} + \frac{k+1}{2m_a \cdot m_e})$$

then Equation 8.2 is given by:

$$q_{tr}^{avg} = q_e^{avg} \cdot f(m_a, m_e)$$

The average turnaround time that a query would face is related to $f(m_a, m_e)$, and reducing this factor would reduce the average query's turnaround time. The parallelization factors m_a and m_e are related to the processing power of the system, and thus they should sum up to a constant. However, there is the following issue: having more intra-query parallelism is accompanied by paying more threading communication costs. In this context, we can express the relation between m_a and m_e as the following:

$$m_e + m_a + g_1 \cdot m_a + g_2 = c \qquad m_e, m_a, c \geqslant 1 \tag{8.3}$$

where, c is a constant representing the parallel processing power of the system, $g1$ and g_2 are constants representing the synchronizing loss. Assuming that $g_1 = g_2 = y$, we may write Equation 8.3 as:

$$m_e + m_a + y(m_a - 1) = c \qquad m_e, m_a, c \geqslant 1 \tag{8.4}$$

Equation 8.4 means that we can choose to divide our c threads between m_e and m_a, but having more m_a would cost some losses proportional to y, and this loss is

equal to zero when $m_a = 1$. That is the case of the system using only 1 thread for the intra-query parallelism.

In order to minimize q_{tr}^{avg} in Equation 8.1, we need to consider maximizing m_a and/or m_e in $f(m_a, m_e)$. However, maximizing one variable leads to decreasing the other since that both sum to a constant in Equation 8.4. Maximizing m_a gives more benefit for big values, since it appears twice in $f(m_a, m_e)$; however, increasing m_e is more effective on small values, because some portion of m_a would be lost by y factor in 8.4.

In order to have a maximum value of m_a we need to set m_e to 1 in Equation 8.4 then solve for m_a^{max}:

$$m_a^{max} = \frac{c}{y+1} - 1$$

Similarly, we find m_e^{max} by setting $m_a = 1$ in Equation 8.4:

$$m_e^{max} = c - 1$$

Then we are interested in seeing how $f(m_a, m_e)$ is looking by substitute for $m_a = m_a^{max}$ and $m_e = 1$:

$$f(m_a^{max}, 1) = \frac{k+1}{2(\frac{c}{y+1} - 1)} + \frac{1}{\frac{c}{y+1} - 1}$$

$$= \frac{k+3}{2(\frac{c}{y+1} - 1)} \tag{8.5}$$

In the same way we can get $f(1, m_e^{max})$:

$$f(1, m_e^{max}) = \frac{k+1}{2(c-1)} + 1$$

Having $y = 0$ in Equation 8.5 would make $f(m_a^{max}, 1) < f(1, m_e^{max})$ for any value of k with any reasonable value of c. However, increasing the value of y would decrease the difference, then at some value of $y = y_{cr}$, we would have $f(m_a^{max}, 1) = f(1, m_e^{max})$, and we would have $f(m_a^{max}, 1) > f(1, m_e^{max})$ for any $y > y_{cr}$. In order to find y_{cr} we need to solve the following equation for y:

$$f(m_a^{max}, 1) - f(1, m_e^{max}) = 0$$

Doing the basic math and substituting we can have the following:

$$y_{cr} = \frac{c}{\grave{c}} \cdot \frac{\grave{k} + 2}{\grave{k} + 4 + \frac{\grave{k}}{\grave{c}}} \tag{8.6}$$

Where, $\grave{k} = (k+1)$ and $\grave{c} = (c-1)$.
Taking into consideration that $\frac{c}{\grave{c}} \approx 1$, the most affecting factor in y_{cr} is the ratio $\frac{\grave{k}}{\grave{c}}$, which is the ratio of the queue length to the system processing power. If this ratio is high, then we would have smaller y_{cr} and this means that a very small threads synchronization cost y would be still greater than y_{cr} and would cause $f(m_a^{max}, 1) > f(1, m_e^{max})$ so that having $f(1, m_e^{max})$ (maximum enter-queries parallelism and minimum intra-query parallelism) is more beneficial to the system.

8.3 Adaption of The Processing Resources

The processing resources in each working node may all contribute to the processing of a single query. However, any extra thread to process a query requires extra costs expended in the form of threads and data synchronization. If the threads working on a single query are from different working nodes, the system has to pay extra network communication cost. To avoid this type of latency the system tries to keep the execution of each query within a single working node as long as the required data are available locally. In the range of a single query, we still can use more than one thread to execute a single query. The optimizer has the task of deciding the optimal number of such threads. We have already gone through this problem in Section 8.1, and showed that the number of the threads to process a single query is related to the number of queries waiting into the queries' queue. By using Formula 8.6, the optimizer in any working node can easily estimate the number of threads to assign to each query by looking to the number of waiting queries in the queue, as well as the average query execution time, the average thread synchronization cost, and the available number of hardware threads in the system. Generally, the optimizer favors consuming one thread per query as long as the query arrival rate is greater than the system throughput.

8.4 Evaluation

In this section, we provide a practical evaluation to the adaption of the threading and processing resources.

8.4.1 Working Threads

We practically follow the performance behavior of a query with respect to its working threads. In a distributed environment, there are two types of working-threads that might be involved in a single-query execution:local threads which are parallel threads of the working node where the query is being executed, and remote threads which are owned by remote nodes but still handling part of this query. We consider in this section the effect of local threads on the query execution time.
Instead of using the query execution time as a measure of the number of working threads, we use the ratio of execution speedup when using n threads with respect to run the same query with one thread. Having more threads should speedup the query execution by a factor that is ideally the number of the threads; however, this speedup is smaller in the practical world due to the existing of threads scheduling and synchronization costs.
As any typical parallel-processing problem, the important factor in achieving high parallelization speedup is the ratio of the threads maintaining cost to the query processing time. Since there is a correlation between the query type and its execution time, we consider in this evaluation the threading behavior with respect to the query types. Figure 8.1 shows the general behavior of bounded queries on three types: star, tree, and chain. The star query has only one central vertex, and all the the other vertices must have exactly one edge to it. The tree query has the shape of a connected directed acyclic graph (DAG). Finally the chain query is also a DAG, but has one source and one destination. The formal definitions of those types are given in Appendix A, Section A.1.
The three types are compared with respect to the ideal speedup behavior which is equal to the number of working threads. A clear deviation for the three types are observable from the second thread and stop delivering any clear benefit to the tree-query speedup starting from the third thread. Moreover, increasing the threads was harmful starting from the third thread for the star-query, and the fourth thread for the chain-type. The behavior of the queries types correlates with the number of processed triples during their executions as given by Table 8.1.
The unbounded queries draw different behavior shown in Figure 8.2 where we have also the three types of queries with respect to the ideal behavior. The deviation from the ideal is very small at two threads for the three types and slightly starts increasing from the third thread. The differentiation between the three types becomes significant from the sixth thread, but the three types scaled till the seventh threads with a speedup of 4 in the star query and up to 6 for the tree query. The behavior

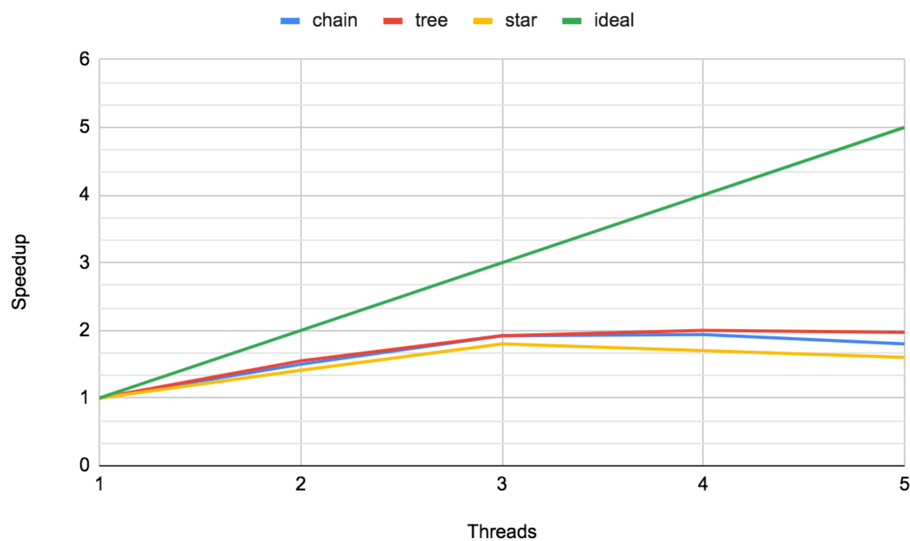

Figure 8.1: Speedup of bounded-queries execution with respect to working threads

difference between the bounded and unbounded query can be explained by recalling their difference in execution and processing in Section 3.3.2. In a general memory-based execution of an unbounded query, the first index call returns a set of triples of size n. The execution goes next by effectively executing n bounded sub-queries in a totally independent way, and requires no synchronization between them except a simple union operation on their results in order to form the final query result. This clearly boosts the speedup of an unbounded query parallel execution and allows better scaling with the used number of threads. On the other hand, a bounded query is typically smaller in size and bounded to at least one vertex in the RDF graph.

Threads	Star	Chain	Tree
1	1	1	1
2	1.41	1.5	1.55
3	1.6	1.92	1.92
4	1.7	1.94	2
5	1.6	1.8	1.97
Processed triples	120	259	412

Table 8.1: Bounded-queries speedup with respect to working threads

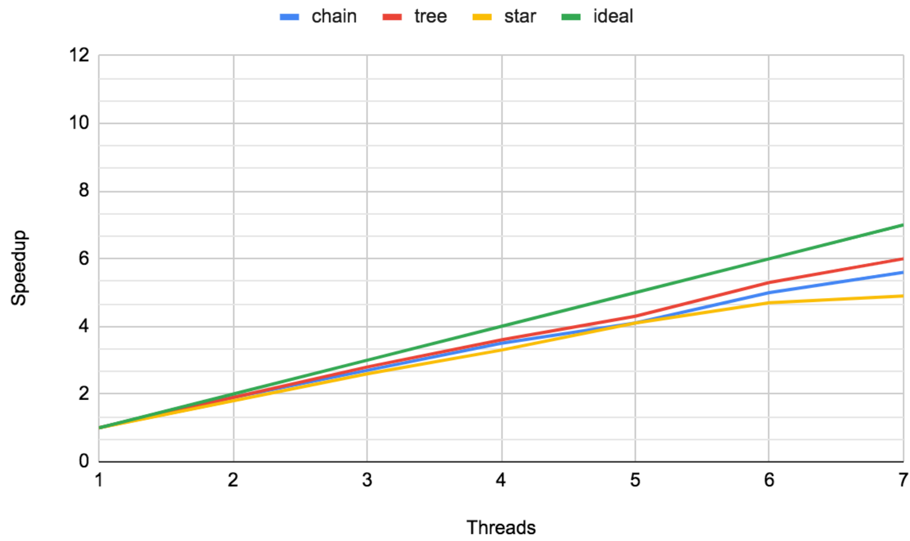

Figure 8.2: Speedup of unbounded-queries execution with respect to working threads

Threads	Star	Chain	Tree
1	1	1	1
2	1.8	1.9	1.9
3	2.6	2.7	2.8
4	3.3	3.5	3.6
5	4.1	4.1	4.3
6	4.7	5	5.3
7	4.9	5.6	6
Triples processed	2124	6235	5412

Table 8.2: Unbounded-queries speedup with respect to working threads

8.5 Distributed Working Nodes

In the previous section, we tested the behavior of the queries performance with respect to the number of local threads processing each query in parallel. In this section, we are going to see the effect of having distributed working nodes that can assign their local threads to process a single query. Having local parallel threads working on a single query requires paying the cost of threads synchronization and the cost of threads initialization. However, having distributed threads working on the same query requires a further cost which is the communication cost, which is the cost required to move the intermediate results across the network.

From the previous section, we find that scaling a typical bounded query with several local threads is practically not an easy task due to the relatively small number of processed triples. Thus, the distributed query processing is only applicable to the unbounded query. This is also followed by AdPart [31] and TriAD [28]. Another important factor to consider regarding the use of distributed threads is data availability. The data required to process a single query might not be available on a single node but on n nodes instead, due to the used partitioning strategy. The system in this case has no option but to use at least n multiple distributed threads to process the query. On the other hand, if the data of one query is only available in a single node due to the partitioning and the lack of enough replication, the system has no option but to process it in that single node using its local threads resources.

Our objective now is to evaluate the effect of the distributed processing on the scale of a single query as well as its relation to the number of triples that are required to be shuffled across the network, the amount of replication, and the load balancing between the working nodes. The effect is measured with respect to the total speedup that is resulted from using the local and distributed threads to process a given query. In this context, we have selected an unbounded query that provides ideal conditions for parallel and distributed processing. The query processes 12448 triples but produces only 2 sets of triples. For testing purpose, we aggressively biased the partitioning of the RDF data to evenly assigns them to the 4 working nodes creating a 100% load balancing between working nodes when they process this query. This 100% balanced rate is kept for runs 1 to 6 in Table 8.3 and Table 8.4. The length of the query is 4 and the number of the working node is set to 4.

In Run 1, there are no border triples which requires no intermediate results to be shuffled across the network. Instead, only 2 triples are shared and unified to produce the final query result. This is not affected by the 0% replication that is set for this run. The speedup scored the highest value rated 15 out of 16 threads. The perfect balancing allows each node to process equal and big share of triples, and since the resulted triples are very limited, they cost a very small time to finally transfer them between the nodes. However, there is a small chance for these ideal circumstances to be practically available all together in the real world. As a contrast from Run 1, the query in Run 2 touches the border area and requires 1106 border triples, which need to be shuffled across the network to finish the query execution. This dramatically decreases the speedup to rate 7 out of 16. This is due to the delay of moving the triples then join them with the local intermediate results. Run 3 requires 2017 border triples which sinks the speedup to rate 4.1 out of 16.

Table 8.4 lists the values of runs 5 to 4, where we switched the 0% replication to 100%. This also means that any node has access to all the data locally, and thus requires no border triples to be shuffled from neighbouring nodes. Some nodes had to do more work since they process the replicated data besides their own share of data. Nevertheless, the effect was negligible on the recorded speedup which rated to 15 out of 16.

Run	Result	Processed	Border	Shuffled	Balance	Replication	Speedup
1	2	12448	0	2	100%	0%	$\frac{15}{16}$
2	2	12448	1106	1106	100%	0%	$\frac{7}{16}$
3	2	12448	2017	2017	100%	0%	$\frac{4.1}{16}$

Table 8.3: The speedup with respect to border triples

Run	Result	Processed	Border	Shuffled	Balance	**Replication**	Speedup
4	2	12448	0	2	100%	**100%**	$\frac{15}{16}$
5	2	12448	1106	2	100%	**100%**	$\frac{15}{16}$
6	2	12448	2017	2	100%	**100%**	$\frac{15}{16}$

Table 8.4: The speedup with respect to border triples with full replication

8.5.1 Queries Stream

In the previous subsection, we tested the distributed query processing under special circumstances and with an ideal circumstances. In this part, we consider more real-world related cases, where the system is expected to receive many queries, and a queuing model similar to what was presented in Section 8.2 is modeling the queries' arrival trend. To test how the system responds to such a stream of queries, we generated 16 random queries and ran them in four rounds of runs. In the first run, the whole 16 queries are run in one single batch and sent to the system together. In the second run, we have two batches such that each batch is composed of 8 queries, and the bathes are executed on the system sequentially. In the same manner, the third, fourth, and fifth runs include 4, 8, and 16 batches respectively.

In Figure 8.3, We sketched the performance comparison of UniAdapt which performs graph-based partitioning using METIS [45], and AdPart [31] which has a partitioning strategy that is based on a hash-based algorithm. The graph-based algorithm

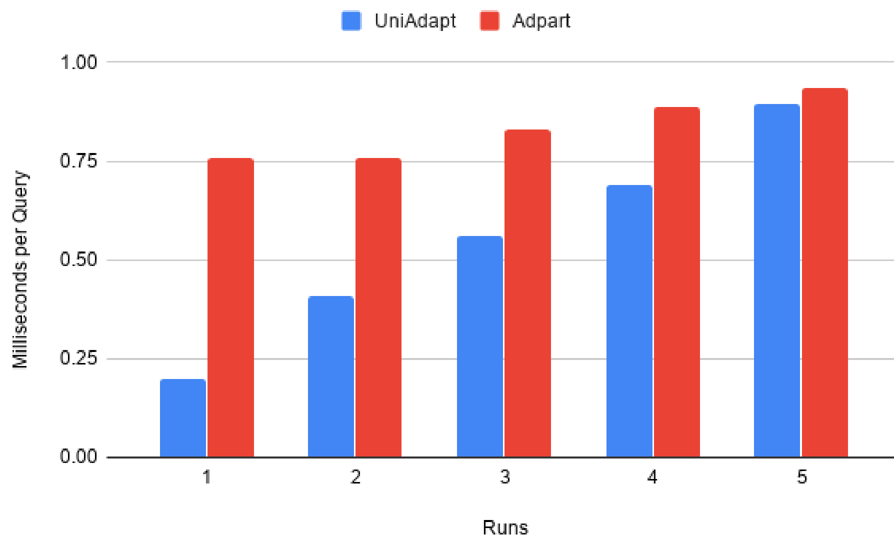

Figure 8.3: The systems' performance comparison of queries stream

Run	Batches	Q. per Batch
1	1	16
2	2	8
3	4	4
4	8	2
5	16	1

Table 8.5: The query streams runs specifications

aims to keep the execution of each query within a local partition for the sake of reducing the communication cost, while the hash partitioning aims to quickly distribute the data to working nodes, which at the same time, supports better-distributed processing for each query[1].

As per Table 8.5, Run 5 has 16 batches with 16 queries each, and each batch is executed separately. In this situation, each system should try to best serve each query. However, the results of Run 5 (Figure 8.3) show that AdPart and UniAdapt are close in their results which means that the communication cost paid by AdPart is similar to the latency paid by the local execution of UniAdapt. However, as we move to Run 4 where each batch is composed of 2 queries, UniAdapt is scaling better since it can better utilize the local processing resources with the second query of each batch.

[1]More details about partitioning are given in Section 5.3

The semi-linear trend of improvement continues when moving down to Run 3,2 and 1. At Run 1, all of the 16 queries are executed in one batch allowing the system to best utilize its local processing resources since it has now more queries to assign to its working nodes. Moreover, if a node happens to be waiting for some border triples to arrive from a remote node, having more queries in the queue allows the node to utilize the waiting time by running the next queries. AdPart gradually makes use of this privilege to decrease the effect of the communication cost as seen in Figure 8.3. These practical evaluations clearly show that having a stream of queries with a high arrival rate highly supports the direction of the local execution of the queries. By reviewing a real world scenario [78], the high arrival rate of SPARQL queries is the expected trend, and the system throughput is the point of performance bottleneck. These evaluations' runs focus on the initial lifetime of the systems where no adaption steps are taken. However, AdPart adaptation steps are by performing workload-based replication to support the local execution and avoid the communication cost.

8.6 Summary and Conclusion

To achieve a high distributed speedup on the level of a single query, we need to have a high number of processed triples and a proportionally very small number of shuffled triples. The number of shuffled triples is inversely proportional to the average length of intermediate results, as was seen by Run 1 where the border triples were 0, implying that the length of the intermediate results was equal to the length of the query itself. In such a case, the shuffled triples are not further joined but rather collected to produce the query result. However, the high value of distributed speedup requires also a load balancing between the working nodes, such that we have equal shares of processed triples done by each node. This is artificially achieved in Run 1 by biasing the partitioning. Unfortunately, achieving the goal of load balancing practically contradicts the goal of reducing the border triples, and the system should choose one direction to follow; either aiming to process each query locally to reduce shuffled triples, or to focus on having better load balancing between the working nodes and deal with the resulted extra shuffled triples.

Chapter 9

Conclusion and Future Work

The final chapter of this thesis summarizes its outcomes and contributions, discusses its points of strength as well as its weak points, and states future outlooks for its development.

Contents

9.1 Points of Strength 148

9.2 Limitations/Points of Weakness 149

9.3 Future Works 151

9.4 Summary 152

9.1 Points of Strength

We discuss in the following the points of strength in this thesis.

- **Boosting the performance**. The RDF triple store that implements the proposed universal adaption approach expects the outcome of a performance boost. That performance increase is the final output that is delivered to the applications world. The triple store becomes adaptable to both the workload and the storage space by utilizing both of them for pushing the performance objective.

- **Workload analysis**. While the workload detection methods used by the UniAdapt and the related works share the same basic methodology of global queries graph, our heat queries are more advanced by requiring no fixed thresholds and automatically tuning its effectiveness. That enables them of avoiding the impact of low-quality workload. In such a case, the system makes use of general average measures. Those general measures besides the specific measures (which are based on the heat queries) are transformed into rules which are compared on a single domain by the optimization process. This method makes the workload adaption process very immune to the drop and fluctuation in workload quality as was shown by the practical evaluation Chapter 7.

- **Storage adaption.** The thesis formulates a unified cost model that calculates the benefits of the indexes, replications, as well as the join cache. The benefits as well as the access rates given by the workload analysis are used to define operation rules for the indexes, replications, and the join cache. Since those rules are comparable with each other, any working node can utilize its storage resources with the best option to optimize the performance. That results in adaption on two dimensions: the workload and space availability. That level of adaption is a unique contribution of this thesis.

- **Queries stream.** UniAdapt extends its storage adaption operations to the adaption of its processing resources. It adapts its local processing resources to the rate at which the queries arrive to the system. In case of a high coming rate, the system aims towards better throughput by avoiding synchronization cost. Otherwise, the system aims towards better query execution time. The objective in both cases is better performance.

9.2 Limitations/Points of Weakness

We discuss in the following points the main overheads or points of limitations in the methods presented by this thesis.

9.2.1 Overheads

As a rule of thumb, a dynamic operation comes with extra cost and overheads with respect to the corresponding static operation. In our universal adaption there are the following overheads:

- **Query runtime overhead**. It is the time spent when evaluating the query using dynamic structures with respect to the use of static structures. In UniAdapt this the time spent to check if the required data exist in indexes or in the cache. In a static system, it is well known in advance whether certain data exists in a given index or not. This is because the index is either fully built or not built at all. However, in UniAdapt the cost of such a check operation is performed in the hashed part of the indexes. Moreover, the count of checks is delimited by the query size which is a very limited value with respect to the number of the processed triples for the query evaluation. The same is applied for checking the cache and replications. This makes the total runtime overheads remove to in average to a constant time per query, assuming the query size is constant.

- **Storage overheads**. That is the amount of storage space paid by the system on storing the statistics and workload analysis structures. Using the rule-based algorithm (Section 6.2.2), the system needs only to store the statistic on the level of heat queries that represent the workload and not on the level of the data set vertices. As a result, the storage overheads of the universal adaption are minimal and much smaller than the space saved or utilized by the adaption operation, when compared to the fixed allocations of structures. That is for example the space wasted on having 6 indexes in RDF-3X while the workload circumstances might need only three indexes.

- **Adaption operation overhead**. That is the time spent on running the adaption algorithm (see Section 6.2.2) and moving the data between the local and remote structures. The Algorithm is optimized to run on the scale of the workload heat queries which is small in size with respect to the total data set. Moreover, the algorithm considers the most relevant parts of the heat queries

and stop once an equilibrium status is reached. Nevertheless, the running time of the adaption algorithm is not trivial with respect to the queries execution time. However, we assume that the triple store would eventually have some free time slots that can be accumulated and used to perform adaption operations which will be beneficial to the query executions and system throughput. For this reason, the query execution time is the metric we used to evaluate the impact of the universal adaption operation.

9.2.2 Worst Cases Scenarios

The variations to the use of the universal adaption are either the use of fixed static allocation of storage structures or the isolated partial adaption. In the related works, the allocations in the first approach are based on the hard observations of certain workload trends in some queries sample. Those observations are simulated by the general rules of UniAdapt. The general rules have the advantage of being evaluated dynamically and thus tuned according to the current state of the collected workload till the moment of the adaption process. The cases where hard observations could perform better are those cases where the specific workload trends are not detectable. The case that affects the specific rules. UniAdapt detects these cases and shuts down its specific rules while activating its general rules. As a result, the worst-case of UniAdapt performs better than the average performance of the hard-setting systems. That is also supported by the practical evaluation (Chapter 6) in the cases where workload quality drops to low levels.
The second variation is to follow a partial adaption on the replication and adopt hard-setting on the indexes and cache. This approach might perform better than universal adaption in the case of that its workload analysis is better, and only the replication matters for the queries' performance. However, the workload analysis method of universal adaption is an advanced version of the global queries graph which is the basic method used by the related works. Moreover, in order for the partial adaption of replication to work it needs the existence of detectable trends within the workload. However, the universal adaption uses those trends to further support the performance with cache and indexes. Thus, the worst-case scenarios of the universal adaption are still better than the worst cases of the partial adaption approaches, under the assumption of using the same data partitioning technique. There are some limitations coming from using the graph partition approach that are discussed in the next section.

9.2.3 Partitioning Limitations

The most observable limitation of UniAdapt comes from the limitation of the graph-based partitioning. The system does not perform a dynamic change of the partitioning type from graph-based to hash-based. Instead, the system supports its graph-based partitioning with replications that serve the purpose of load balancing and decreasing the communication cost. That strategy performs very well in most of the possible workload and storage space scenarios except for the case of limited storage space and low workload quality that is composed of short unbounded queries arriving with low arriving rate such that the queries queue is empty most of the time, as well as a very fast network connecting the working nodes. In those circumstances, a hash-based partitioning system might be able to serve the queries in a higher parallel speedup. Similar conditions are shown in Section 7.3.3 except for the network speed part. However, having all of these parameters at the same time is not common. In fact, the more common case is to have a detectable and repeated trend in the workload. That enables the UniAdapt to overcome the partitioning parallelization issues. Moreover, the more concerning case about server performance is the high queries arriving rate in which queries are building up in the queue. In that case, the system's throughput is the more important value to support rater than the single queries distributed executions.

Another limitation of graph-based partitioning is that it takes a longer time to finish with respect to hash-based partitioning. However, it is a one-time operation that is performed at the system startup (and whenever new data is added). Thus it has no impact on the operational phase of the system.

9.3 Future Works

9.3.1 Partitioning

When UniAdapt starts, it performs graph-based partitioning that is based on METIS. The system collects and builds its workload in the next stage. The workload knowledge is used to support the partitioning with workload-aware replication. However, the partitioning itself is not changed. This can be extended by allowing the partitioning to adapt itself with the new status of the workload. That can be achieved by performing a full rerun of the METIS considering the current workload. Since that can take considerable time, a method that works on the partitions border might be more preferable. A baseline of this method is given by [71]. The system can enhance its initial partitioning without paying any extra storage cost.

The METIS partitioning time which is identified as one limitation in the previous section can be overcome by performing lazy partitioning in which the system starts by assigning the data to the working nodes following the order of the raw data-set, then lazily preforms graph partitioning at each node. An exchange phase comes next to reach an acceptable status of global graph partitioning. Other METIS enhancement methods like distributed graph partitioners [49, 74] are orthogonal to our work.

9.3.2 Workload Analysis

As a future work to our workload analysis method, we propose to consider next the temporal effect of the workload. The workload trends have temporal effect [12] and these effects can be in the short or long terms. The short-term trends are detected in queries logs of DBpedia. These trends are explained by [12] as users' behavior in which they issue several consecutive queries. At each query, they change some variables based on the results of the previous query. These short-term trends should be separated by the workload analysis. The long term trends have the strongest impact on queries performance. More analysis is needed to measure a proper time window out of which a certain trend is considered old and should be pruned out. This would make the workload structures keep the newest trends which are still in effect, and forget the old trends that are no longer in effect.

9.3.3 Optimization's Overheads

In Section 9.2.1, we identified the overheads that are associated with the optimization operation. They were classified into three types: Query runtime overhead, storage overhead, and the adaption operation overheads. Although those overheads are generally small, we propose to include the query runtime and storage overheads in the adaption process itself. That can be achieved by defining a general operational rule such that the benefit of the adaption process is compared against its cost. The costs of the process are the identified overheads. That rule would help the optimizer to decrease the accuracy of the optimization process in the cases where the overheads went too high.

9.4 Summary

Chapter 2 presented the background of RDF as data model. It reviewed the main methods of storing and indexing the RDF triples in central and distributed triple

stores. The query processing was considered and the role of indexes was detailed to motivate the dynamic indexes and its adaption in the next chapters. We moved towards the distributed triple stores and the related data partitioning problem, with the requirements to perform data replications. We then focused our review on the related works that considered adaption with the workload at the levels of partitioning and replications. We showed the open issues with those works and the lack of a universal adaption approach.

Chapter 3 dealt with building the cost model, the workload analysis, and the concepts of the access and operational rules. The cost model divided the storage space into resources, consumers, and options. It defines the optimizer objective to try finding the best-performing assignments. The effective benefit of each assignment is related to its rate of usage by the query processor and its relative performance gain. The rate of usage was derived from the workload on two levels: general and specific. Both of them are mapped into access rules that are dynamically evaluated over the workload, and produce dynamic access rates to data parts. The concept of the heat query was presented where we showed that heat queries represent the heart of each access rule to the indexes, replications, and join cache.

Chapter 4 derived the operational rules of the indexes and provided an evaluation of the dynamic indexes approach. The query execution and the role of indexes were revisited to formulate the benefits of triples indexing. For each index, we derived a specific access rule by projecting the heat queries rule, and a general rule based on the average usage for each index. The two access rules were aggregated into one access rule per index. We then used the benefit formula to derive an operation rule for each index. The same methodology was used to derive an operational rule for the join cache. That operational rule is comparable with the operational rules that were derived for each index. The chapter concluded by providing a practical evaluation of the dynamic index adaption as a stand-alone adaption operation.

Chapter 5 provided distributed storage and data partitioning. The problem resulting from partitioning is overcome with replications. In this context, the system required two types: border replication and load balancing replication. We integrated both types into the cost model by defining the access rule for each type, which is transferred into operational rule by deriving the benefit functions. The two operational rules are aggregated to create a single set of replication operational rules. That operational rules are also comparable with the previous operational rules about the indexes and join cache.

Chapter 6 put together the operational rules of the indexes, join cache, and the

replications into one universal adaptation process. By which the storage is filled with the best assignments in light of the workload. When the workload queries are collected, the formulas embedded in the access rules are updated, and new benefits in the operational rules are calculated, and the most beneficial options replace the worst-performing options. Optimization techniques to the rules-based algorithm were adopted to keep it scalable and avoid causing high storage overheads.

Chapter 7 provided the practical evaluation of the universal adaption given in Chapter 6. We created different workload and space availability scenarios and measured the systems' performances. We compared our system with two adaptable systems: AdPart and WARP. In each run, a new collection of workload parameters were created. Moreover, we used three levels of storage capacity to test the systems' abilities to adapt their storage structures. In different levels of space availability and different workload parameters, our system was the best in adapting its resources and showed superior performance in most of the cases. To test the system performance under extreme circumstances, we generated a workload that contains no detectable frequent patterns. However, our system relied in theses cases on its general rules that are based on the average measurements and avoided the lack of specific frequent patterns. That allowed the system to keep its lead in almost all cases.

Chapter 8 provided an approach to optimize the number of working threads per query with respect to the queries arriving rate. When the rate is too high and the queries start to build up in the queue, the number of threads per query is minimized and the system focuses on the throughput. Otherwise, the system focuses on serving the query as fast as possible. A practical evaluation is also provided to show the effect of the processing adaption on the query execution.

Appendix A

Basic Theoretical Foundations

Some of the theoretical foundations that are related to the workload, queries shapes, the basic of the adaption algorithm and finally the access specifications to indexes in main memory and hard disk.

A.1 Queries Shape

As we have shown in Section 2.6, a query can be modeled as a graph that contains variables and constants. The evaluation of the query is to find sub-graphs in the RDF graph, such that they match the query graph and substitute its variables. However, as we have also detailed in Section 2.6, the system may need different types of indexes to run the query, and there is a strong relationship between the query-graph shape and the types of indexes needed to efficiently evaluate it. In order to state clearly this relation, we present first the general types of query graphs which are mainly found in users queries log [12]:

A.1.1 Star Queries

The star queries have the simplest graph-shape in terms of its execution complexity. The shape has one central vertex called v_r, and multiple vertices that have one direct edge to the central node in any direction. That can be formally defined of in the following:

Definition A.1 (Star Query) *A query q is considered star query, if and only if its graph $q_G = \{q_V, q_E\}$ satisfies the following two properties*[1]*:*

- $\exists! v_r \in q_V : \exists(v_r, v) \in q_E \lor \exists(v, v_r) \in q_E$*; and*

[1] The conditions are written according to the set-theory symbols.

- $\forall (v_1, v_2) \in q_E, v_1 = v_r \vee v_2 = v_r$.

A.1.2 Chain Queries

The chain query chains its triple patterns in a single dimension. Each triple pattern within the query graph, except the start and end pattern, is connected with exactly one pattern from the left and another pattern from the right. The patterns that are located at the graph's start and end, are connected with only one other pattern. The chain query can be formally defined as in the following:

Definition A.2 (Chain query) *A query q is considered chain query, if and only if its graph $q_G = \{q_V, q_E\}$ satisfies the following three properties:*

- $\exists ! v_{start} \in q_V \mid \exists ! v \in q_V : (v_{start}, v) \in q_E$;
- $\exists ! v_{end} \in q_V \mid \exists ! v \in q_V : (v, v_{end}) \in q_E$;
- $\forall v \in q_v, v \neq v_{start}, v \neq v_{end}, \exists ! v_1 \in q_V : (v, v_1) \in q_E, \exists ! v_2 \in q_V : (v_2, v) \in q_E$.

A.1.3 Tree Queries

Tree query combines both chain and star shape, by allowing each vertex in its graph to have more than two edges, but the graph should have no cycle. We can state its official definition as the following:

Definition A.3 (Tree query) *A query q is considered tree query if and only if its graph q_G is a DAG (Directed Acyclic Graph).*

DAG is well known and defined by the graph theory [18], and can be detected in linear time by Depth First Search (DFS) algorithm.

A.1.4 Cyclic Queries

As a contrast to a tree query, a cyclic query contains at least one cycle and thus it is not a DAG. In this context, any query that is not a tree query is a cyclic query, given that the query is connected graph (Definition 3.2).

A.1.5 Queries Length, Size, and Evaluation Size

As per Definition 3.3, the query's length is the maximum distance that can be found between any two vertices in its graph. Depending on the query shape, a query may have different lengths. The star query has the exact length of 1, while the

chain query may easily grow in length reflecting an increase in complexity. The tree and cyclic queries may also have different lengths. The length measurement has a very important effect on query execution. A long query can be evaluated more efficiently if the RDF data are stored and structured as a graph; however, if the data is partitioned also as a graph in a distributed environment, a long query evaluation may require moving intermediate results across the distributed cluster, as will be more explained in Section 5.1. A long query would cause more performance degradation if the partitioning is done on the level of single vertices.

Although the number of vertices in a query graph is limited and expected to be small, the analyzing process is expected to analyze a high number of queries. Thus, an inefficient algorithm to find the query graph could cause a performance issue. Fortunately, we can find the query length according to Definition 3.3 in linear time, since it can be mapped to a Graph Diameter problem, and can be found by running the Breadth-First Search (BFS) algorithm one time on the undirected graph version $\hat{q_G}$ of the query graph q_G starting from any vertex and ending by marking the vertex with maximum BFS value. We rerun the BFS again starting from the marked vertex. The maximum distance recorded by the second BFS round is the query length q_l. This algorithm directly applies to any query of types: star, chain, tree, and cyclic.

The query size is the count of vertices in its query graph, while the query evaluation size is the total number of vertices that have been processed during its evaluation; and finally, the query result size is the count of vertices in the query's answer set as given by definition 3.2. We state these parameters as in the following:

Definition A.4 (The size of the query, its evaluation, and its result) *The size of a query q is given by the cardinality of its vertices set: $|q_V|$. The size of its answer q_a is given by cardinality of its answer: $|q_a|$. The size of its evaluation is total number of vertices processed by its evaluation.*

A.2 Workload Quality

The methods used by the system to analyse a workload are affected by the detectability of the frequent patterns. Their impact is also related to the quality of the workload itself. In this context, we present two numerical values that are used to shape the quality of a given workload. The first is the workload locality ratio which represents the extent to which a workload tends to target certain parts of the data-set. The second is the quality ratio which represents the probability of a query to be part of the previously collected workload. These values will be used to classify

the workload in the practical evaluation that takes place in Chapter 7.

Definition A.5 (Workload Quality Ratio) *For a given RDF graph G and workload w that has access $p_w(v)$ to each $v \in G$ follows a normal distribution, we refer to the workload quality ratio as the standard deviation of the given access probability function p_w.*

Definition A.6 (Workload Frequency Ratio) *The ratio of queries that have frequency more than 1 at the time of adding to the workload with respect to the total number of queries.*

A.2.1 The Basic of The Adaption Algorithm

In sections (3.2.1 and 3.2.2), we presented the corners of the cost model used to optimize the system resources. We discuss here the naive method of generating all the options with their benefits, then performing the optimization reduction.
If R is a set of all the resources required for optimization, and C is a set of all related consumers, we can get based on Formula 3.1 the following set of candidate options:

$$candOpt = \{(r, c, op_i, b_i) \forall r \in R, \forall c \in C, \forall op_i \in op(r, c), b_i = benefit(op_i)\}$$

Given that the system has the method and the implementation to calculate the benefit of each option by having the absolute benefit $\eta(op_i)$ and the access rate $\rho(op_i)$ given in Formula 3.1.
The optimization process can reduce *candOpt* to the following reduced options set:

$$redOpt = \{(r, c, op_m) \forall r \in R, \forall c \in C, op_m = bestOption(op(r, c))\}$$

where $bestOption(op(r, c))$ is a function returns the option with maximum benefit value in $op(r, c)$. If there is more than one option with equal maximum benefit, the function chooses the one with the lowest order in the set.
Unfortunately, the size of $|candOpt|$ is considerably big and equal to the product of $|R| \cdot |C|$ multiplied by the number of options available for each $r \in R$ and $c \in C$. This results in a non-feasible reduction process to generate *redOpt*.
The above problem can be overcome by assuming that all of the resources are of equal importance. For instance, if we are optimizing the main memory, all the bytes are of equal importance to the system. Thus, we don't have to include the resources

in *candOpt*. Instead, we consider only the consumers and their options, and apply the reduction in the same way on *candOpt* to produce *redOpt*, which can now be written as:

$$redOpt = \{(c, op_m)\forall c \in C, op_m = max(op(r, c))\}$$

In order to assign the consumers in *redOpt* to resources in R, we need first to sort *redOpt* descendingly by $benefit(op_m)$ and assign them sequentially to the available resources. However, the size of $|candOpt|$ without the resources is still big since it contains all the consumers whose size is proportional to the data size in the storage model. Handling, sorting, and reducing of *candOpt* is still a costly operation, especially when we take into consideration the necessity to update the access rate and accordingly the benefit of all the consumers when the workload changes. Fortunately, we present a more optimal method to perform the optimization process in the next chapter where we dramatically decrease the required processing steps.

A.3 Index on Hard Disk

Any index can exist in main memory, secondary storage, or in a remote node. The time required to access an indexed element is greatly affected by the hierarchical location of the index. The index access time, in its turn, affects the query execution plans.

A.3.1 Access Time

Accessing a block of data on the hard disk has two delay factors: the average delay time required to randomly access a single block, and the serial access time that is the average time required to transfer the data serially starting from a certain block. The first delay is the summation of several sub-delays, where the biggest comes from what is called the seek time. That is the time required to move the disk's actuator's arm to the target track on the disk where the block of data is residing. Thus the access time of b serial blocks on a typical hard disk can be written as:

$$accessTime(b) = randomDelay^{HD} + \frac{b}{transferRate^{HD}} \tag{A.1}$$

After consuming *randomDelay* seconds, the hard disk requires $1/transferRate$ seconds for transferring each block. Assuming that $\hat{b}$ is the number of blocks that can be transferred serially within the *randomDelay* seconds. We can rewrite the disk access time as:

$$accessTime(b) = \frac{\hat{b} + b}{transferRate}$$

The above formula means that we would pay a cost of reading $\hat{b}$ blocks whatever is the amount of the required blocks b, and if $b = \hat{b}$ then this cost is 50% of the total delay cost. Since that typically, the random delay is relatively much bigger than the serial transfer delay per block, $\hat{b}$ is not a trivial number. In our system architecture Section 6.1, each triple has a fixed size of bytes because of the use of a dictionary that allocates 32 bits for each element. Thus, a triple has the size of 3×32 bits, or 12 bytes. Based on one measurements, a hard disk has a delay of 14.2 ms to access any random block on the disk, and 0.08 ms per block for the following serial blocks. The initial delay, in this case, is equivalent to the time of transferring 714 KB or 178 blocks giving that the size of a block in the file system is 4 KB. Considering the size of a triple that we fixed earlier, the initial delay required to access any triple on disk is equivalent to the time required to access 59K triples afterwards. That clearly states that the costs of accessing random elements in indexes stored on disk are approximately constants and equal to the initial value of the *randomDelay* stated in Formula A.1, and practically not related to the size of the returned data. Based on this, we may fix the index access time $\alpha(\chi^{HD})$ for a hard disk index χ^{HD} as:

$$\alpha(\chi^{HD}) \approx randomDelay$$

In the data structure level, a typical hard disk index in a classical database storage system uses the B-tree or B^+-tree, since they can simulate within their leafs the blocking structure of a typical file system. Similarly, an RDF index uses the same data structure to build its indexes. For example, RDF3X [56] uses B^+-tree in all of its six indexes.

A.4 Triples in Main Memory

In contrast to the hard disk, the main memory not only much faster but also provides a random access possibility with no big penalty on performance. However, accessing serial elements in memory (e.g. scanning an array) is still faster than accessing the same elements randomly. This is due to the optimization provided by the hardware prefetcher as well as the CPU cache which makes the CPU fetcher read memory on the level of blocks. That draws similar conceptual behaviour to the hard disk which is at a lower level in the storage hierarchy. The prefetcher is sometimes fast enough such that it makes the access time effectively zero, when a thread is accessing a serial

big-enough array residing in memory. This is because the prefetcher could be able to fetch faster than the CPU computations. However, the smaller the size of the serial data to be read, the less effective the prefetcher is. In our hardware which is used in the evaluation of Chapter 7, reading a serial block of 80 bytes decreases the reading cost per byte to 50% with respect to pure random access. The reading cost per element decreases rapidly with increasing the block size and becomes effectively zero for any block size that is bigger than 1400 bytes. The 80 bytes can hold approximately 7 triples, while the 1400 bytes can be mapped to approximately 112 triples. This should give an insight into the access to indexes in memory. The exact behavior of the memory access time depends on the type and properties of the used hardware as well as the level of the optimization facilities available in the compiler of the used programming language.

Given the fact the main memory is the place where the processing is taking place, and since it is generally smaller in size, it is considered a very precious resource, and has special attention in our optimization process.

Appendix B

Mathematical Symbols

We append indexes of the main mathematical symbols that have been used throughout the thesis.

B.1 Mathematical Symbols Used in Chapter 3

Symbol	Description
r	resource unit.
c	consumer unit.
$op(r, c)$	function that returns the options to utilize resource unit r with consumer c.
$\eta(op)$	performance benefit to the system of having option op.
$\rho(op)$	probability of access of option op.
$benefit(op)$	the result benefit of option op.
C	set of consumers.
R	set of resources.
G	RDF data graph.
V	set of vertices in the data set.
E	set of edges in the data set.
P	of all the edges' labels in the RDF data.
p_e	the property associated with edge $e \in E$.

D	the RDF data set.
t	triple pattern.
$match(t, d)$	function that returns 1 when triple d matches triple pattern t, or returns 0 otherwise.
q	a query.
q_G	a query's graph of query q.
q_V	vertices set of query q.
q_E	edges set of query q.
q_a	answer of query q.
q_l	length of query q.
Q^t	workload collected up to time t.
$Q(t_1, t_2)$	referrers to the workload collected in the time period $[t_1, t2)$.
ϖ	an access rule.
s	source pattern of rule ϖ.
$\hat{V}$	a set of vertices $\hat{V} \subseteq V$ defined by pattern source s of rule ϖ.
a	a function that assigns a relative access value to each $v \in \hat{V}$ of rule ϖ.
ϖ_{op}	an operational rule.
Δ	the performance gain function of operational rule ϖ_{op}.
q_{lm}	average queries length in workload Q.
h_q	a heat query.
H	heat queries set.
H_j	set of heat join maps.
P_j	set of predicates of heat join maps' set.
$\grave{h}$	anonymized version of heat query h.
$effect(\grave{h})$	effect of anonymized heat query $\grave{h}$.
$heat(h)$	a function that returns for the heat query h the summation of all of its heat values.
$access(v)$	the rate of access of v as expected by the anonymized heat queries set.

$H_q(v)$	a function that returns for any $v \in V$ the heat query that v is associated with, or null if v does not belong to any heat query.
$freq(h, v)$	is the frequency of v as given by the anonymized heat query h.
$access(v, \chi)$	the $access(v)$ for the index χ.
$access(p_1, p_2)$	the access function of p_1 and p_2 in the set of heat join maps.
$p_w(v)$	the probability of access for vertex v by a workload w.
$\varpi_{he}(\chi)$	heat query access rule for index χ.

B.2 Mathematical Symbols Used in Chapter 4

Symbol	Description
χ	an index type.
$indexLookup(\chi, key)$	perform a lookup operation in index *chi* for the given key.
$indexMatch(\chi, key, d)$	returns 1 if the triple d matches the triple pattern represented by the key.
$getOptimalIndex(key)$	returns the optimal index to the given key.
$key(v)$	a function that maps a vertex v to a triple pattern that is consistent with the index type value χ.
$cost(v, \chi)$	the storage overhead of indexing v in index χ given in number of triples.
$\eta_{idx}(v, \chi)$	absolute benefit function of assigning v to χ .
$\varpi_{ge}(\chi)$	the general access rule for index χ.
R_{sp}^{idx}	the set of specific indexes rules.
R_{as}^{idx}	set of aggregated index access rules.
R_{op}^{idx}	set of index operational rules.
$size(t_1, t_2)$	the storage cost of storing (t_1, t_2) in a cache index.
$\varpi_{che,op}$	cache-index operational Rule.
R_{op}^{che}	set of cache-index operational rule.
$\varpi_{che,sp}$	the cache index specific access rule.

B.3 Mathematical Symbols Used in Chapter 5

Symbol	Description
h_i	a working node
H	set of working nodes
Z	Network transfer rate
ζ	a random function representing the size of the current traffic in the network at the time of message sending.
$metis(v)$	function returns the partition which vertex v belongs to.
r_i	certain data partition as sub-graph of the global data graph
$imbalance(j)$	the maximum allowed load imbalance among the partitions for element j of the METIS constraints vector.
S_m	storage space allocated for the main partitioning share in a given host.
S_d	total storage in a given host.
τ	load imbalance factor for the current working node.
P_o	maximum imbalance per partition.
$border(i)$	function returns the vertices in in partition r_i that have at lest one edge to another partition.
$outdepth(v, i)$	function returns the distance between any vertex $v \notin r_i$ and the partition border $border(i)$.
q_l	the length of query q in number of hops.
L	average queries length in the system
p_{border}	the probability of a query at certain partition to access its border region.
$p_{rem}(v, i)$	probability of a vertex $v_m \notin r_i$ to contribute in queries answers at partition i.
δ	certain outdepth.
$\varpi_{br,ge}(\delta)$	general border- replication rule at outdepth δ at certain partition.
$\varpi_{br,sp}$	specific border replication rule.
ϖ_{ba}^{pr}	proposed load-balancing replication access rule.
ϖ_{ba}^{as}	assigned load-balancing replication access rule.
$\varpi_{ge}(\chi)$	general access rule for index χ.

$\varpi_{idx,sp}(\chi)$	specific access rule for index χ.
$\varpi_{che,sp}$	cache-index Specific Rule.
$\varpi_{che,op}$	cache-index Operational Rule.
ϖ_{br}	unified border-replication access rule.
$\varpi_{r,op}(\chi)$	border-replication operation rule of index χ.
R^{he}	set of all heat query specific rules.
$R_{op}^{ba,pr}$	set of proposed load-balancing replication operation rules.
$R_{op}^{ba,as}$	set of assigned load-balancing replication operation rules.
R_{sp}^{idx}	set of index specific access rules.
$R_{sp,op}^{idx}$	set of index specific operational rules.
R_{op}^{bo}	the set of border-replication operational rules.
R_{op}^{rep}	the set of replication operational rules.
R_{op}^{idx}	the index set of operational rules.
R_a	set of assigned system operational rules.
R_p	set of proposed operational rules.

Bibliography

[1] Jans Aasman and Parsa Mirhaji. Knowledge graph solutions in healthcare for improved clinical outcomes. In *International Semantic Web Conference (P&D/Industry/BlueSky)*, volume 2180 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2018.

[2] Daniel J. Abadi, Adam Marcus, Samuel R. Madden, and Kate Hollenbach. Scalable semantic web data management using vertical partitioning. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 411–422. VLDB Endowment, 2007.

[3] Ahmed Al-Ghezi and Lena Wiese. UniAdapt: universal adaption of replication and indexes in distributed RDF triples stores. In *SBD@SIGMOD*, pages 2:1–2:6. ACM, 2019.

[4] Sofia Alexaki, Vassilis Christophides, Gregory Karvounarakis, Dimitris Plexousakis, and Karsten Tolle. The ICS-FORTH RDFSuite: Managing voluminous RDF description bases. In *SemWeb*, volume 40 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2001.

[5] Peter Ansell Alison Callahan, Jose Cruz-Toledo and Michel Dumontier. BIO2RDF: Linked data for the life sciences. `https://download.bio2rdf.org/files/release/3/release.html`.

[6] Günes Aluç, Olaf Hartig, M. Tamer Özsu, and Khuzaima Daudjee. Diversified stress testing of RDF data management systems. In *International Semantic Web Conference (1)*, volume 8796 of *Lecture Notes in Computer Science*, pages 197–212. Springer, 2014.

[7] Amazon. Amazon simpleDB. `http://aws.amazon.com/en/simpledb/`, 2012.

[8] Ali Assi, Hamid Mcheick, and Wajdi Dhifli. Data linking over RDF knowledge graphs: A survey. *Concurr. Comput. Pract. Exp.*, 32(19), 2020.

[9] David Beckett. The design and implementation of the redland RDF application framework. *Comput. Networks*, 39(5):577–588, 2002.

[10] David Beckett and Ivan Herman. RDF primer - turtle version. `https://www.w3.org/2007/02/turtle/primer/#L1995`, 2007.

[11] Tim Berners-Lee. Linked data. `https://www.w3.org/DesignIssues/LinkedData.html`, 2006.

[12] Angela Bonifati, Wim Martens, and Thomas Timm. An analytical study of large SPARQL query logs. *Proc. VLDB Endow.*, 11(2):149–161, 2017.

[13] Dan Brickley and R.V. Guha. W3C recommendation 25 February 2014. `http://www.w3.org/TR/rdf-schema/`.

[14] Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF schema. In *International Semantic Web Conference*, volume 2342 of *Lecture Notes in Computer Science*, pages 54–68. Springer, 2002.

[15] Francesca Bugiotti, Jesús Camacho-Rodríguez, François Goasdoué, Zoi Kaoudi, Ioana Manolescu, and Stamatis Zampetakis. SPARQL query processing in the cloud. In *Linked Data Management*, pages 165–192. Chapman and Hall/CRC, 2014.

[16] Stefano Ceri, Mauro Negri, and Giuseppe Pelagatti. Horizontal data partitioning in database design. In *SIGMOD Conference*, pages 128–136. ACM Press, 1982.

[17] Eugene Inseok Chong, Souripriya Das, George Eadon, and Jagannathan Srinivasan. An efficient SQL-based RDF querying scheme. In *VLDB*, pages 1216–1227. ACM, 2005.

[18] Sanjoy Dasgupta, Christos H. Papadimitriou, and Umesh V. Vazirani. *Algorithms*. McGraw-Hill, 2008.

[19] DBpedia. DBpedia version 2016-04. `http://dbpedia.org/`.

[20] Alex DeJong, Radmila Bord, Will Dowling, Rinke Hoekstra, Ryan Moquin, Charlie O, Mevan Samarasinghe, Paul Snyder, Craig E. Stanley Jr., Anna Tordai, Michael Trefry, and Paul Groth. Elsevier's healthcare knowledge graph and the case for enterprise level linked data standards. In *International Semantic Web Conference (P&D/Industry/BlueSky)*, volume 2180 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2018.

[21] Michel Dumontier, Alison Callahan, Jose Cruz-Toledo, Peter Ansell, Vincent Emonet, François Belleau, and Arnaud Droit. Bio2RDF release 3: A larger, more connected network of linked data for the life sciences. In *International Semantic Web Conference*, volume 1272 of *CEUR Workshop Proceedings*, pages 401–404. CEUR-WS.org, 2014.

[22] Fredo Erxleben, Michael Günther, Markus Krötzsch, Julian Mendez, and Denny Vrandecic. Introducing wikidata to the linked data web. In *International Semantic Web Conference (1)*, volume 8796 of *Lecture Notes in Computer Science*, pages 50–65. Springer, 2014.

[23] The Apache Software Foundation. Apache cassandra. `https://cassandra.apache.org/`.

[24] The Apache Software Foundation. Apache hadoop. `http://hadoop.apache.org`.

[25] The Apache Software Foundation. Apache HBase. `https://hbase.apache.org`.

[26] Luis Galárraga, Katja Hose, and Ralf Schenkel. Partout: A distributed engine for efficient RDF processing. In *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14 Companion, pages 267–268, New York, NY, USA, 2014. ACM.

[27] Fabien Gandon and Guus Schreiber. RDF 1.1 XML syntax. `https://www.w3.org/TR/rdf-syntax-grammar/`, 2014.

[28] Sairam Gurajada, Stephan Seufert, Iris Miliaraki, and Martin Theobald. TriAD: A distributed shared-nothing RDF engine based on asynchronous message passing. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 289–300, New York, NY, USA, 2014. Association for Computing Machinery.

[29] Hae Chull Lim, Jaeho Lee, Byung Gon Kim, Youn Hee Kim, Hae Chull Lim, Jaeho Lee, Byung Gon Kim, and Youn Hee Kim. The path index for query processing on RDF and RDF schema. In *The 7th International Conference on Advanced Communication Technology, 2005, ICACT 2005.*, volume 2, pages 1237–1240, 2005.

[30] Xingwang Han, Zhiyong Feng, Xiaowang Zhang, Xin Wang, Guozheng Rao, and Shuo Jiang. On the statistical analysis of practical SPARQL queries. In

Proceedings of the 19th International Workshop on Web and Databases, WebDB '16, pages 2:1–2:6, New York, NY, USA, 2016. ACM.

[31] Razen Harbi, Ibrahim Abdelaziz, Panos Kalnis, Nikos Mamoulis, Yasser Ebrahim, and Majed Sahli. Accelerating SPARQL queries by exploiting hash-based locality and adaptive partitioning. *The VLDB Journal*, 25(3):355–380, June 2016.

[32] Stephen Harris and Nicholas Gibbins. 3store: Efficient bulk RDF storage. In *PSSS*, volume 89 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2003.

[33] Steve Harris, Nick Lamb, and Nigel Shadbolt. 4store: The design and implementation of a clustered RDF store. In *Scalable Semantic Web Systems Workshop - SSWS2009*, pages 94–109, 2009.

[34] Patrick Hayes. RDF semantics, W3C Recommendation 10 February. `https://www.w3.org/TR/rdf-mt/`, 2004.

[35] José-Miguel Herrera, Aidan Hogan, and Tobias Käfer. BTC-2019: the 2019 billion triple challenge dataset. In *ISWC (2)*, volume 11779 of *Lecture Notes in Computer Science*, pages 163–180. Springer, 2019.

[36] Johannes Hoffart, Fabian M. Suchanek, Klaus Berberich, and Gerhard Weikum. YAGO2: A spatially and temporally enhanced knowledge base from wikipedia. *Artif. Intell.*, 194:28–61, 2013.

[37] Katja Hose and Ralf Schenkel. WARP: workload-aware replication and partitioning for RDF. In *ICDE Workshops*, pages 1–6. IEEE Computer Society, 2013.

[38] Jiewen Huang, Daniel J. Abadi, and Kun Ren. Scalable SPARQL querying of large RDF graphs. *Proc. VLDB Endow.*, 4(11):1123–1134, 2011.

[39] Mohammad Farhan Husain, James P. McGlothlin, Mohammad M. Masud, Latifur R. Khan, and Bhavani M. Thuraisingham. Heuristics-based query processing for large RDF graphs using cloud computing. *IEEE Trans. Knowl. Data Eng.*, 23(9):1312–1327, 2011.

[40] Yannis E. Ioannidis and Stavros Christodoulakis. On the propagation of errors in the size of join results. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, SIGMOD '91, pages 268–277, New York, NY, USA, 1991. Association for Computing Machinery.

[41] Daniel Janke, Steffen Staab, and Matthias Thimm. Impact analysis of data placement strategies on query efforts in distributed RDF stores. *J. Web Semant.*, 50:21–48, 2018.

[42] Zoi Kaoudi and Ioana Manolescu. RDF in the clouds: a survey. *VLDB J.*, 24(1):67–91, 2015.

[43] George Karypis. METIS: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. `https://www.lrz.de/services/software/mathematik/metis/metis_5_0.pdf`.

[44] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, 1998.

[45] Karypis Lab. METIS: family of graph and hypergraph partitioning software. `http://glaros.dtc.umn.edu/gkhome/views/metis`, 2020.

[46] G. Ladwig and A. Harth. CumulusRDF: Linked data management on nested key-value stores. In *Proceedings of the 7th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2011) at the 10th International Semantic Web Conference (ISWC 2011), Bonn, Germany, October 24th, 2011*, pages 30–42, 2011.

[47] Kisung Lee and Ling Liu. Efficient data partitioning model for heterogeneous graphs in the cloud. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 46:1–46:12, New York, NY, USA, 2013. Association for Computing Machinery.

[48] Yannis Manolopoulos, Jaroslav Pokorný, and Timos K. Sellis, editors. *Advances in Databases and Information Systems, 10th East European Conference, ADBIS 2006, Thessaloniki, Greece, September 3-7, 2006, Proceedings*, volume 4152 of *Lecture Notes in Computer Science*. Springer, 2006.

[49] Daniel W. Margo and Margo I. Seltzer. A scalable distributed graph partitioner. *Proc. VLDB Endow.*, 8(12):1478–1489, 2015.

[50] Akiyoshi Matono, Toshiyuki Amagasa, Masatoshi Yoshikawa, and Shunsuke Uemura. A path-based relational RDF database. In *ADC*, volume 39 of *CRPIT*, pages 95–103. Australian Computer Society, 2005.

[51] Deborah L. McGuinness and Frank van Harmelen. OWL web ontology language overview. `https://www.w3.org/TR/owl-features/`, 2004.

[52] mkomo.com. A history of storage cost. `https://mkomo.com/cost-per-gigabyte-update`, 2014.

[53] Guido Moerkotte and Thomas Neumann. Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, VLDB '06, pages 930–941. VLDB Endowment, 2006.

[54] G. E. Moore. Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter*, 11(3):33–35, 2006.

[55] Raghava Mutharaju, Sherif Sakr, Alessandra Sala, and Pascal Hitzler. D-SPARQ: distributed, scalable and efficient RDF query engine. In *International Semantic Web Conference*, volume 1035 of *CEUR Workshop Proceedings*, pages 261–264. CEUR-WS.org, 2013.

[56] Thomas Neumann and Gerhard Weikum. The RDF-3X engine for scalable management of rdf data. *The VLDB Journal-The International Journal on Very Large Data Bases*, 19(1):91–113, 2010.

[57] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems, Third Edition.* Springer, 2011.

[58] Nikolaos Papailiou, Ioannis Konstantinou, Dimitrios Tsoumakos, and Nectarios Koziris. H2RDF: adaptive query processing on RDF data in the cloud. In *WWW (Companion Volume)*, pages 397–400. ACM, 2012.

[59] Nikolaos Papailiou, Dimitrios Tsoumakos, Panagiotis Karras, and Nectarios Koziris. Graph-aware, workload-adaptive SPARQL query caching. In *SIGMOD Conference*, pages 1777–1792. ACM, 2015.

[60] Peng Peng, Lei Zou, Lei Chen, and Dongyan Zhao. Query workload-based RDF graph fragmentation and allocation. In *EDBT*, pages 377–388. OpenProceedings.org, 2016.

[61] François Picalausa and Stijn Vansummeren. What are real SPARQL queries like? In *SWIM*, page 7. ACM, 2011.

[62] SWAT Projects. The lehigh university benchmark (LUBM). `http://swat.cse.lehigh.edu/projects/lubm/`.

[63] Eric Prud'hommeaux and Andy Seaborne. SPARQL query language for RDF. `https://www.w3.org/TR/rdf-sparql-query`, 2008.

[64] Roshan Punnoose, Adina Crainiceanu, and David Rapp. Rya: a scalable RDF triple store for the clouds. In *1st International Workshop on Cloud Intelligence (colocated with VLDB 2012), Cloud-I '12, Istanbul, Turkey, August 31, 2012*, page 4. ACM, 2012.

[65] Padmashree Ravindra, HyeongSik Kim, and Kemafor Anyanwu. An intermediate algebra for optimizing RDF graph pattern matching on mapreduce. In *ESWC (2)*, volume 6644 of *Lecture Notes in Computer Science*, pages 46–61. Springer, 2011.

[66] Laurens Rietveld, Rinke Hoekstra, Stefan Schlobach, and Christophe Guéret. Structural properties as proxy for semantic relevance in RDF graph sampling. In *International Semantic Web Conference (2)*, volume 8797 of *Lecture Notes in Computer Science*, pages 81–96. Springer, 2014.

[67] R.V.Guha. RDFDB: An RDF database. `http://www.cs.cmu.edu/afs/cs/usr/niu/rdf/`, 2000.

[68] Muhammad Saleem, Muhammad Intizar Ali, Aidan Hogan, Qaiser Mehmood, and Axel-Cyrille Ngonga Ngomo. LSQ: the linked SPARQL queries dataset. In *International Semantic Web Conference (2)*, volume 9367 of *Lecture Notes in Computer Science*, pages 261–269. Springer, 2015.

[69] Alexander Schätzle, Martin Przyjaciel-Zablocki, Christopher Dorner, Thomas Hornung, and Georg Lausen. Cascading map-side joins over HBase for scalable join processing. In *SSWS+HPCSW@ISWC*, volume 943 of *CEUR Workshop Proceedings*, pages 59–74. CEUR-WS.org, 2012.

[70] Amazon Web Services. DynamoDB. `http://aws.amazon.com/dynamodb/`.

[71] Zechao Shang and Jeffrey Xu Yu. Catch the wind: Graph workload balancing on cloud. In *ICDE*, pages 553–564. IEEE Computer Society, 2013.

[72] Bin Shao, Haixun Wang, and Yatao Li. Trinity: a distributed graph engine on a memory cloud. In *SIGMOD Conference*, pages 505–516. ACM, 2013.

[73] Longxiang Shi, Shijian Li, Xiaoran Yang, Jiaheng Qi, Gang Pan, and Binbin Zhou. Semantic health knowledge graph: Semantic integration of heterogeneous medical knowledge and services. *BioMed Research International*, 2017:1–12, 01 2017.

[74] Nasrin Mazaheri Soudani, Afsaneh Fatemi, and Mohammadali Nematbakhsh. An investigation of big graph partitioning methods for distribution of graphs in vertex-centric systems. *Distributed Parallel Databases*, 38(1):1–29, 2020.

[75] Raffael Stein and Valentin Zacharias. RDF on cloud number nine. In *Workshop on New Forms of Reasoning for the Semantic Web: Scalable and Dynamic*, 2010.

[76] Markus Stocker, Andy Seaborne, Abraham Bernstein, Christoph Kiefer, and Dave Reynolds. SPARQL basic graph pattern optimization using selectivity estimation. In *WWW*, pages 595–604. ACM, 2008.

[77] 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 *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 553–564. VLDB Endowment, 2005.

[78] Patrick van Kleef. DBpedia usage report. `https://medium.com/virtuoso-blog/dbpedia-usage-report-as-of-2018-01-01-8cae1b81ca71`, 2018.

[79] W3C. Linked data. `https://www.w3.org/standards/semanticweb/data`, 2009.

[80] Lu Wang, Yanghua Xiao, Bin Shao, and Haixun Wang. How to partition a billion-node graph. In *ICDE*, pages 568–579. IEEE Computer Society, 2014.

[81] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore: sextuple indexing for semantic web data management. *Proceedings of the VLDB Endowment*, 1(1):1008–1019, 2008.

[82] Kevin Wilkinson, Craig Sayers, Harumi Kuno, and Dave Reynolds. Efficient RDF storage and retrieval in Jena2. In *Proceedings of the First International Conference on Semantic Web and Databases*, SWDB'03, pages 120–139. CEUR-WS.org, 2003.

[83] Buwen Wu, Hai Jin, and Pingpeng Yuan. Scalable SAPRQL querying processing on large RDF data in cloud computing environment. In *Pervasive Computing and the Networked World*, pages 631–646. Springer Berlin Heidelberg, 2013.

[84] Buwen Wu, Yongluan Zhou, Pingpeng Yuan, Ling Liu, and Hai Jin. Scalable SPARQL querying using path partitioning. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, pages 795–806. IEEE, 2015.

[85] Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao, and Zhongyuan Wang. A distributed graph engine for web scale RDF data. In *Proceedings of the 39th international conference on Very Large Data Bases*, PVLDB'13, pages 265–276. VLDB Endowment, 2013.

[86] Xiaofei Zhang, Lei Chen, Yongxin Tong, and Min Wang. EAGRE: towards scalable I/O efficient SPARQL query evaluation on the cloud. In *ICDE*, pages 565–576. IEEE Computer Society, 2013.

[87] Xiaofei Zhang, Lei Chen, and Min Wang. Towards efficient join processing over large RDF graph using Mapreduce. In *Proceedings of the 24th International Conference on Scientific and Statistical Database Management*, SSDBM'12, pages 250–259. Springer-Verlag, 2012.

[88] Matthäus Zloch, Maribel Acosta, Daniel Hienert, Stefan Dietze, and Stefan Conrad. A software framework and datasets for the analysis of graph measures on RDF graphs. In *ESWC*, volume 11503 of *Lecture Notes in Computer Science*, pages 523–539. Springer, 2019.