National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science

Scalable, workload aware indexing and query processing over unstructured data

DOCTORAL DISSERTATION

NIKOLAOS PAPAILIOU

Athens, 01/11/2016

National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science

Scalable, workload aware indexing and query processing over unstructured data

DOCTORAL DISSERTATION

NIKOLAOS PAPAILIOU

Athens, 01/11/2016

. . . .
NIKOLAOS PAPAILIOU
Doctor of Electrical and Computer Engineering, NTUA

This work has received funding from IKY fellowships of excellence for postgraduate studies in Greece - SIEMENS program.

To my family.

Acknowledgments

This dissertation is the result of my graduate studies at the Computing Systems Laboratory of the National Technical University of Athens, under the supervision of Prof. Dimitrios Tsoumakos and Prof. Nectarios Koziris. I would like to take this opportunity to express my deepest gratitude and appreciation to all the people that guided my way these years.

First of all, I am deeply thankful to my advisor, Dimitrios Tsoumakos, who was my main mentor and source of encouragement and inspiration. I learned a lot from him, ranging from skills such as holding a high bar for one's research all the way to paper writing and system design. This dissertation would not have been the same without his continuous effort and support.

I would also like to extend my deep gratitude to my advisor, Nectarios Koziris, who gave me the opportunity to study a challenging research subject. His constant support, both intellectual and material, was of paramount importance to my study. I truly enjoyed working in a research environment that stimulates innovation and has the potential to support research on cutting edge technologies.

Furthermore, this thesis would not have been possible without the financial support of IKY fellowships of excellence for postgraduate studies in Greece - SIEMENS program. I owe great debt of gratitude and respect to the IKY state scholarships foundation as well as to Mrs. Eirini Stamogiorgou who managed the program and was my primary point of contact.

I would also like to express my special appreciation and thanks to several other people at the Computing Systems Laboratory who have assisted me along the way: Ioannis Konstatninou, Katerina Doka, Ioannis Giannakopoulos, Ioannis Mytilinis, Dimitris Sarlis and Victor Giannakouris. Thank you all for the wonderful collaboration, the endless discussions and your friendship throughout these years.

Last but not the least, I would like to thank my family and all my friends for their unconditional love, support and help, as well as for the joy they bring in my life.

Contents

List of Figures

List of Tables

Abstract

The pace at which data are described, queried and exchanged, using unstructured data representations, is constantly growing. Semantic Web technologies have emerged as one of the prevalent unstructured data sources. Utilizing the RDF description model, they attempt to encode and make openly available various World Wide Web datasets. Therefore, the constantly increasing volume of available data calls for efficient and scalable solutions for their management. In this thesis, we devise distributed algorithms and techniques for data management, which can scale and handle huge datasets. We introduce H2RDF+, a fully distributed RDF store that combines the MapReduce processing framework with a NoSQL distributed database. Creating 6 indexes over HBASE tables, H2RDF+ can process complex queries making adaptive decisions on both the join ordering and the join execution. Joins are executed using in distributed or centralized resources, depending on their cost. Furthermore, we present a novel system that addresses graph-based, workload-adaptive indexing of large RDF graphs by caching SPARQL query results. At the heart of the system lies a SPARQL query canonical labelling algorithm that is used to uniquely index and reference SPARQL query graphs as well as their isomorphic forms. We integrate our canonical labelling algorithm with a dynamic programming planner in order to generate the optimal join execution plan, examining the utilization of both primitive triple indexes and cached query results. By monitoring cache requests, our system is able to identify and cache SPARQL queries that, even if not explicitly issued, greatly reduce the average response time of a workload. The proposed cache is modular in design, allowing integration with different RDF stores.

Another ever-increasing source of unstructured data is the Internet traffic. Network datasets collected at large networks such as Internet Exchange Points (IXPs) can be in the order of Terabytes per hour. To handle analytics over such datasets, we present Datix, a fully decentralized, open-source analytics system for network traffic data that relies on smart partitioning storage schemes to support fast join algorithms and efficient execution of filtering queries. In brief, Datix manages to efficiently answer queries within minutes compared to more than 24 hours processing when executing existing Python-based code in single node setups. Datix also achieves nearly 70% speedup compared to baseline query implementations of popular big data analytics engines such as Hive and Shark.

Περίληψη

Ο ρυθμός με τον οποίο τα δεδομένα περιγράφονται, ερωτώνται και ανταλλάσσονται χρησιμοποιώντας μη δομημένες αναπαραστάσεις δεδομένων συνεχώς αυξάνεται. Μια από τις κυριότερες πηγές τέτοιων δεδομένων είναι οι τεχνολογίες Σημασιολογικού Ιστού, οι οποίες χρησιμοποιούν το RDF μοντέλο για την αναπαράσταση των δεδομένων του παγκόσμιου ιστού. Η μεγάλη αύξηση των διαθέσιμων RDF δεδομένων επιβάλει την εύρεση αποδοτικών και κλιμακώσιμων λύσεων για την διαχείρισή τους. Σε αυτή την διατριβή χρησιμοποιούμε κατανεμημένες μεθόδους διαχείρισης των RDF δεδομένων, οι οποίες μπορούν να κλιμακώσουν σε απεριόριστα μεγάλο αριθμό δεδομένων. Παρουσιάζουμε το H2RDF, μια πλήρως κατανεμημένη βάση αποθήκευσης RDF δεδομένων, η οποία συνδυάζει το πλαίσιο επεξεργασίας του MapReduce με μια κατανεμημένη NoSQL βάση. Δημιουργώντας 6 διαφορετικά ευρετήρια δεδομένων με HBASE πίνακες, το H2RDF μπορεί να επεξεργαστεί σύνθετα ερωτήματα με κλιμακώσιμο τρόπο κάνοντας προσαρμοστικές αποφάσεις για την σειρά και τον τρόπο εκτέλεσης των συνενώσεων. Οι συνενώσεις εκτελούνται κατανεμημένα ή κεντρικά, σε έναν υπολογιστή, ανάλογα με το κόστος τους. Επιπλέον, παρουσιάζουμε ένα καινοτόμο σύστημα που στοχεύει στην προσαρμοστική και βασισμένη στα ερωτήματα που εκτελούνται, δεικτοδότηση RDF γράφων με τη χρήση μιας κρυφής μνήμης για αποτελέσματα SPARQL ερωτημάτων. Στην καρδιά του συστήματος βρίσκεται ένας αλγόριθμος που παράγει κανονικοποιημένες ετικέτες για SPARQL ερωτήματα και χρησιμοποιείται για την μονοσήμαντη δεικτοδότηση και αναφορά σε SPARQL υπογράφους, αντιμετωπίζοντας το πρόβλημα των ισομορφικών γράφων. Ένας αλγόριθμος δυναμικού προγραμματισμού χρησιμοποιείται για την εύρεση του βέλτιστου πλάνου εκτέλεσης των ερωτημάτων, εξετάζοντας την αξιοποίηση τόσο των βασικών RDF ευρετηρίων καθώς και

των προσωρινά αποθηκευμένων αποτελεσμάτων SPARQL ερωτημάτων. Με την παρακολούθηση των αιτημάτων στην κρυφή μνήμη, το σύστημά μας είναι σε θέση να προσδιορίσει και να τοποθετήσει στην κρυφή μνήμη ερωτήματα που, αν και δεν έχουν ζητηθεί, μπορούν να μειώσουν τους χρόνους εκτέλεσης των ερωτημάτων των χρηστών. Η προτεινόμενη κρυφή μνήμη είναι επεκτάσιμη, επιτρέποντας την ενσωμάτωσή της σε πολλαπλές RDF βάσεις δεδομένων.

Μια ακόμα πηγή συνεχώς αυξανόμενης ποσότητας δεδομένων είναι και η κίνηση δεδομένων στο Internet. Αυτό γίνεται περισσότερο εμφανές σε κόμβους ουδέτερης διασύνδεσης (IXPs) από τους οποίους πλέον διέρχονται έως και Terabytes δεδομένων ανά ώρα. Για την αποδοτική διαχείριση και επεξεργασία τέτοιων δεδομένων παρουσιάζουμε το Datix, ένα πλήρως κατανεμημένο, ανοιχτού κώδικα σύστημα ανάλυσης δεδομένων κίνησης δικτύων. Το Datix βασίζεται σε τεχνικές έξυπνης κατανομής των δεδομένων, οι οποίες μπορούν να χρησιμοποιηθούν για την υποστήριξη γρήγορων συνενώσεων και αποδοτικών λειτουργιών επιλογής δεδομένων. Σαν αποτέλεσμα, το Datix πετυχαίνει να εκτελεί σε λίγα λεπτά ερωτήματα που απαιτούσαν έως και μέρες χρησιμοποιώντας τις υπάρχουσες τεχνολογίες κεντρικής επεξεργασίας. Επίσης παρουσιάζει έως και 70% μείωση χρόνου εκτέλεσης σε σχέση με αντίστοιχες δημοφιλείς πλατφόρμες κατανεμημένης επεξεργασίας, όπως το Hive και το Shark.

CHAPTER 1

Introduction

This dissertation introduces efficient algorithms that enable distributed query execution and indexing over unstructured or semi-structured data.

1.1 Motivation

The amount of publicly and privately available data has been exploding within the recent years [Manyika 11]. Companies and organizations alike capture trillions of bytes of information about their customers, suppliers, and operations. At the same time, interconnected devices such as smart phones, sensors, etc. contribute to the "data deluge". Managing and analyzing such large data sets, so-called "big data", is becoming a key basis of competition, underpinning new waves of productivity growth, innovation, and consumer surplus. It comes as no surprise that, personalized services, the rise of multimedia, social media, and the Internet of Things in turn fuel exponential data growth for the foreseeable future [Lohr 12].

Traditionally, data management platforms were designed for structured data. The relational model [Codd 70] has been driving research and innovation on data management systems for at least 4 decades. However, lately a series of disruptive technologies and applications have challenged the status quo on the production, management, analysis and utilization of data. Rather than being scarce, specific, private and limited in access, data are now produced in high volumes, have vast variety, are stored with low cost and are increasingly open and accessible. In

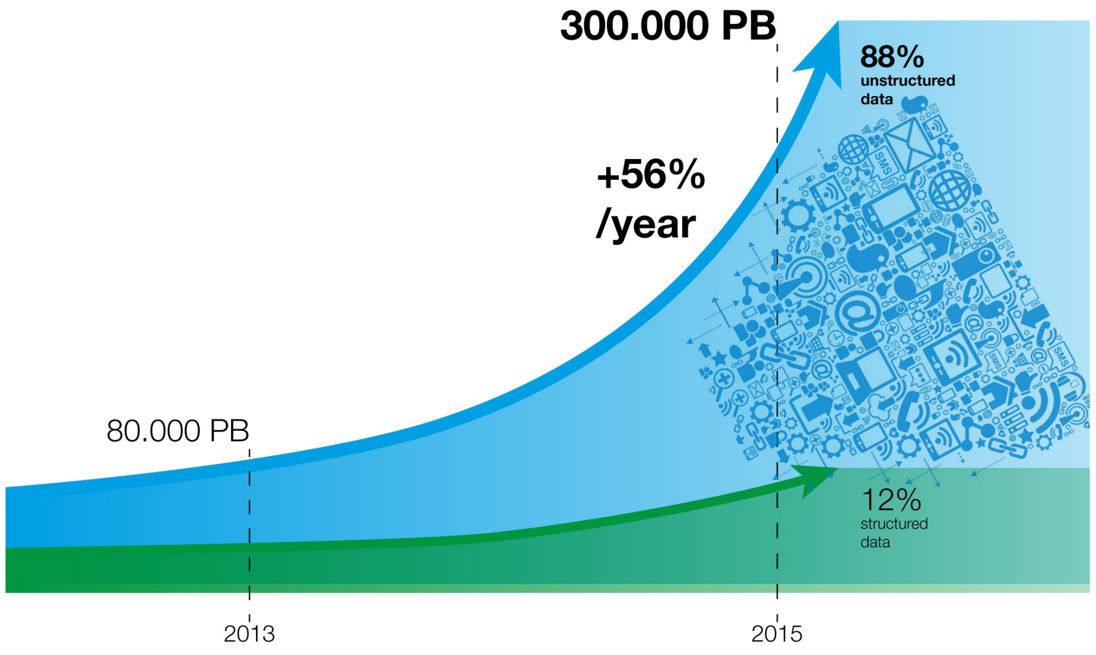


Figure 1.1: *Unstructured Data Growth*

this highly complex and diverse data landscape, the ever-growing data are generated in multiple formats having no standardized form.

Figure 1.1 depicts the recent trends in data growth. It is clearly visible that the total amount of unstructured data has surpassed the respective amount of structured data. Only a mere 12% of the existing data is structured. What is more, the amount of unstructured data grows exponentially with an increase ratio of 56% per year. Unstructured content is fundamentally different from structured data and must be handled appropriately for use in big data analysis applications.

Cloud and Distributed Computing are emerging as the main technologies that attempt to tackle the "big data" challenge. While distributed computing leverages the power of decentralized resources, Cloud Computing has become the de-facto, pay-as-you-go paradigm for allocating and deallocating such resources, transforming the design and philosophy of the entire IT industry. Developers with innovative ideas can directly start implementing them without worrying about investing large capital expenditures for the purchase and maintenance of hardware resources. They can avoid overspending for a service that does not meet the expected user engagement as well as underspending for a service that gets quickly popular. Therefore, this resource elasticity has emerged as the biggest advantage of the Cloud Computing model.

1.2 Semantic Web

One of the largest sources of data nowadays is the World Wide Web, an information space where documents and other resources can be shared, interlinked and accessed via the Internet. Information production and consumption on the Web is moving toward the so-called "Semantic Web". The "Semantic Web" describes an extension of the Web where computers will be able to process efficiently the data and reason about them exactly as human do. This vision was originally expressed by Tim Berners-Lee who stated the following:

"I have a dream for the Web, in which computers become capable of analyzing all the data: the content, links, and transactions between people and computers."

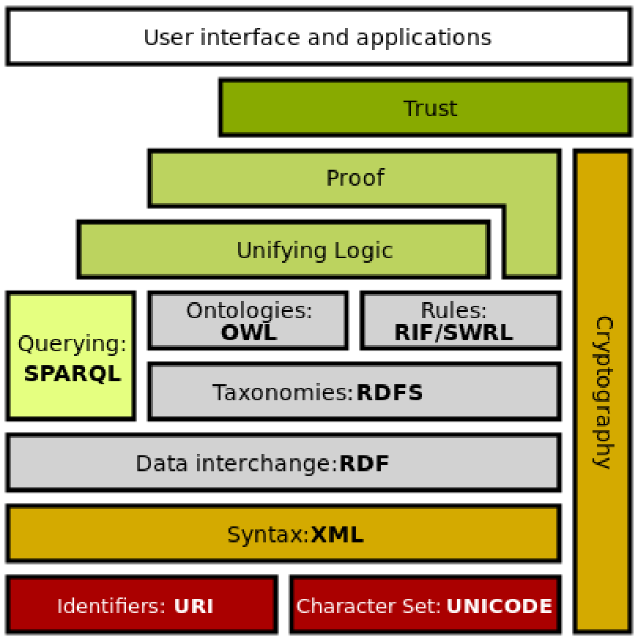


Figure 1.2: *Semantic Web Technologies*

This era targets the representation and integration of data originating from different application, enterprise, and community boundaries. The term "Semantic Web" is often used more specifically to refer to the formats and technologies that enable it. Figure 1.2 depicts all the various technologies that have been presented for materializing the "Semantic Web" concept. Initially, the Resource Description Framework (RDF)[1] has been proposed for the data representation. As mentioned before, the schema-dependent data representation formats, like the traditional relational representation, fail to adjust and handle the diversity and constantly changing

[1]`https://www.w3.org/RDF/`

nature of the Internet data. As a result, the Semantic Web community has acknowledged the RDF standard [Manola 04] together with the SPARQL query language [Prud'hommeaux 06] as its de facto technologies.

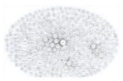
DBpedia	RDF-encoded Wikipedia	1.89 billion triples
BIO2RDF	RDF-encoded biological data	2.7 billion triples
DATA.GOV	US government data in RDF	5 billion triples
Semantic Web Challenge	Crawled Web data	2 billion triples
United States Census 2010	US population statistics	1 billion triples
yago select knowledge	Yago facts from Wikipedia, Wordnet, Geonames	0.12 billion triples
	Linked Open Data cloud	30 billion triples

Figure 1.3: *Semantic Web Datasets*

The schema-free nature of RDF data allows the formation of a common data framework that facilitates the integration of various data sources. This property has led to an unprecedented increase in the rate at which RDF data is created, stored and queried, even outside the purely academic realm and, consequently, to the development of many RDF stores [Weiss 08, Neumann 10a, Zeng 13, Atre 08, Bonstrom 03] that target RDF indexing and efficient SPARQL querying performance. Figure 3.1 contains a list of the most popular public datasets along with their respective sizes. To present a more detailed picture of the "Semantic Web" data growth, Figure 1.4 illustrates the evolution of the publicly available linked datasets as well as their interconnections.

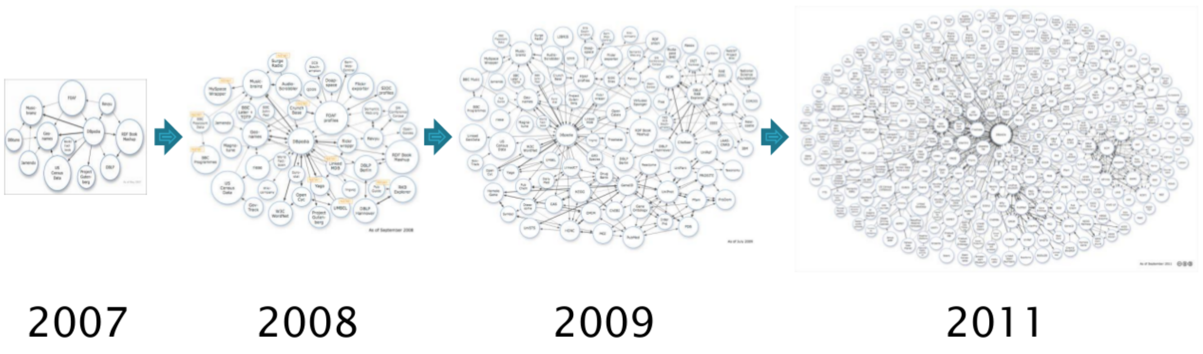


Figure 1.4: *Semantic Web Technologies*

1.2.1 Resource Description Framework (RDF)

The Resource Description Framework (RDF) is a flexible data model that expresses unstructured information as statements about resources. In essence, the RDF data model is naturally represented as a directed labeled graph, where nodes correspond to resources or literals and edges are used to describe the relations between the resources. Each RDF statement, triple, has the form (s p o) where s stands for the subject, p for the property, and o for the object. Viewing RDF from the graph perspective, each triple corresponds to a directed and labeled graph edge. Figure 1.5 shows a visual representation of an RDF triple.

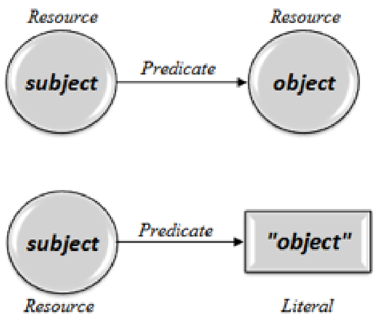


Figure 1.5: *RDF triples*

Triples are merged together to form a knowledge graph that contains all the information of the individual triples. This requires that all resources are uniquely identified, i.e., all resources have a unique resource identifier (URI). This identification is a central notion for the vision of the "Semantic Web" as it extends the unique identification of web pages, present in the current version of the Web, to any resource of the physical or digital world.

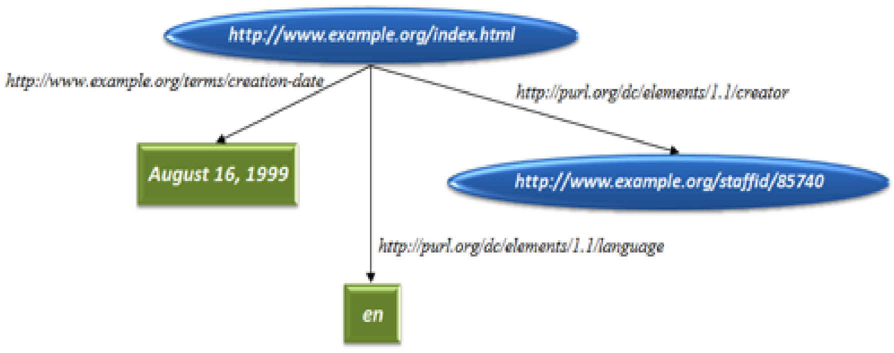

http://www.example.org/index.html
http://www.example.org/terms/creation-date
http://purl.org/dc/elements/1.1/creator
August 16, 1999
http://www.example.org/staffid/85740
http://purl.org/dc/elements/1.1/language
en

Figure 1.6: *Knowledge Graph*

For example, the knowledge graph of Figure 1.6 contains information about the creator, the language and the creation date of the web page http://www.example.org/index.html.

1.2.2 SPARQL Query Language

The RDF graph based data specification defines the syntax and semantics of the SPARQL query language. SPARQL can be used to express queries across diverse data sources, whether the data is stored natively as RDF or viewed as RDF via middleware. The basic idea of SPARQL querying is based on subgraph matching. Therefore, queries in this context are in essence graph patterns containing bound or variable labels on their nodes and edges. For example, Figure 1.7 depicts a SPARQL query that retrieves all FullProfessors that work for NTUA along with their names, emails and telephone numbers.

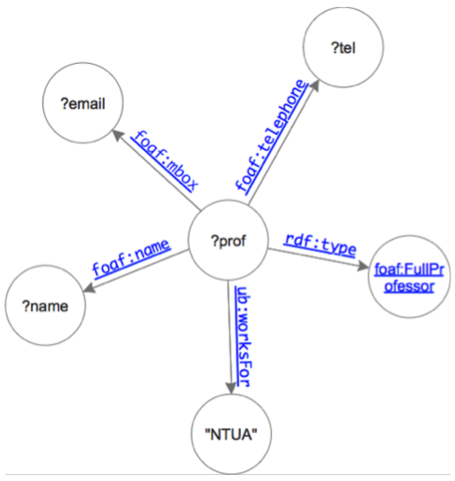


Figure 1.7: *SPARQL query*

Nodes with names that start with "?" are query variables. A SPARQL query engine must find all subgraphs of the entire RDF knowledge graph that match with the query in hand. SPARQL also specifies more generic capabilities like optional graph patterns, property paths, inference as well as result aggregation and ordering.

1.2.3 Contributions

The first contribution of this thesis, is H_2RDF+ (Chapter 2), a fully distributed, cloud-enabled, RDF store that combines the MapReduce processing framework with a NoSQL distributed database. H_2RDF+ is highly scalable, performing distributed Merge and Sort-Merge joins over a multiple index scheme. Depending on aggressive byte-level compression and result grouping over fast scans, it can process both complex and selective join queries in a highly efficient manner. Furthermore, it adaptively chooses for either single- or multi-machine execution based on join complexity estimated through index statistics. Our experimental evaluation demonstrates that H_2RDF+ excels in multi-join and nonselective queries, scaling linearly to the amount of available resources while achieving interactive, comparable to centralized RDF stores, execution for selective queries.

However, the move from the schema-dependent relational to the schema-free RDF data has brought forth new indexing and querying challenges. Extensively used schema based optimizations cannot be easily extended in the RDF context. For instance:

- Grouping data that are accessed together using tables.
- Indexing according to filtering and join operations.
- View materialization of frequently queried data patterns.

In fact, RDF databases assume limited knowledge of the data structure. This usually happens via RDFS triples [Brickley 14]. However, RDFS information is not as rich and mandatory as the SQL schema. It can be incomplete and change rapidly along with the dataset. Therefore, most RDF databases target the indexing of individual RDF edges resulting in query executions with much more joins than if processing the same dataset in a schema-aware manner. For example, TPC-H Query 2 written in SPARQL requires 26 joins while the respective SQL formulation contains only 5 joins [Gubichev 14]. In contrast, RDF databases that use RDFS information to store and group RDF data [Stuckenschmidt 04, Tran 10], fail to effectively adapt to schema changes and to non-conforming data.

In state-of-the-art graph databases [Zhao 07, Yan 04], we note that frequent and discriminative graph pattern indexing is extensively used. However, RDF stores have not yet taken advantage of these techniques. What is more, these schemes focus on *static* indexing of important graph patterns, namely they index subgraphs solely based on the underlying dataset, without any regard for the workload. However, the diversity of the applied SPARQL workloads together with the requirement for high performance for all the different workloads calls for dynamic, workload-driven indexing solutions (e.g., [Idreos 11]).

Furthermore, SPARQL queries tend to increase in complexity and become hard to optimize using dynamic programming algorithms. For instance, DBpedia reports that its SPARQL endpoint query log contains queries with up to 10 patterns [Gallego 11]; analytical queries in the biomedical domain may include more than 50 patterns [Sahoo 10]. Experimental results show that finding the optimal join plan using dynamic programming approaches can have prohibitive execution times for queries with more than 15 patterns [Neumann 10a, Gubichev 14]. While there exist several greedy, heuristic approaches for SPARQL query planning [Tsialiamanis 12, Papailiou 13], they might select a suboptimal join plan, adding overheads to the query execution time [Gubichev 14] and cannot be easily integrated with a cached result discovery algorithm that finds *all* usable cached results.

As a result, the second contribution of this thesis (Chapter 3) is the adaptive indexing and caching of SPARQL query results in order to effectively detect frequent query patterns and offer data grouping and view materialization properties. We propose a caching framework that can offer re-usability of computed SPARQL results. By actively monitoring the SPARQL query workload, our system can detect cross-query frequent patterns and trigger their materialization and caching in order to boost the performance of consequent queries and adapt to the workload. We also introduce a SPARQL query simplification algorithm, used to handle complex query graphs and offer near optimal execution plans that take into account the utilization of cached query subgraphs. In brief, this work introduces:

- A SPARQL query simplification algorithm (Section 3.2), based on the star simplification techniques, that splits the query in a skeleton graph and multiple star graphs.
- A SPARQL query *canonical labelling* algorithm (Section 3.3) that is able to generate canonical string labels, identical among all isomorphic forms of a SPARQL query. These labels are used as cache keys to store and retrieve information related to SPARQL query graphs. Furthermore, our query simplification technique manages to effectively reduce the complexity of the canonical labelling algorithm for complex query graph structures.
- A *Dynamic Programming Planner* [Moerkotte 08] is extended in order to issue cache requests for all query subgraphs using their *canonical label* (Section 3.4). The planner is also extended to handle multi-way join exploration and cached result discovery over our simplified query graphs. The resulting optimal execution plan may thus involve, in part or in whole, cached query subgraphs.
- A *Cache Controller* process monitors all cache requests issued by the workload queries, enabling it to detect cross-query *profitable* patterns and trigger their execution and caching.

The proposed cache is modular in design, allowing integration with different RDF stores. Incorporating it to H_2RDF+ we prove that workload-adaptive caching can reduce average response times by up to two orders of magnitude and offer interactive response times for complex workloads and huge RDF datasets.

1.3 Network Data Analytics

Another source of "big unstructured data" is the actual Internet traffic which increases at a pace that makes it difficult to track its growth and trends in a systematic and scalable way. Indeed, recent studies show that Internet traffic continues to grow by more than 30% annually as it has done the last twenty years and is expected to continue at the same pace in the future [Cisco 13].

Therefore, network traffic analytics have become a necessity nowadays. It is essential to provide a systematic and scalable method of analyzing such network monitoring data and thus, aiding network administrators in identifying the different types of traffic that traverses a network. The extracted knowledge provides invaluable assistance in effectively dealing with factors such as bandwidth, security and application priority or detecting problems such as phishing attempts, DDoS attacks, spoofing etc.

Network datasets collected at large Internet Service Providers (ISPs) and Internet Exchange Points (IXPs) have been at the forefront of network analytics. ISPs serve, depending on their footprint, thousands to tens of millions of end-users daily and facilitate billions of network connections [Poese 10]. IXPs consist of physical machines (core switches) to facilitate traffic exchange among different types of networks [Chatzis 13a]. Some of the most successful IXPs, connect more than 600 networks and are handling aggregate traffic that is peaking at multiples of TB per second. To put this traffic into perspective, on an average business day in 2013, one of the largest IXPs, AMS-IX in Amsterdam, exchanged around 25 PB while AT&T and Deutsche Telekom reported carrying 33 PB and 16 PB of data traffic respectively [Chatzis 13a].

Current techniques for capturing, storing and analyzing network traffic data rely on centralized architectures that fail to cope with their ever increasing volumes and generation rate. Surprisingly, in the era of Big Data and Cloud Computing, centralized proprietary tools, e.g., sFlow[2] and NetFlow[3] or custom serial scripts are still considered state-of-the-art. An sFlow record contains Ethernet frame samples and captures the first 128 bytes of each sampled frame. This implies that in the case of IPv4 packets the available information consists of the full IP and transport layer headers (i.e., source and destination IPs and ports, protocol information and byte count) and 74 and 86 bytes of TCP and UDP payload, respectively.

In order to extract aggregated monthly or weekly reports on massive amounts of monitoring data, system administrators rely on proprietary script-based approaches, something that limits both the expressiveness and the size of the analyzed data. Up till now, there are few big-data enabled open-source tools that can be used by IXP administrators to perform typical reporting and monitoring tasks on the collected passive monitoring data. Taking into account that both the size of the data and the posed regulations do not allow IXPs to keep the collected data for a long amount of time, most of the times the collected data is not evaluated, and its "secrets" remain hidden and lost. Lately, it has been shown that passive monitoring data does contain valuable information, not only for the IXP itself, but for the entire Internet structure [Chatzis 13b]. Examples of such information are depicted in Figure 1.8 where a global view of the entire internet was achieved using analytics over data from one IXP. Although this

[2]`http://www.sflow.org/`

[3]`http://www.cisco.com/c/en/us/products/ios-nx-os-software/ios-netflow/index.html`

		week 45	educated guesses of ground-truth
Peering Traffic	IPs	232,460,635	unknown $< 2^{32}$
	#ASes	42,825	approx. 43K
	Subnets	445,051	450K+
	countries	242	250
Server Traffic	IPs	1,488,286	unknown
	#ASes	19,824	unknown
	Subnets	75,841	unknown
	Countries	200	250

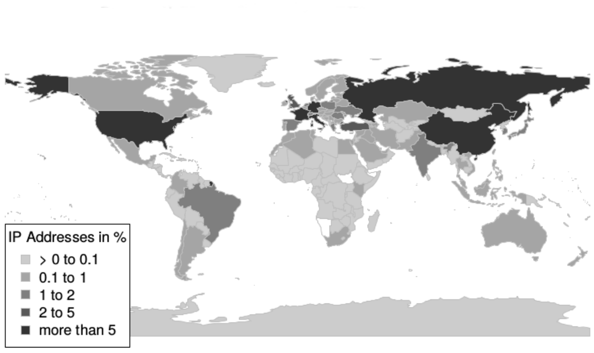


Figure 1.8: *IXP Summary statistics and percentage of IPs per country*

information was extracted using simple script-based centralized approaches, a more systematic approach with the use of state-of-the-art big-data processing tools is currently missing.

1.3.1 Contributions

The last part of this thesis (Chapter 4), attempts to overcome the aforementioned problems related to both scalability and efficient query executions. We design and implement Datix: a scalable, network traffic data analysis system. Our system is based on distributed techniques for data analytics, such as MapReduce [Dean 08] and is capable of solving the more general problem of log processing as described in [Blanas 10]. Our goal is to implement efficient distributed join algorithms to combine the information from a main dataset, in our case being the sFlow data collected at an IXP, with additional complementary information provided by secondary datasets, such as the mapping of IP addresses to their corresponding AS, IP geolocation, i.e., country of origin, and IP profile information, e.g., reverse DNS lookup as reported by Internet measurement studies such as ZMap [Durumeric 13]. Our contributions can be summarized as follows:

- We introduce a smart way of pre-partitioning the dataset in files that contain records of a certain range of values, so as to facilitate data processing and query execution.
- Using this particular partitioning scheme we are able to efficiently execute filtering queries, for example across a certain time period, avoiding the need to process the entire dataset but instead accessing only the necessary files.
- We integrate these features into Datix, an SQL compliant network data analysis system, by implementing distributed join algorithms, such as map join [Blanas 10], in combination with custom-made user-defined functions that are aware of the underlying data format.

1.4 Document Outline

The rest of this document is organized as follows. Chapter 2 describes the design and implementation of the H_2RDF+ distributed datastore. Section 2.1 introduces H_2RDF+. Section 2.2 describes H_2RDF+'s architecture. Section 2.3 discusses our indexing and storage decisions. Section 2.5 analyzes our distributed join algorithms. Section 2.6 introduces H_2RDF+'s query planner. To evaluate the presented system, Section 2.7 contains extensive experiments and comparisons of H_2RDF+ and other state-of-the-art systems.

In Chapter 3, we extend the work on H_2RDF+ by introducing a SPARQL query caching framework. Section 3.2 describes our SPARQL query simplification algorithm. Section 3.3 analyses our SPARQL query canonical labelling algorithm. Section 3.4 introduces our extended Dynamic Programming Query Planner. Section 3.6 discusses our Cache Controller module and Section 3.7 presents the experimental evaluation of our caching framework on top of H_2RDF+.

To handle big network data analytics, Chapter 4 introduces Datix. Section 4.1 motivates the proposed system. Section 4.2 describes our main architectural decisions. In addition, Section 4.3 presents our distributed join algorithms while Section 4.4 contains a detailed experimental evaluation of the system.

A detailed survey of the related work on both RDF datastores and network data analytics is presented in Chapter 5. Lastly, Chapter 6 concludes the thesis.

CHAPTER 2

H_2RDF+ Distributed RDF Datastore

2.1 Introduction

Distributed triple stores decentralize some or all the stages of RDF data management. Yet, they do not flexibly adjust their behavior with respect to the query in hand, either committing on a specific join algorithm or the execution platform's resources. SPARQL queries often require multiple joins over a (possibly large) number of triple patterns and variables that the query contains. Thus, a resolution engine would need to adjust its execution with respect to both query input and complexity. Single joins range in complexity as input and selectivity range. As the number of joins and intermediate results to be processed increase, this should, correspondingly, not lead to an exponential growth of response times.

Furthermore, distributed approaches have not yet taken advantage of maintaining all permutations of RDF elements, namely *spo, pso, pos, ops, osp* and *sop* indexes [Weiss 08]. Such a scheme offers the following advantages: (1) All SPARQL triple patterns can be answered efficiently using a single index scan on the corresponding index. For example, a triple pattern with bound subject and variable predicate/object can be answered using a range scan on the *spo* or the *sop* index. (2) Merge joins that exploit the precomputed orderings can be extensively employed. The existence of all six indexes guarantees that every join between triple patterns can be done using merge joins. More expensive join algorithms are needed only when joining unordered intermediate results. In the H_2RDF+[1] case, we maintain both of these properties

[1]https://github.com/npapa/h2rdf, http://h2rdf.googlecode.com

while moving towards a distributed and scalable environment. We summarize the main contributions of this work as follows:

- We devise an indexing scheme for storing RDF data implemented in HBase, which allows bulk-import jobs to load and index large RDF datasets. We optimize the retrieval capabilities of our distributed index by applying aggressive compression and minimizing the storage requirements. The latter is coupled with the use of an intermediate result materialization that maintains groups of bindings. Our indexes compensate the fact of being nearly $10\times$ slower than disk-based B^+trees by achieving great scalability and parallel scanning performance.
- We present fully scalable, distributed (MapReduce based) versions of the well-studied multiway Merge and Sort-Merge join algorithms.
- We devise a join cost model and use the estimated cost of each join to greedily decide on the order of joins and the platform (central or distributed) of their execution.
- We perform a thorough experimental evaluation of our system. Results show that H_2RDF+ can be orders of magnitude faster than a state-of-the-art centralized store [Neumann 10a] for complex, non-selective joins, while being only tenths of a second slower in selective ones. Moreover, it proves 6–8 times faster than its previous version [Papailiou 12] and up to orders of magnitude faster than an alternative MapReduce-based scheme [Husain 11]. H_2RDF+ easily scales to 14 billion triples (2.5TB) using a cluster of 35 VMs, providing linear scalability in terms of the amount of available resources.

2.2 Architecture

Figure 2.1 presents an overview of the H_2RDF+ [Papailiou 13, Papailiou 14] architecture. The system uses HBase[2] as a distributed indexing substrate for large triple datasets. RDF data is imported in HBase through a bulk import process. Users are able to pose SPARQL queries parsed by a Jena [Carroll 04] parser that checks the query syntax and creates the query graph. Our *Join Planner* and *Executor* modules choose the join that needs to be executed, the algorithm to be used, the method of execution (centralized or distributed) and the required resources on a per-join basis. Our system is available as an open-source project and offers RDF data indexing and SPARQL querying functionality as well as a user friendly web-interface.

[2]https://hbase.apache.org/

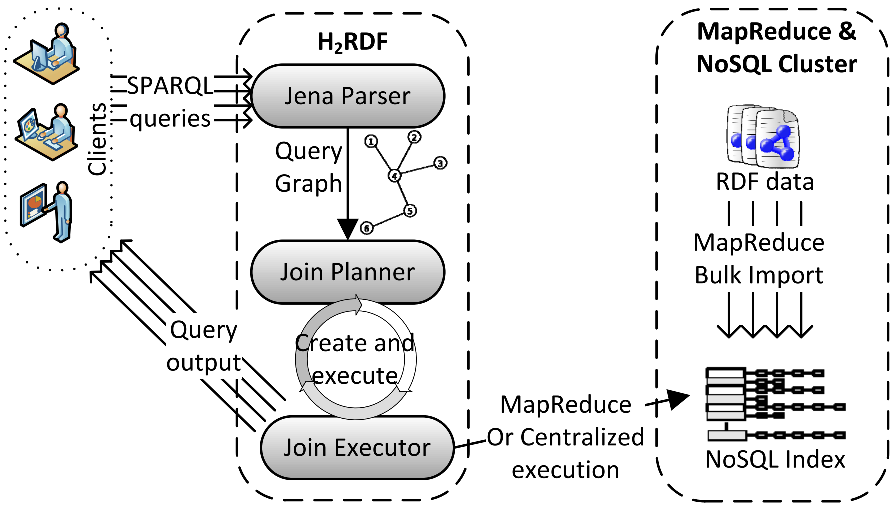


Figure 2.1: *H_2RDF+ architecture*

2.3 Indexing Scheme

In this section, we explain the decisions made about the number and type of indexes used in our system. Although most state of the art centralized RDF stores use combinations of Hexastore-like indexes, distributed approaches have not yet taken advantage of this technique to offer increased scalability and fast querying. Maintaining all six permutations of RDF elements, namely *spo, pso, pos, ops, osp* and *sop*, offers the following advantages: (1) All SPARQL triple patterns can be answered efficiently using a single index scan on the corresponding index. For example, a triple pattern with bound subject and variable predicate/object can be answered using a range scan on the *spo* or the *sop* index. (2) Merge joins that exploit the precomputed orderings can be extensively employed. The existence of all six indexes guarantees that every join between triple patterns can be done using merge joins. More expensive join algorithms are needed only when joining unordered intermediate results. In the H_2RDF+ case, we maintain both of these properties while moving towards a distributed and scalable environment. We move from local disk B^+-trees to distributed key-value tables (HBase) and from centralized to distributed, MapReduce-based, join processing.

2.3.1 HBase Indexes

HBase is a distributed, NoSQL key-value store that can handle large amounts of data using commodity machines. HBase tables are, in practice, range partitioned sorted key-value maps. In our system, we use an HBase table for each index. As HBase uses a key-value model, our indexes store all triples in keys and leave the values empty.

RDF triples contain long string URIs and literals that can add a lot of space overhead, especially in the case of multiple indexes. To achieve a space-efficient implementation, we use IDs instead of strings and keep two separate HBase tables that work as dictionaries to translate string values to IDs and vice versa. This mapping from string-based IDs to byte-based IDs is created during data import with respect to the occurrence frequencies of the string literals in the dataset: A very frequent predicate will get an ID with value close to zero. In order to take advantage of the frequency related IDs we apply byte-level, variable-length encoding when storing IDs in HBase. Variable length encoding leads to smaller byte representations for frequently used values and thus achieves high compression.

2.3.2 Index Statistics

Apart from the six indexes described above, we keep aggregated index statistics that can be used to estimate triple pattern selectivity as well as join output size and join cost. We have two categories of aggregated indexes:

1. With two out of the three triple elements bound, namely *sp_o, ps_o, po_s, op_s, os_p* and *so_p*. For example, the *sp_o* table contains a set of (subject, predicate, count) key-values, were the count represents the number of triples that contain the respective combination of subject, predicate.

2. With one bound element, namely *s_po, p_so, p_os, o_ps, o_sp* and *s_op*. For example the *p_so* index contains a set of (predicate, count, average) key-values, where count is the number of distinct subjects related to this predicate and average is the average number of objects related to each subject.

2.4 Bulk RDF data indexing using MapReduce

In this section, we thoroughly describe our MapReduce bulk indexing process that can handle the indexing of massive RDF datasets. It consists of four highly scalable MapReduce jobs that:

- Translate RDF literals to integer IDs with respect to the literal's occurrence frequency in the dataset. A very frequent predicate will get an ID with value close to zero. Both the String-ID and the ID-String dictionaries are stored in separate HBase tables. The frequency aware ID mapping in conjunction with our variable length encoding scheme for writing IDs inside our indexes achieves great reductions in storage space requirements.
- Generate and load HBase tables for all 6 RDF triple indexes along with their respective aggregated statistics.

In order to handle web scale RDF datasets our bulk indexing process needs to minimize the number of I/O and network operations and avoid unnecessary iterations over the RDF dataset. It also avoids the execution of HBase API calls for each tuple insertion; instead, bulk import MapReduce jobs directly create HFiles (the HBase file format) which are then loaded directly in HBase tables.

2.4.1 First MapReduce job

In the first MapReduce job each mapper reads one block of RDF triples and generates a sorted-map that contains all the unique string labels found in the file followed by a counter that represents the number of times each string label was found in the respective RDF block. At the cleanup phase of each mapper this sorted-map is used to produce the following information:

- For each RDF block, a file that contains all its distinct string labels is created. This file will be used in subsequent steps in order to efficiently retrieve the relevant IDs that are needed to translate all the triples in the block.
- Each mapper emits all the wordcounts of the sorted-map in order for the reducers to produce a global string label wordcount.
- We also use a sampling rate in order to sample the input triples and generate balanced partitions on both the distinct string label space and the indexing space (for all possible triple orderings). Concerning the distinct string label space for each sampled triple the mappers emit three special key-values one for each of its string labels(s, p, o). All the sample key-values are sent to a special reducer that is responsible for generating a load balanced partitioning of the string label space. At this moment, we cannot yet create the partitioning of the indexing space because we require the translated IDs that are not yet decided. Therefore, we just generate for each RDF block a sample file that contains its sampled triples.

The first job issues two types of reducers: 1)the first reducer which handles the sampled key-values and generates the string-label partition and 2)the wordcount reducers that sum up all local counters for each distinct label. Each wordcount reducer maintains a sorted list containing its wordcount key-values sorted by their counts. The reducers write their output at the cleanup phase and thus generate blocks of locally ordered, according to the count, key-values.

2.4.2 Second MapReduce job

The second job is responsible for giving globally unique and frequency aware IDs to all distinct string labels present in the dataset. The mappers of this job read the locally ordered, according to the count, wordcount output produced in the previous step. In order to avoid globally sorting all the distinct string labels according to their frequency count we assign IDs using a loose global order that requires no more communication information. The first MapReduce job utilized a HashPartitioner to split the labels between the reducers and therefore we can assume, with high probability, that all output blocks will contain labels from all the frequent and non-frequent classes of string labels. Taking advantage of this property we assign IDs using the following formula:

$$ID = \begin{cases} locID * R + redID, & \text{if } locID < min \\ offset[redID] + locID - min, & \text{if } locID >= min \end{cases} \tag{2.1}$$

where:
ID : is the global ID assigned to a label
$locID$: is the local ID inside each block that is assigned according to the local order of counts
min : is the minimum number of key-values produced by a wordcount reducer in the first MapReduce job
$redID$: is the ID of the reducer that produced the respective output block
$offset[]$: all reducer IDs will be interleaved until the minimum number of keys is reached. In order to avoid introducing holes to the assigned ID space we then start assigning contiguous ID regions to each reducer. This is achieved using the $offset$ table that contains the first ID of the respective contiguous region. The $offset$ is computed using the values $numKeys[redID] - min$ for each reducer.

Both the min number of output key-values and the $offset$ table can be easily computed by the output of the previous job. We can observe that this formula interleaves IDs between the wordcount blocks, generates a loose order of assigned global IDs and introduces no holes to the assigned ID space. We note here that by generating this loose ordering we avoid resorting and reshuffling the data and introducing unnecessary overhead.

Using the above formula, the mappers of the second job can assign independently the global IDs for each string label. For each string label the mappers emit two key-values in order to create both the string-to-ID and ID-to-string HBase dictionaries. This job also takes as input the distinct string label blocks generated in the first step. The mappers that read those files emit for each distinct label a key-value containing as key the label and as value the block ID.

To handle the two indexing spaces we use two separate partitioners for this job. The first partitioner is the string label partitioner computed in the previous step. The second partitioner handles the ID space and just splits it in continuous regions of a certain size. The load balancing of the ID space partitioner is achieved due to the fact that we used a contiguous ID space that contains no holes. We use two types of reducers for this job. The ID space reducers simply create the respective HFiles and directly load them to HBase tables. The string label reducers both generate the respective HFiles and also a file that contains for each string-ID pair a list of blocks that this is required. The second file is used in the following job to translate the distinct string label blocks.

2.4.3 Third MapReduce job

The third job handles the translation of the distinct string label blocks. It utilizes the files generated in the previous job. We assign one reducer to each RDF triple block. The job reads the files that contain for each string-ID pair a list of blocks that this is required and generates for each a key-value with key the block ID and value the string-ID mapping. Each reducer gets all the translations of an RDF triple block and just outputs them to an HDFS file.

2.4.4 Fourth MapReduce job

The last job parses the RDF triples again. First, each mapper reads the translation file for the corresponding RDF block and loads it into a memory hash map. It then parses the RDF triples, translates the string values and emits key-values for all 6 different orderings of the RDF triple. This job also requires the computation of a load balanced total order partitioner for the hexastore indexing space. Before starting the job, we translate the sample triple files created in the first job using our HBase dictionaries and generate a load-balanced partitioner for the indexing space. Each reducer of this job takes as input a sorted range partition of the indexing space and, while iterating over it, computes the aggregated statistics described above and creates the corresponding HFiles for both the primary and the aggregated indexes. The statistics maintained in the aggregated indexes, described in the previous section, can be easily computed while iterating over the sorted indexes and thus are computed without introducing additional network or I/O overhead.

2.4.5 Indexing storage space

H_2RDF+ utilizes an aggressive compression scheme for storing its indexes using: 1)variable length encoding for writing IDs in conjunction with frequency based String-ID mapping, 2)Google Snappy compression[3], also known as "Zippy" compression to further compress the resulting HBase tables. Our variable length encoding scheme is presented in Table 2.1 and can support IDs with up to 8 byte length.

Our encoding scheme uses variable length prefixes in order to encode the length of each ID. The specific selection of prefixes achieves the following objectives:

[3]`https://code.google.com/p/snappy`

Positive Prefix	Negative Prefix	Total Bytes	ID Bits
10******	01******	1	6+0=6
110*****	001*****	2	5+8=13
1110****	0001****	3	4+16=20
11110***	00001***	4	3+24=27
111110**	000001**	5	2+32=34
11111100	00000011	6	0+40=40
11111101	00000010	7	0+48=48
11111110	00000001	8	0+56=56
11111111	00000000	9	0+64=64

Table 2.1: *Variable length encoding scheme*

- Maintains the byte ordering property of the encoded IDs. This means that a raw byte comparator would order the variable length IDs in the same order as a value comparator. This property is really important because our indexes depend heavily on byte order.
- The variable length prefixes allow more IDs to be encoded with less bytes. A static prefix definition would require at least 5 bits to encode all the different cases. This means that only 2^3=8 IDs could be encoded using only one byte. However we can encode 2^6=64 IDs using only one byte. The same is true for all IDs that can be encoded with less than 5 bytes.

2.5 Join Execution Algorithms

Our work makes a twofold contribution relative to the join execution engine: We present a multi-way merge join algorithm and a sort-merge join algorithm, both executed over our distributed index. The former performs efficient joins over already sorted data (i.e., the HBase index tables); the latter performs joins when some of the data is unsorted (i.e., when intermediate results exist). The two algorithms can be executed in both distributed (via MapReduce) and centralized (over a single cluster node) mode.

2.5.1 MapReduce Merge Join Algorithm

This algorithm is designed to join multiple triple query patterns over the same variable. For example, suppose that we want to perform the following join on variable `department`:

select * where{
?person ub:memberOf **?department** .
?department ub:subOrganizationOf ?university .
?department rdf:type ub:Department .
}

We can get the triples ordered by `department` if we do the following three range scans: (for each range scan we specify the index table and the bound values in the respective order) {pos, ub:memberOf}, {pso, ub:subOrganizationOf}, {pos, rdf:type, ub:Department}. To execute the distributed merge join over those scans, we first specify the largest scan (i.e., the scan that spans the most HBase regions). We implement the merge join algorithm as a Map-only job over the regions of the largest scan. Each mapper processes a sorted partition of the scan (region), which translates to a sorted partition over the join variable's keys. The mapper has a local scanner over the large pattern and initializes the respective scanners over the other query patterns respecting the range of the the join variable's keys.

For example, let us assume that the largest pattern of the above join is the first containing two regions with the following join variable ranges: `[Dep0, Dep5)` and `[Dep5, Dep10)`. Note that we use string values here for readability; the partitions are in the ID space. The first mapper will initialize two scanners: {pso, ub:subOrganizationOf, `[Dep0, Dep5)`}, {pos, rdf:type, ub:Department, `[Dep0, Dep5)`} and merge join them with the local region scanner. The second mapper will handle the range `[Dep5, Dep10)` respectively.

2.5.2 MapReduce Sort-Merge Join Algorithm

This algorithm is only used when we join intermediate (thus unordered) results. It can take as input one or more intermediate results and one or more triple queries. For example, suppose that we want to perform the following join on variable `department`:

select * where{
?y **?department** ?w . (1)
?z **?department** . (2)
?person ub:memberOf **?department** . (3)
?department rdf:type ub:Department . (4)
}

The first two patterns present intermediate results that contain bindings for all the variables depicted in the pattern's name. These patterns are not ordered by the join variable. At first, we check the triple query scans (triple patterns (3) and (4)) and find the maximum partition of the join variable in the same way that we described above. The sort-merge join is executed as a MapReduce job that takes as input only the intermediate result patterns (triple patterns (1) and (2)). Each mapper reads bindings from the intermediate results and maps them using as key the binding of the join variable. The job uses the maximum join variable partition to produce a global ordering of the reduce keys. This means that each reducer will get a sorted range of the join variable's keys. The reducer initializes the index scans for its respective key range and then merges all intermediate and triple patterns by iterating over the sorted input. In case we need to join only intermediate results we utilize a hash partitioner and perform a hash join.

For example, let us assume that the largest pattern of the above join is, as before: `[Dep0, Dep5), [Dep5, Dep10)`. The first reducer will get *all* the intermediate bindings in the first range and will initialize two scanners: {pos, ub:memberOf, `[Dep0, Dep5)`} and {pos, rdf:type, ub:Department, `[Dep0, Dep5)`}. The reducer will iterate over all patterns and produce the join results. The same will happen with the second reducer over the second range.

2.5.3 Centralized Join Algorithms

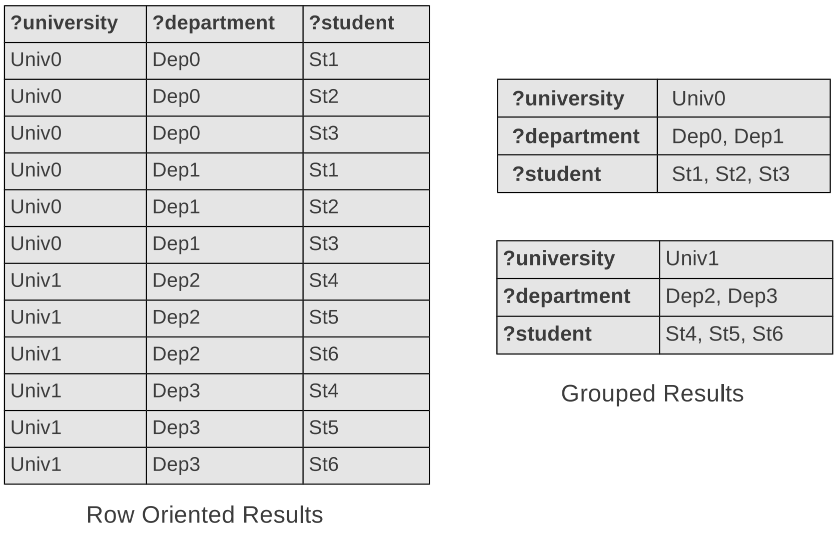
We also implement the classic versions of the merge and sort-merge join algorithms in a centralized environment. The only difference is that we use HBase scanners in order to iterate over the sorted relations rather than local B^+-tree or file scanners.

2.5.4 Intermediate results format

SPARQL queries involve multiple joins and feeding results of one join to the next. Intermediate results can become really large and grow exponentially with each subsequent join. This is why we need to have a space-efficient representation of the intermediate results. Standard row oriented databases create all result tuples at the end of each join. Instead, we opt for a lazy materialization of intermediate tuples and try to maintain grouped results as much as possible. Our lazy materialization maintains groups of bindings that contain: 1) a set of the names of variables contained in the result, 2) for each variable, a list of its bindings. The bindings contained inside a group must satisfy the property of all-to-all connection, i.e., the respective tuples can be materialized by a nested loop over all variables. As an example, suppose that we execute the following join:

select * where{
?department ub:subOrganizationOf **?university** .
?student ub:undergraduateDegreeFrom **?university** .
}

Our sorted indexes can retrieve all departments and students grouped per university. We need to exploit this grouping as much as possible in order to avoid generating all intermediate result tuples. Assume our database contains 2 universities, each having 2 departments and 3 students. The row-oriented results of the join are depicted in Fig. 2.2 (left). Instead of materializing all these combinations we store grouped results as depicted in Fig. 2.2 (right). Note that there is no explicit connection between students and departments (students and departments connect only with the university and not with each other), thus the all-to-all connection property applies. Extending our example with larger figures, if our database contains 100 universities, each of them with 30 departments on average and 100K students, a row-oriented scheme would create $100 \times 30 \times 100K = 300M$ result tuples by replicating a lot of times the IDs of universities and departments. To store these results, we would need to write three times as many IDs ($900M$). For the same example, our scheme would create 100 groups, one for each university, each group containing 30 bindings for the department variable and 100K bindings for the student variable. Thus, we would need to output $100 + 100 \times 30 + 100 \times 100K =$ 10,003,100 IDs which is orders of magnitude smaller that the previous requirement. We also apply byte level, variable length encoding on IDs and achieve a highly compressed output size.



?university	?department	?student
Univ0	Dep0	St1
Univ0	Dep0	St2
Univ0	Dep0	St3
Univ0	Dep1	St1
Univ0	Dep1	St2
Univ0	Dep1	St3
Univ1	Dep2	St4
Univ1	Dep2	St5
Univ1	Dep2	St6
Univ1	Dep3	St4
Univ1	Dep3	St5
Univ1	Dep3	St6

Row Oriented Results

?university	Univ0
?department	Dep0, Dep1
?student	St1, St2, St3

?university	Univ1
?department	Dep2, Dep3
?student	St4, St5, St6

Grouped Results

Figure 2.2: *Grouped intermediate results*

As stated before, groups are split on demand according to the sequence of joins. For example, lets assume that we want to use the above results in the following join:

select * where{
?department ?university ?student .
?professor ub:worksFor **?department** .
}

This join, on variable department, is executed using the sort merge join algorithm described in the previous section. In Fig. 2.3 we can see how we use the grouped results in the join procedure. Initially, in the map phase, we split the group according to the join variable, thus we create one group for each department. Note that the map output is not split across the student bindings because those bindings maintain the all-to-all connection with the rest bindings. In the reduce phase groups of professors per department are retrieved from the index and are merged with the inputs to form the output groups.

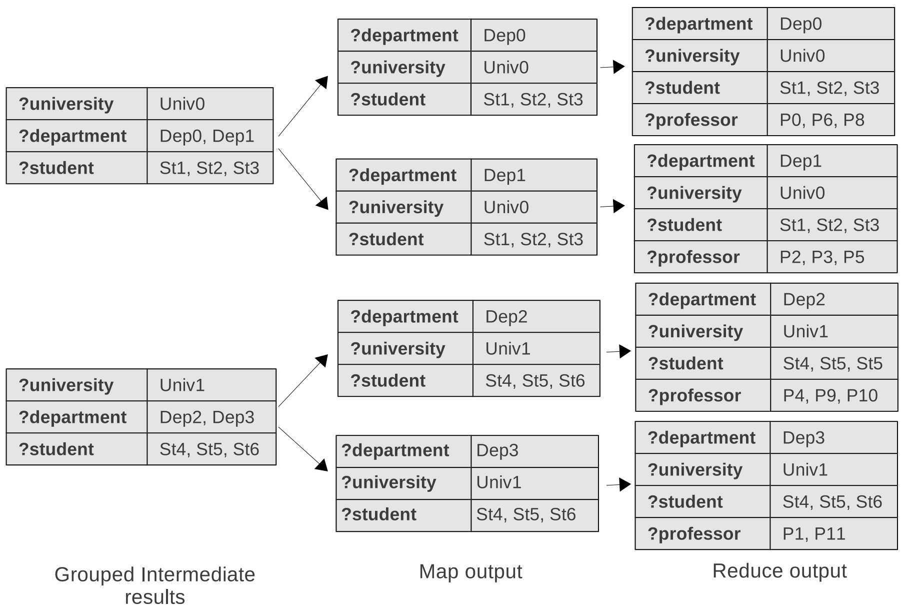


Figure 2.3: *Join on grouped intermediate results*

2.6 Query Planning and Execution

Deciding on the query execution plan is an important aspect that greatly influences performance, since SPARQL queries usually require multiple joins on different variables. The H_2RDF+ planner decides on the execution order of the different joins so as to minimize the total query execution time. To find the optimal join order we have to consider the different combinations in which the joins can be performed. Obviously, the number of choices grows exponentially to the number of joining variables, making the problem computationally expensive. Instead, we use a greedy, cost-based, online planner that decides on the join that must be executed in every step of the query. To derive the costs of possible joins we devise a detailed join cost model that takes advantage of our stored statistics. Our cost model can be also used to help the planner decide on whether the join will be executed in a centralized or a distributed fashion. The incentive behind this decision is that distributed MapReduce jobs cannot offer real-time response times for small joins and are beneficial only in case of large joins. In this section we present the join cost model as well as our greedy join planner.

2.6.1 HBase scan performance evaluation

Our join execution heavily depends on HBase scans and thus in order to derive an applied cost model we need to stress-test their performance. The key parameters of a scan are the seek latency and the read throughput. After doing some experiments on scanning our indexes we found out that a seek operation takes on average 16ms and the average read throughput reaches 400,000 key-values(triples)/second. Detailed performance evaluation for those features can be found in Section 2.7.4. These values are infrastructure specific and can change across different installations but they can be estimated by a simple benchmarking test that runs once for every different installation.

We integrate this performance knowledge into our merge join algorithm in order to make it more efficient. Except from sequentially scanning the input relations a merge join algorithm may need to jump forward on one relation if we know that there are no possible join results in this range. In this case we need to take the decision of whether to seek to the next position by initializing a new scanner or read all intermediate values sequentially. From the above metrics we can easily note that the time needed for a seek operation is equal to the time needed to sequentially read nearly 6,400 key-values. Thus the merge join algorithm uses the seek operation only if it is expected to discard more than 6,400 key-values.

2.6.2 Merge join cost model

The merge join algorithm is controlled by its input triple queries(Q). The total cost of the join (in terms of completion time) is:

$$MJcost(Q) = \sum_{i \in Q} ReadKeys(Q,i)/thr \tag{2.2}$$

$$ReadKeys(Q,i) = \min\{(\min_{j \in Q} n_j) \cdot o_i \cdot SeekOverhead, n_i o_i\} \tag{2.3}$$

n_i: number of join variable's bindings for the i^{th} query.
o_i: average bindings of the non-joining variables corresponding to one join variable binding. Refers to the i^{th} query.
thr: the scan throughput discussed earlier.
SeekOverhead: the seek overhead (6,400 key-values)
ReadKeys(Q,i): the number of key-values that will be read from the i^{th} query.

The cost of the merge join algorithm depends on the number of key-values that need to be read. To estimate this size we first find the minimum number of input join keys among the joining queries. A merge join algorithm would need to read at most that amount of keys from each relation using seeks to pass over irrelevant keys. As stated before we use an heuristic to decide whether to perform a seek operation and thus in the worst case scenario our merge join algorithm would always seek paying each time the *SeekOverhead*.

2.6.3 Sort-Merge join cost model

In this algorithm we have to join both a set of input scans(Q) and a set of intermediate results(I). The total cost of the join (in terms of time) is:

$$SMJcost(Q, I) = (2 \sum_{i \in I} n_i o_i + \sum_{i \in Q} ReadKeys(Q \cup I, i))/thr \tag{2.4}$$

The cost of the sort-merge join algorithm is divided in two main parts. The first part is the cost of joining the intermediate results. The intermediate patterns are read twice, once in the map and once in the reduce phase. For the triple queries we use the same estimation described in the above section.

2.6.4 Join Planner

The cost model described in the previous section is a step towards finding the optimal join execution plan, i.e., the join order with the minimum total execution cost. Our planner uses a greedy algorithm that in each step of the execution selects the smallest cost join to be executed.

Algorithm 1: H$_2$RDF+ PLANNER

```
1: $V \leftarrow \{v_1, v_1, \ldots, v_n\}$ //join variables
2: $TQ \leftarrow \{tq_1, tq_1, \ldots, tq_m\}$ //triple queries
3: //$TQ(v)$ triple queries that contain variable $v$
4: while $V \neq empty$ do
5:    $Jstruct \leftarrow empty$ //Join's required information
6:    $v_{join} \leftarrow min_{v_i \in V}\{Greedy(v_i, TQ(v_i))\}$
7:    $Jstruct.addJoin(v_{join}, TQ(v_{join}))$
8:    $V.remove(v_{join})$
9:    $TQ.remove(TQ(v_{join}))$
10:   if $Jstruct.executionType() = MR$ then
11:      $executeMapReduce(Jstruct)$
12:   else if $Jstruct.executionType() = Cent$ then
13:      $executeCentralized(Jstruct)$
14:   end if
15: end while
```

Our greedy join planner is presented in Algorithm 1. Set V contains all the variables that need to be joined in order to answer the query. Set TQ contains all the triple queries that need to be joined. While V contains more variables we need to execute more joins. Using our greedy function we select the most beneficial variable to be joined. The selected variable is fully joined in the current job (multi-way join), which means that all its queries are joined and we remove it from V.

Our greedy function is presented in Algorithm 2. This function checks if the join requires a merge or a sort-merge join algorithm and then computes the costs of executing the join in centralized or distributed manner. The centralized cost is the cost described in the previous section. The distributed MapReduce cost is computed by dividing the centralized cost by the minimum of partitions and number of mappers in the cluster. This number is the maximum amount of parallelism that will be present when executing the distributed job. We also add an overhead called *MRoverhead* which is the amount of time required to setup a MapReduce job. A MapReduce job with no input data needs at least 30 sec to finish. Thus our incentive is to use centralized jobs when quick responce times can be achieved and leverage the parallelism of distributed execution only when we face large joins.

Algorithm 2: $Greedy(v, TQ)$

```
1:  //$TQ$ contains the triple queries to be joined
2:  //Split $TQ$ in scans and intermediate results
3:  $(Q, I) \leftarrow splitPatterns(TQ)$
4:  if $I \neq empty$ then
5:      //Sort-merge join
6:      $cost \leftarrow SMJcost(Q, I)$
7:  else
8:      //Merge join
9:      $cost \leftarrow MJcost(Q)$
10: end if
11: //Compute MapReduce Cost
12: $MRcost \leftarrow cost/\min(patitions, mappers) + MRoverhead$
13: if $cost < MRcost$ then
14:     $Jstruct.addExecutionType(Cent)$
15:     return $cost$
16: else
17:     $Jstruct.addExecutionType(MR)$
18:     return $MRcost$
19: end if
```

2.6.5 Elastic Execution

In this section, we focus on the resource adaptivity properties offered by our system. H_2RDF+ is able to decide, on the fly, on the number of resources required to process each join in hand. In effect, it is able to automatically estimate the amount of required resources, be they threads (centralized case) or map/reduce tasks (distributed case), on a per-join basis. The adaptive decisions for each join are done during runtime and they scale according to the estimation of the join cost.

For small join costs we use centralized execution and scale the resources using a different number of concurrent threads. Both our merge and sort-merge join algorithms can be executed in parallel by partitioning the join variable key space. Utilizing the statistics held in our aggregated indexes we can estimate the number of join variable bindings contained in each of the joined relations. We use the estimation of the maximum number of bindings contained in a join relation to decide at runtime how many threads will be launched. We then greedily split the join variable key space and assign work to different execution threads. We launch a thread only if it is estimated to process more than a minimum amount of input bindings. We also pose a limit to the number of concurrent threads in order to avoid costly context switching between them.

Larger joins are executed using distributed MapReduce jobs. The resources required for the MapReduce execution also scale with the join cost. The resources available on a MapReduce cluster are the number of concurrent mappers and reducers. Assuming these are set for a specific cluster, we want to occupy only the number of mappers and reducers required for the execution of each join. The cost of a MapReduce join is proportional to its input data. As discussed in [Papailiou 13], input data are split in HBase regions, each region having a configured size. In our implementation, every map task handles one HBase region and thus the region size is configured to contain the amount of data required to amortize the initialization overhead of launching the task. If the region had less data, the initialization overhead of a map task would be greater than the actual processing of the data. Therefore, the number of resources occupied is proportional to the size of input and, by extension, to the join cost. If the map tasks launched by the join are less than the cluster's concurrent mappers (which we anticipate to be the case for less costly joins on medium to large size clusters), the remaining resources will be proportionally allotted to other users' joins. In the opposite case, all the mapper slots and the cluster resources will be occupied.

2.7 Experiments

In this section we present a thorough performance evaluation of the H_2RDF+ system.

2.7.1 Cluster configuration

Our experimental setup consists of an OpenStack private cluster of 6 VM containers. Each container has a 2×6-core Intel Xeon®CPUs at 2.67GHz, 48 GB of RAM and two 2TB disks setup with RAID 0. Worker VMs feature a 2-virtual core processor, 4GB of RAM and 300GB of storage space, allowing the cluster to support a total of 36 VMs. The clusters we use for our evaluation consist of variable numbers of VMs (10 to 35) plus a single VM in the role of the HDFS, MapReduce and HBase master. Each worker VM runs 2 mappers and 2 reducers, each consuming 512MB of RAM. We utilized Hadoop v1.1.2 and HBase v0.94.5 respectively.

2.7.2 Compared Systems

We compare the performance of H_2RDF+ against three state-of-the-art RDF stores: RDF-3X [Neumann 10a], HadoopRDF [Husain 11] as well as the first version of our distributed system H_2RDF [Papailiou 12]. We evaluate the latest version (v0.3.7) of RDF-3X [Neumann 10a, Neumann 10b]. HadoopRDF was built using the latest SVN rev. 158 from the project repository.

All the above systems process queries using dictionary IDs rather than strings and URIs. We have observed that the last step of translating query result IDs to strings is a challenging task for all compared triple stores. In some cases, it requires time comparable or even larger than the actual processing. In this paper, we focus on the join execution engine. Thus, in order to provide a fair comparison, we have also removed the translation task from all the compared systems.

2.7.3 Data Sets Used

To test the system under web-scale, realistic conditions we utilize two datasets. The Yago2 dataset [Hoffart 11] consists of real data gathered from various resources such as Wikipedia, WordNet, GeoNames, etc, and contains more than 120 million triples. This dataset is relatively small; we use it to show that distributed query execution can perform better even for small datasets when large non-selective queries are required. The LUBM dataset generator [Guo 05] creates datasets with academic domain information, enabling a variable number of triples by controlling the number of *university* entities. By varying this parameter between 1K to 100K, we create datasets ranging from 1.4 million (25GB) to 13.8 billion triples (2.5TB). This dataset is widely used to compare performance of triple stores especially when arbitrarily large datasets are required. Lehigh university has also published a suite of test queries[4] that offer a good mixture of SPARQL queries.

2.7.4 Index comparison

In this section we evaluate the performance of our indexing scheme. Initially, we consider space requirements. As mentioned in Section 2.3.1, H_2RDF+ uses an aggressive compression scheme using variable length encoding and smaller IDs for frequent string values. We also compress our index tables using the Google Snappy compression[5] also known as “Zippy” compression.

[4]`http://swat.cse.lehigh.edu/projects/lubm/queries-sparql.txt`
[5]`https://code.google.com/p/snappy`

We choose the Snappy library because it offers very high decompression speed and reasonable compression. Snappy's CPU-efficient decompression algorithm makes it a perfect candidate for NoSQL stores by exploiting the trade-off between I/O and CPU bandwidth.

Dataset	**Raw Size**	**RDF-3X**	**H_2RDF**	**H_2RDF+ (no Snappy)**	**H_2RDF+**
LUBM1k	28 GB	9 GB	25 GB	27 GB	7 GB
LUBM10k	276 GB	77 GB	214 GB	241 GB	62 GB
LUBM20k	549 GB	156 GB	529 GB	545 GB	121 GB
Yago2	26 GB	12 GB	33 GB	35 GB	10 GB

Table 2.2: *Comparison of storage requirements*

In Table 2.2 we register the storage requirements of the compared systems for the LUBM and Yago datasets. The "Raw Size" column contains the size of the dataset serialized using the *N-Triples* format. Although storing 6 rather than 3 indexes and more detailed statistics, H_2RDF+ manages to have smaller space requirements than its previous version due to: 1) the smaller ID values, as H_2RDF uses the 8-byte MD5-hash of the string values, 2) the byte-level variable length encoding in conjunction with the frequency-aware ID mapping, 3) the block level Snappy compression. RDF-3X also offers a highly compressed storage scheme due to its gap compression [Neumann 10a] (stores only the difference between subsequent triples in the index). The difference between the storage requirements of RDF-3X and H_2RDF+ results mainly from the frequency-aware ID mapping and the block-level Snappy compression used in H_2RDF+ (achieves $\sim$70% storage reduction).

	RDF-3X	**H_2RDF**	**H_2RDF+**
Local Scan Throughput (million triples/sec)	17	0.73	1.13
Remote Scan Throughput (million triples/sec)	-	0.2	0.4
Seek latency cold cache (ms)	1	86	16
Seek latency hot cache (ms)	0.2	17	7

Table 2.3: *Comparison of scan throughput and seek latency*

We also study the retrieval efficiency of the indexes and their respective technologies. As mentioned in Section 2.6.1, scan throughput and seek latency are very important metrics that need to be optimized and evaluated. From Table 2.3, we deduce that our new indexing scheme achieves substantial improvements in all categories compared to our previous one. We notice a 54% improvement in local (the client is in the same host with the HBase server responsible for the data) scan throughput and 100% improvement in remote scan performance (the client

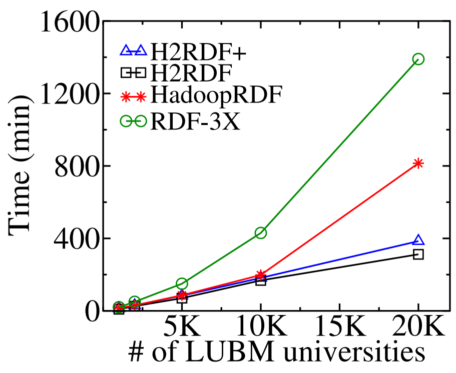


Figure 2.4: *Import scalability versus dataset size*

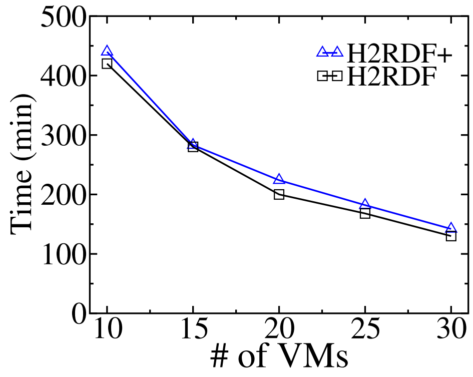


Figure 2.5: *Import scalability versus cluster resources*

scan data from a remote HBase server). We also greatly reduce the latency of a seek operation due to the more compact representation of HBase key-values.

Compared to RDF-3X, scan/seek times are almost an order of magnitude larger. RDF-3X maintains extremely efficient clustered $B^{+}trees$ that are placed in local disk storage. Our indexes suffer from retrieval overheads related to the distributed architectures of both HBase and HDFS. This performance overhead is alleviated by the capability of distributed, concurrent scanning inside MapReduce jobs.The impact of using distributed indexes will be made visible in the next section in the case of small, selective queries; the effect disappears when processing distributed, non selective joins.

2.7.5 Data Import

In this section we evaluate the performance of the H_2RDF+ bulk indexing process. H_2RDF+'s indexing process is consisted of 4 MapReduce jobs that materialize all 6 combinations of RDF indexes. To test the efficiency of our indexing method we compare it using RDF-3X, HadoopRDF as well as the first version of our distributed system H_2RDF.

We first test the scalability properties of all the compared systems regarding the RDF dataset size. To do so we utilize the LUBM dataset generator that can generate RDF datasets with variable size. We import LUBM datasets containing 1K to 20K universities, i.e., 0.14 to 2.7 billion triples (28 to 549GB of data respectively). H_2RDF+, H_2RDF and HadoopRDF were executed using a cluster of 25 worker and 1 master nodes, while RDF-3X uses a 4$\times$16-Core server with 128 GB RAM and 1TB disk. Total import times are presented in Figure 2.4. This is the time needed for all systems to load the full dataset according to their indexing scheme.

RDF-3X, being a centralized system, parses all triples sequentially in order to create its indexes. It doesn't exploit the parallelism capabilities offered by the modern multicore architecture of CPUs. It also reads the input data several times in order to generate the various different orderings of the triples. This iterative scan of input data results in an increasing import complexity. As we can see in Figure 2.4, RDF-3X introduces the slowest, among the compared systems, import times for loading RDF datasets.

HadoopRDF needs to execute four different MapReduce jobs which take as input the whole dataset. This means that it needs to scan the data four times resulting in low import performance. Additionally, some of these jobs do not equally partition the reduce input data and thus overload some reducers while leaving others idle. The load balance between the available computing resources is one of the most important properties that need to be handled in order for distributed systems to offer good scalability properties. We can observe that HadoopRDF, while materializing only one ordering of the triples in raw HDFS files, requires $2\times$ more time than H_2RDF+ for loading the LUBM20k dataset.

We also compare H_2RDF+ to our previous H_2RDF system. H_2RDF used 2 MapReduce jobs to materialize 3 of the 6 RDF triple indexes. The first job was a sampling job that created a load balanced partitioning of the indexing space, while the second one used the partitioning to generate and load the 3 materialized HBase RDF indexes. In Figure 2.4 we can observe that H_2RDF+ manages to import 3 additional indexes and keep more detailed statistics than H_2RDF at a mere 10-20% overhead. This is mainly attributed to:

- Our optimized indexing procedure that minimizes the times that the raw dataset is read. While requiring 4 Map-Reduce jobs to import the dataset, only 2 of them read the raw dataset while the rest process output files that are quite smaller than the original dataset. This greatly reduces the I/O time needed for executing our import process.
- The aggressive compression used in all the indexing steps. We use our variable length encoding to write all intermediate and final results of our indexing process and thus save both storage space and I/O time.
- The extensive use of sampling to generate load balanced partitions for all our MapReduce computation steps.

Another important point is the indexing scalability to the number of available computing resources. We import the LUBM10k dataset (1.3 billion triples) using clusters with different number of worker nodes. We use clusters with 10, 15, 20, 25, 30 worker nodes. The corresponding results are presented in Figure 2.5. We can observe that H_2RDF+ manages to maintain the scalability properties of H_2RDF while introducing a more complex RDF indexing process. Using the MapReduce framework we can gain almost linear speedup when we increase the number of worker nodes: At 10 workers we achieve an import speed of 49 Ktriples/sec, while using 30 nodes we almost triple the import speed at 142 Ktriples/sec. We also observe that H_2RDF+ introduces only a small time overhead (10-20%) compared to H_2RDF in all tests.

2.7.6 Direct Comparison

	Yago2		
	H_2RDF+	H_2RDF	HadoopRDF
Import(min)	31	26	72
YQ1(sec)	0.9	0.9	52
YQ2(sec)	1.5	1.7	68
YQ3(sec)	154	952	1832
YQ4(sec)	87	728	1495
	LUBM10k		
	H_2RDF+	H_2RDF	HadoopRDF
Import(min)	182	168	198
LQ1(sec)	0.6	0.6	152
LQ3(sec)	0.8	0.8	231
LQ4(sec)	2.1	2.4	1289
LQ2(sec)	95	635	915
LQ9(sec)	151	787	1488
	LUBM20k		
	H_2RDF+	H_2RDF	HadoopRDF
Import(min)	385	312	815
LQ1(sec)	0.8	0.8	378
LQ3(sec)	0.9	1	449
LQ4(sec)	2.3	2.4	2650
LQ2(sec	131	880	1367
LQ9(sec)	292	1034	2933
	LUBM100k		
	H_2RDF+	H_2RDF	
Import(min)	1154	985	
LQ1(sec)	0.8	0.9	
LQ3(sec)	1.1	1.1	
LQ4(sec)	2.4	2.5	
LQ2(sec)	412	1853	
LQ9(sec)	890	2761	

Table 2.4: *Performance comparison of H_2RDF+, H_2RDF and HadoopRDF for LUBM and Yago2 datasets*

In order to provide a direct, fair comparison among the different systems, we first test the performance of H_2RDF+ versus the other distributed systems. We utilize four datasets, namely LUBM with 10k, 20k and 100k universities and Yago2, consisting of 1.3 billion, 2.7 billion, 13.8 billion and 120 million triples respectively. H_2RDF+, H_2RDF and HadoopRDF were executed using a cluster of 25 worker and 1 master nodes. In Table 2.4 we register the data import times and response times for the selected queries. For a fair comparison to the centralized RDF-3X, we run both systems using the same *total* amount of resources: For RDF-3X, we use two single-server configurations (a 2×Quad-Core with 8 GB RAM, 8 GB swap and 1TB disk and a 4×16-Core with 128 GB RAM and 1TB disk); for H_2RDF+, we use as many worker VMs as the corresponding RDF-3X's server capacity allows. These results are reported in Table 2.5.

	Yago2			
Resources	8CPU/8GB RAM		64CPU/128GB RAM	
	H_2RDF+	RDF-3X	H_2RDF+	RDF-3X
Import(min)	164	157	26	149
YQ1(sec)	0.9	0.7	0.9	0.7
YQ2(sec)	1.5	1	1.6	0.9
YQ3(sec)	241	3037	138	1929
YQ4(sec)	123	2973	79	2068
	LUBM10k			
Resources	8CPU/8GB RAM		64CPU/128GB RAM	
	H_2RDF+	RDF-3X	H_2RDF+	RDF-3X
Import(min)	912	605	162	576
LQ1(sec)	0.6	0.4	0.6	0.3
LQ4(sec)	2.1	0.8	2.1	0.7
LQ2(sec)	373	2297	89	1277
LQ9(sec)	411	68	141	51
	LUBM20k			
Resources	8CPU/8GB RAM		64CPU/128GB RAM	
	H_2RDF+	RDF-3X	H_2RDF+	RDF-3X
Import(min)	2075	1526	349	1398
LQ1(sec)	0.8	0.4	0.9	0.4
LQ4(sec)	2.3	0.8	2.2	0.8
LQ2(sec)	706	Failed	119	2065
LQ9(sec)	753	Failed	264	289

Table 2.5: *Performance comparison of H_2RDF+ and RDF-3X*

Query Performance: LUBM provides a SPARQL query benchmark. The compared systems do not support OWL reasoning so we only test queries that do not require reasoning or in some cases (e.g., query LQ9) we remove the hierarchy of the `rdf:type` predicate by querying for explicit types with no subclasses. We show results for five such queries (identified as LQ) that provide a good mixture of both simple and complex structures. The selected set covers all variations of the LUBM test queries; moreover they are able to highlight the different decisions and characteristics of H_2RDF+ and the compared systems. Yago2 does not provide benchmark queries; Relative to the LUBM queryset, we have created a set of representative test queries. In detail, there are two main categories of SPARQL queries tested: the ones that contain some selective pattern and have small number of results (LQ1, LQ3, LQ4, YQ1, YQ2) and the ones that contain no selective patterns and represent more complex join structures (LQ2, LQ9, YQ3, YQ4).

H_2RDF+ performs noticeably better in queries with large input: We exploit the orderings provided by our indexes via the distributed Merge and Sort-Merge join algorithms and achieve almost 7× performance gain compared to H_2RDF and 10× compared to HadoopRDF. We also outperform RDF-3X in most of the complex queries both when running on the small and the large server configuration. For example, for LQ2, RDF-3X requires almost 12GB of memory to execute the query for LUBM10k and proves 6× slower than H_2RDF+ in the small server setting. For LUBM20k (and large server setting) this increases to 14× slowdown compared to H_2RDF+. Our system achieves 3–6× smaller response times when moving to a larger cluster, while RDF-3X's speedup is mainly attributed to the bigger amount of memory (no swapping). In LQ9, RDF-3X manages to perform better, as it loads query data in memory. Yet, this approach does not scale; the system runs out of memory for LUBM20k on the small server.

For small, selective queries H_2RDF+ uses centralized execution and manages to obtain performance comparable to RDF-3X. The difference in performance is mainly attributed to the lower scan throughput and the higher seek latency provided by our distributed HBase indexes. We also note that there is a small improvement compared to H_2RDF due to the more optimized indexing scheme. We also note that HadoopRDF has really poor performance for all selective queries due to the fact that it only executes MapReduce joins that process all input data and cannot take advantage of query selectivity.

From these results, we deduce that H_2RDF+ processes all query types according to the goals set in its design: It manages to correctly identify selective vs. non-selective joins, performing either distributed or centralized joins, each join being performed in the most advantageous strategy. In high-selectivity queries, it is almost as efficient as RDF-3X, with a small difference (few tenths of a second) due to the fact that our index is shared across multiple cluster nodes. This small performance difference is alleviated by our system's ability to serve multiple concurrent queries (see Section 2.7.8). For more data-intensive queries, it proves greatly superior to both central solutions and competitive Hadoop-based schemes due to both our join strategy and the ability to group multiple bindings.

LUBM full scale evaluation: Table 2.4 also contains H_2RDF+ and H_2RDF import and query execution times for the LUBM100k dataset that consists of 14 billion triples (2.5 TB), using a cluster of 1 master and 35 worker nodes. Our system achieves an import speed of 202 Ktriples/sec, a state-of-the-art performance according to [W3C 15]. Query response times follow the trend described in the previous experiments: For selective queries, centralized joins are selected, resulting in times that range between 0.8 and 2.4 sec. For non-selective queries with huge input sizes, such as LQ2 and LQ9, it achieves 3–4 times smaller response times compared to H_2RDF.

2.7.7 Join algorithm comparison

In this section, we compare the performance of our join algorithms over joins with different input sizes. In order to test the scalability of our algorithms we generate the following benchmark setup: We use a cluster of 25 VMs and the `ud:takesCourse` property from the LUBM20k dataset which contains 515 million triples that describe connections between students and courses. We randomly sample the corresponding data using variable sampling rates and store the sampled triples in a new HBase index. Figure 2.6 shows the execution times required to join the full `ud:takesCourse` relation with the sampled one using different join algorithms. We range the sampled triples from 5 to 500 million.

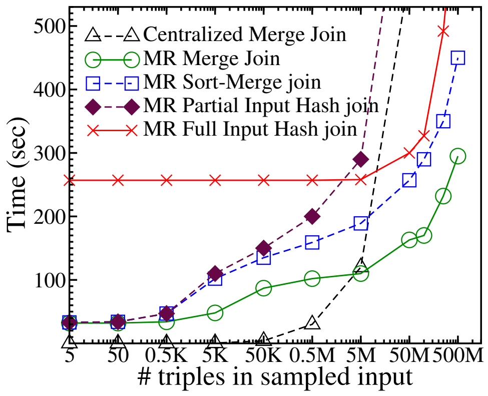


Figure 2.6: *Join algorithm scalability*

We notice that for joins that contain one selective input triple pattern, the most efficient join strategy is the centralized Merge join algorithm. This is because MapReduce joins always incur an initialization overhead of almost 30 seconds. The performance of the centralized join deteriorates with the input size due to the fact that the algorithm does not exploit the parallel scanning capabilities of our distributed indexes.

Relative to MapReduce-based join algorithms, we consider the Merge, Sort-Merge, Partial Input Hash and the Full Input Hash [Papailiou 12] join algorithms. We can clearly note that the Merge join algorithm has the best scalability performance due to the fact that it performs the join on sorted relations and minimizes the overhead of data movement. But this algorithm cannot be executed on intermediate, non-sorted relations. In this case, we can see that the Sort-Merge join proves to be the most scalable join algorithm. The difference between the Sort-Merge and the Partial Input Hash joins is the MapReduce partitioning method. The Sort-Merge join partitions the input data using a total order partitioner that takes advantage of the sorted indexes while the Partial Input join partitions using a Hash partitioner. This has impact on the reduce phase of the join: The Sort-Merge join performs a scalable merge join in the reduce phase while the Partial Input join executes a random HBase `get` on the indexes for each key. The second approach proves not scalable when the small input increases in size. Lastly, in the case that we have no sorted-indexed relation in the join, we need to fall back to the Full Input Hash join.

2.7.8 Concurrent execution of selective queries

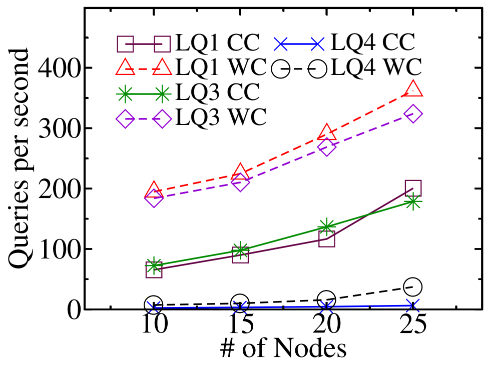


Figure 2.7: *Query throughput scalability*

For the case of centralized joins, we show that concurrent execution can result in very large query throughput. To achieve this, H_2RDF+ utilizes a zookeeper quorum that is responsible for the distribution of centralized joins to the cluster nodes. Each node has a maximum capacity of joins that can be simultaneously processed, set to 4 in our experiments. All tests are executed using the LUBM5k dataset. We use 10, 15, 20 and 25 worker nodes to see the impact of increasing the cluster size on the execution throughput. Results for queries LQ1, LQ3 and LQ4 are presented in Figure 2.7. We present the average query throughput in queries per second. We run the same test twice to get the cold(CC) and warm cache(WC) throughput. We do not implement any special caching scheme but rely on HBase's caching. We notice that the warm cache execution results in 2 to 3 times higher throughput compared to the cold cache execution which means that our system can take advantage of caching. We observe an almost linear throughput increase to the number of worker nodes: For example LQ1 has a throughput of 65 q/sec (a 15.4 ms per query) in a 10-node cluster (40 executors). This is a speed-up of 40× as the individual execution of LQ1 takes 0.6 sec. LQ1 and LQ3 have almost the same performance due to their similar execution cost. H_2RDF+ needs 0.6 and 0.7 sec to answer LQ1 and LQ2 respectively. As for the smaller throughput of LQ4, this is due to its increased execution cost, as it needs approximately 2 sec to be answered. LQ4 exhibits the same scalability and warm cache properties discussed previously.

2.7.9 Query Scalability

In this section we evaluate the scalability properties of our distributed query processing. We use LQ9 because it is one of the most complex queries tested, requiring three distributed joins. We test query execution scalability using different dataset sizes and number of worker VMs. The scalability results for LQ9 are presented in Figure 2.8. We test the performance of the MapReduce join execution using different dataset sizes using a 25-node cluster. The input and result size of LQ9 depends on the dataset size (directly affecting LQ9's execution time as well). Another parameter tested here is the region size effect on the MapReduce join execution. Large regions exhibit lower performance for small datasets because the number of tasks created fail to fully utilize the cluster resources. For larger datasets, all region sizes achieve good performance, as a result of having enough regions to fully utilize the cluster. For smaller region sizes (64MB or 32MB) the complexity is almost linear to the size of the input data.

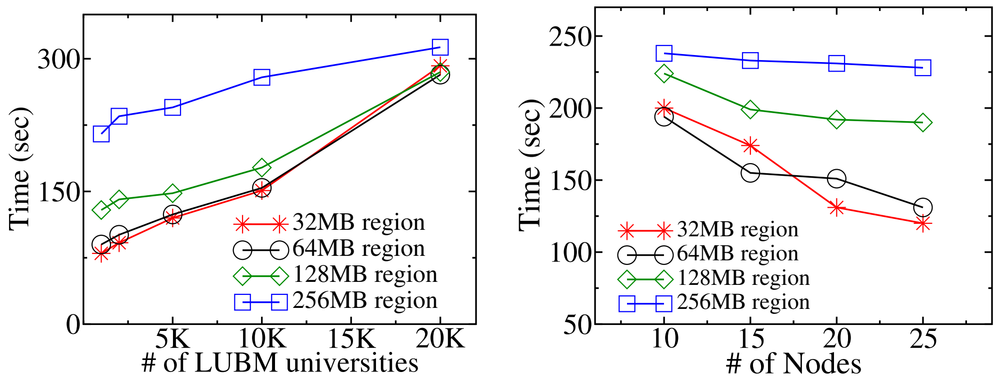


Figure 2.8: *Distributed query scalability for different number of universities and nodes for the LQ9*

Figure 2.8 also shows the LQ9 query execution time as the number of nodes increases. All tests are executed using the LUBM5k dataset. We vary the cluster size from 10 to 25 worker nodes and we vary the maximum region size from 32MB to 256MB. In the 32MB case, the join execution is highly scalable, gaining great speedup by adding more nodes. The deviations from linear speedup are mainly caused by the fact that the number of map tasks may or may not fit well to the cluster's capacity. As the region size grows, we note that adding more worker nodes does not significantly affect the speedup due to the fact that larger region sizes incur fewer tasks which cannot fully utilize cluster resources.

2.7.10 Join Planner and Elastic Execution

In this section, we compare the performance of our join algorithms and test our planner's decisions over different input queries. Moreover, we demonstrate the adaptive execution properties of our system. In order to test the scalability of our algorithms we use the sampling-based join experiment presented in Section 2.7.7. Figure 2.9 shows the ammount of computing resources occupied by our join algorithms to execute a merge join of the full `ud:takesCourse` relation of the LUBM1k dataset(25 million triples) with the sampled ones using different join algorithms. We range the sampled triples from 10 to 500 million.

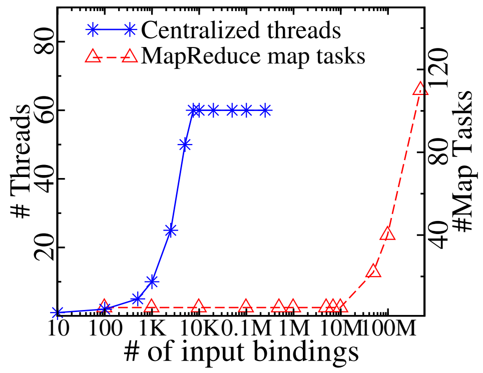


Figure 2.9: *Adaptive query resource allocation*

Our merge join algorithm can be executed either in a centralized or a distributed environment. In Figure 2.9 we depict the number of dedicated resources for every join execution. We can see the amount of resources committed to the join scale proportional to the join cost. For centralized joins, the planner scales the number of concurrent threads while for MapReduce joins it scales the number of mappers. For our cluster configuration, threads were launched only when they had a minimum amount of 100 binding to process. We also set the maximum amount of concurrent threads to 60.

Regarding distributed joins our join planner can decide on the fly for the amount of map tasks required to execute the join according to its cost. This is done by finding the maximum join input scan, i.e. the join relation scan that spans the most HBase regions. When finding the maximum scan, one map task is assigned to each of its regions. As the size of the sampled triples range from 10 to 500 million triples the size of the largest scan varies. When the sampled triples are less than the 25 million of the `ud:takesCourse` relation the maximum scan is the `ud:takesCourse` relation of LUBM1k that spans 4 regions and thus 4 map tasks are launched. When the sampled triples get bigger they become the larger scan and then on the utilized map tasks scale proportionally to their size.

CHAPTER 3

SPARQL Query Caching

3.1 Introduction

The schema-free nature of RDF data allows the formation of a common data framework that facilitates the integration of data originating from different application, enterprise, and community boundaries. This property has led to an unprecedented increase in the rate at which RDF data is created, stored and queried, even outside the purely academic realm[1] [2] and consequently to the development of many RDF stores [Weiss 08, Neumann 10a, Zeng 13, Atre 08, Papailiou 13, Bonstrom 03] that target RDF indexing and high SPARQL querying performance.

However, the move from the schema-dependent SQL data to the schema-free RDF data has introduced new indexing and querying challenges and made a lot of the well-known relational database optimizations unusable. In fact, RDF databases assume limited knowledge of the data structure mainly in the form of RDFS triples [Brickley 14]. However, RDFS information is not as rich and obligatory as the SQL schema; it can be incomplete and change rapidly along with the dataset. Therefore, most RDF databases are targeting the indexing of individual RDF edges resulting in query executions with much more joins than processing the same dataset in a schema-aware relational database. In contrast, RDF databases that use RDFS information to store and group RDF data [Stuckenschmidt 04, Tran 10], fail to effectively adapt to schema

[1]http://www.bbc.co.uk/blogs/bbcinternet/2012/04/ sports_dynamic_semantic.html
[2]http://data.gov.uk/

changes and to non-conforming, to the schema, data. In general, RDF databases have not yet effectively benefited from the classic schema-aware optimizations used in SQL databases:

- Grouping data that are accessed together using tables.
- Indexing according to filtering and join operations.
- View materialization of frequently queried data patterns.

We argue that all those optimizations can be employed by an RDF database, without any prior knowledge of both the data schema and the workload, by actively monitoring query requests and adapting to the workload.

Result caching is a methodology that has been successfully employed over different applications and computing areas to boost performance and provide scalability. Given the complexity and very high execution latencies [Neumann 10a, Papailiou 13] of several SPARQL query patterns, caching of frequent RDF subgraphs has the potential of boosting performance by even orders of magnitude. While indexing of graph patterns is extensively used in state of the art graph databases [Zhao 07, Yan 04], RDF stores have not yet taken advantage of these techniques. What is more, these schemes focus on *static* indexing of important graph patterns, namely they index subgraphs based solely on the underlying dataset, without any regard for the workload. However, the diversity of the applied SPARQL workloads together with the requirement for high performance for all the different workloads calls for dynamic, workload-driven indexing solutions (e.g., [Idreos 11]).

However, SPARQL queries tend to increase in complexity and become hard to optimize using dynamic programming algorithms. For example, DBpedia reports that its SPARQL endpoint query log contains queries with up to 10 patterns [Gallego 11] and analytical queries in the biomedical domain can include more than 50 patterns [Sahoo 10]. Experimental results show that finding the optimal join plan using dynamic programming approaches can have prohibitive execution times for queries with more than 15 patterns [Neumann 10a, Gubichev 14].

In this work we argue for a *workload-adaptive* RDF caching engine that manages to dynamically index frequent workload subgraphs in real time, and utilize them to decrease response times. The major contributions of this paper are:

- We propose a novel SPARQL query simplification algorithm (Section 3.2), based on the star simplification techniques presented in [Gubichev 14], that splits the query in a skeleton graph and multiple star graphs.
- We introduce a SPARQL query *canonical labelling* algorithm that is able to generate canonical string labels, identical among all isomorphic forms of a SPARQL query. We use these labels to identify and request query graphs to or from the cache. Most importantly, this scheme enables unique identification of all common subgraphs inside any SPARQL workload.

- We integrate our SPARQL query *canonical labelling* algorithm with a state-of-the-art *Dynamic Programming Planner* [Moerkotte 08] by issuing cache requests for all query subgraphs using their canonical label. The resulting optimal execution plan may thus involve, in part or in whole, cached query subgraphs. In addition, we not only examine the utilization of exact cached subgraphs, but also larger, more general graphs that can be used to provide the input of a query subgraph.
- A *Cache Controller* process complements the caching framework. The controller monitors all cache requests issued by the workload queries, enabling it to detect cross-query *profitable* subgraphs and trigger their execution and caching.

Our caching framework is modularly designed so as to support different RDF engines and their respective indexing and cost models. To showcase this, we integrate our prototype implementation with three state-of-the-art RDF engines: RDF-3X [Neumann 10a], TriAD [Gurajada 14] and H_2RDF+ [Papailiou 13]. Extensive evaluation results using diverse workloads show that our caching framework is able to effectively optimize complex SPARQL queries and automatically detect, cache and utilize SPARQL query results. The proposed query simplification approach substantially improves the query optimization and cached result discovery overheads. Using query abstraction and *profitable* result caching, we improve cache utilization, achieving up to three orders of magnitude speedups for several workloads and datasets.

3.2 SPARQL query simplification

There are two common ways of representing a SPARQL query. In the one, we have the SPARQL query graph representation in which nodes denote query variables while edges represent triple patterns that connect them. This structure follows the graph based nature of the RDF data and depicts the subgraph pattern that needs to be extracted from the underlying dataset. For example, Fig. 3.1 depicts the SPARQL query graph of the following query:

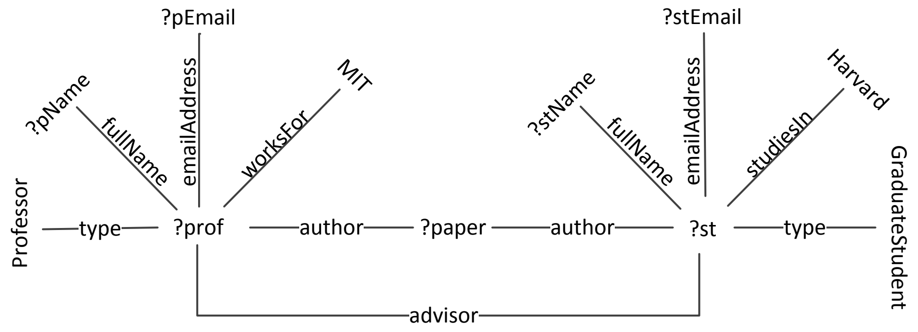


Figure 3.1: *SPARQL query graph for Q_e*

```
Example query $Q_e$:
select * where {
?prof type Professor . (p1)
?prof fullName ?pName. (p2)
?prof emailAddress ?pEmail . (p3)
?prof worksFor "MIT" . (p4)
?prof author ?paper . (p5)
?prof advisor ?st . (p6)
?st author ?paper . (p7)
?st fullName ?stName . (p8)
?st emailAddress ?stEmail . (p9)
?st studiesIn "Harvard" . (p10)
?st type GraduateStudent . (p11)}
```

Another way of representing a SPARQL query is the *join graph*. In this representation, every triple pattern of the query is turned into a node and two nodes are connected if they share a common query variable. Therefore, nodes represent scans of the dataset that retrieve the triples that match with the respective triple pattern. The edges of the graph correspond to the joins that need to be performed in order to answer the query. This graph resembles the join graphs used for query optimisation in traditional relational query optimizers. For our example the join graph of Q_e is presented in Fig. 3.2

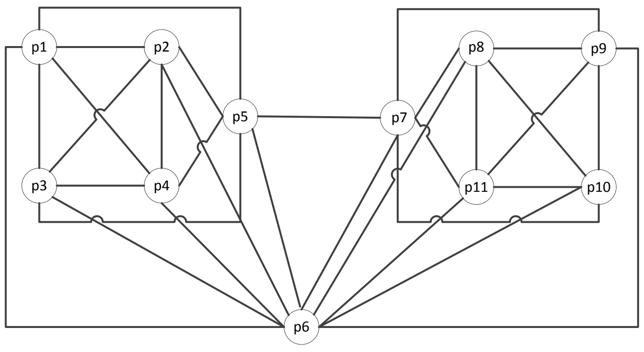


Figure 3.2: *Join graph for* Q_e

Finding the optimal join execution plan for a SPARQL query is a very challenging task due to the complexity of the SPARQL join graphs. As mentioned before, SPARQL queries are very verbose and can contain as much as 50 patterns. Running dynamic programming planners and checking all possible query plans for query graphs with more than 10 patterns becomes prohibitive for state-of-the-art query planners [Gubichev 14, Papailiou 15]. Furthermore, SPARQL queries are described in the primitive RDF graph format. This lack of knowledge of the dataset schema and its grouping properties, results in reconstruction of frequent data patterns for each query using joins. For example, a frequent problem is the attribute retrieval for a certain record (e.g., name, email, phone number for a professor). While this query would require one filtering operation on top of a table using traditional relational databases, in SPARQL it is described as a star graph pattern that needs to be constructed from index scans, of primitive RDF edges, using joins.

Star patterns are very common and widely used in SPARQL query graphs [Gubichev 14]. Star SPARQL graph patterns are transformed to cliques in the join graph representation. This is due to the fact that *all* their triple patterns need to be joined with respect to their common query variable. Query planners use the join graph representation to find the optimal execution plan and thus large cliques, present in the join graph, lead to a worst case exponential complexity for dynamic programming planners [Moerkotte 08]. Canonical labelling of SPARQL queries, used for detecting usable cached results, is also based on the join graph representation. As shown in [Papailiou 15], both canonical labelling and the extended dynamic programming planner that checks the usability of cached results face exponential complexity for the commonly used SPARQL star query patterns.

However, star query pattern optimization have been successfully tackled. Most of the state-of-the-art RDF datastores index RDF data using multiple or all combination of Subject, Predicate, Object ordering permutations [Weiss 08, Neumann 10a, Papailiou 13]. Since the input data can be retrieved in several orderings, the most beneficial query plan for a star query is in most cases a sequence of two-way merge joins [Neumann 10a, Gubichev 14] or a multi-way merge join [Papailiou 13]. In the case of multi-way merge joins [Papailiou 13] there is actually no alternative join execution plan for answering the query. Additionally, when using a sequence of two-way merge-joins [Neumann 10a], heuristics like characteristic sets [Neumann 11] for cardinality estimation and hierarchical characterisation for join ordering [Gubichev 14] can provide near optimal join plans for star patterns in linear-time, without examining all possible join orderings. Therefore, while star query patterns are described as complex clique graphs in the join graph query representation, they can be effectively planned using heuristics that avoid their join enumeration complexity.

In this work, we propose a novel query simplification method that reduces the graph complexity of the SPARQL query join graph while it can be used to both provide near optimal join execution plans and detect all possibly usable cached results. Our simplification approach transforms the initial query graph into a skeleton graph and multiple star subgraphs. The star subgraphs are removed from the query graph and are replaced with one high level star node

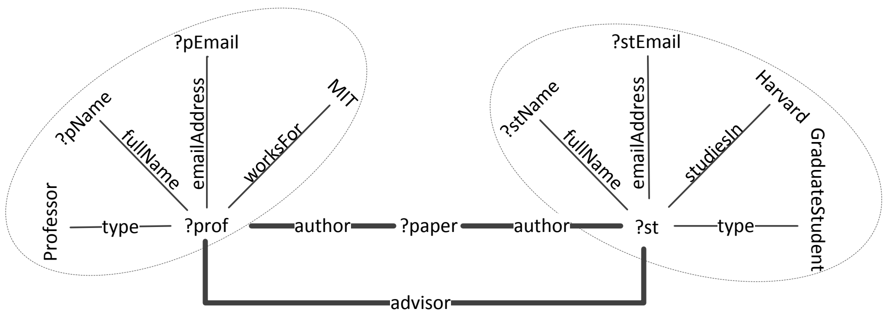


Figure 3.3: *Simplified SPARQL query graph for Q_e*

along with the edges that connect them with other nodes of the query graph. In our example, Fig. 3.3 depicts the skeleton graph of Q_e using bold lines for the edges. The star patterns of the query graph are depicted within dashed ellipses. This simplified query graph maps to a simplified join graph depicted in Fig. 3.4. We note that compared to the initial join graph depicted in Fig. 3.2 the simplified join graph consists of far less nodes and is loosely connected due to the fact that entire clique subgraphs were reduced to simple nodes.

Algorithm 3: SPARQL query simplification

```
Input: SPARQL query graph $G_q = (V_q, E_q)$, join graph $G_j = (V_j, E_j)$
Output: Star subgraphs $St = \langle s, L(s)\rangle$ map of (starID, list of star triple patterns), set of skeleton
        patterns $Sk$, simplified join graph $G_{sj} = (V_{sj}, E_{sj})$
1  $Sk \leftarrow \emptyset$; $St \leftarrow \emptyset$;
2  for $tp \in E_q$ do
3      if $tp.getJoinVariables() > 1$ then
4          $Sk \leftarrow Sk \cup tp$;
5      else
6          $St \leftarrow St.add(tp.getJoinVariableStarId(), tp)$;
7      end
8  end
9  $V_{sj} \leftarrow Sk \cup St.keySet()$;
10 $E_{sj} \leftarrow \emptyset$;
11 for $(p_s, p_d) \in E_j$ do
12     if $p_s \in St \vee p_d \in St$ then
13         $E_{sj} \leftarrow E_{sj} \cup replaceTpIdwithStarId(p_s, p_d)$;
14     else
15         $E_{sj} \leftarrow E_{sj} \cup (p_s, p_d)$;
16     end
17 end
```

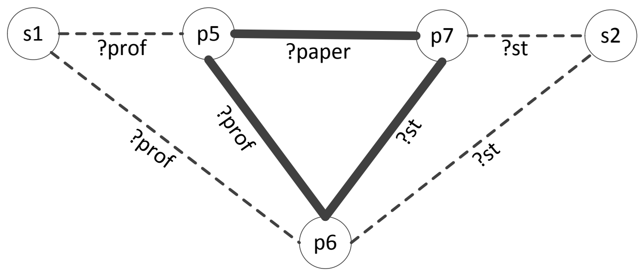


Figure 3.4: *Simplified join graph for* Q_e

A formal description of our query simplification procedure can be found in Algorithm 3. The algorithm takes as input the SPARQL query graph as well as its respective join graph representation and produces the simplified join graph, along with a set containing the skeleton patterns and a map containing the star triple patterns grouped by their starID (a unique ID attributed to each discovered star pattern). To find the skeleton triple patterns, our algorithm iterates over all the patterns of the initial query graph. Each pattern is either a *skeleton* or a *star* pattern. If the pattern contains more than one join variables it is characterised as a skeleton pattern. By join variables we refer to query variables that are present in more than one triple patterns and thus require a join. If a triple pattern contains 2 or more join variables, it is part of a path in the SPARQL query graph and thus it belongs to the skeleton graph of the query. In contrast, if a triple pattern contains only one join variable, it belongs to the star formed around its join variable and is added in the star pattern map using as key the starID that is assigned to the specific join variable. This distinction can create star subgraphs that contain one or more triple patterns.

When the skeleton and star patterns are identified, we build the simplified join graph with nodes: the skeleton patterns and one node per identified star subgraph. The edges of the initial join graph are transferred to the simplified join graph as is, if they connect skeleton patterns. If their source or destination is a star pattern the edge is transformed in order to connect the respective star node in the simplified join graph. The extreme case of stars containing only one triple pattern does not affect the query planning because we just replace the pattern node in the join graph with a star node that contains only one pattern. The complexity of our query simplification algorithm is linear to the size of the query graph because it just iterates over its triple patterns deciding whether they belong to the skeleton of the query or to a star subgraph.

3.3 SPARQL query canonical labelling

In this section, we propose a SPARQL canonical labelling algorithm. We also propose a canonical labelling algorithm that runs on top of our simplified join graph representation, presented in the previous section, and can avoid the exponential labelling complexity introduced by cliques and strongly connected subgraphs of the join graph.

Indexing graph patterns is a challenging task because it requires to tackle the graph isomorphism problem [Köbler 94], a fundamental problem in graph theory. This problem arises when the same query pattern appears in different queries with small deviations such as pattern reordering, variable renaming etc. For example the following SPARQL queries are isomorphic.

```
?prof worksFor ”MIT” .        | ?v1 studiesIn ”Harvard” .
?prof author ?paper .         | ?v2 author ?v3 .
?st author ?paper .           | ?v2 worksFor ”MIT” .
?st studiesIn ”Harvard”       | ?v1 author ?v3
```

All “isomorphs” of the same SPARQL graph must be identified and linked to the same cache entry for a graph caching scheme to work. To address this issue, we extend a solution for graph canonical labelling and introduce the concept of SPARQL graph canonical labelling.

Definition A graph labelling algorithm C takes as input a graph G and produces a unique label L=C(G). C is a canonical graph labelling algorithm if and only if for every graph H which is isomorphic to G we have C(G)=C(H). We call L a canonical label of G. Additionally, L introduces a canonical total ordering between the vertices of G.

The canonical labelling problem shares the same computational complexity with the graph isomorphism problem and belongs to the GI complexity class. GI is one of the few open complexity classes that is not known to be either polynomial-time solvable, or NP-complete [Köbler 94, Hartke 09]. To date, there exists a lot of heuristic evidence that GI is not NP-complete and there are many efficient open-source implementations for both graph isomorphism and canonical labelling algorithms [McKay 14, Junttila 07, Darga 08] that are able to handle really large graphs. One of the first and most powerful canonical labelling algorithms is McKay's *nauty* algorithm [McKay 81] that introduced an innovative use of automorphisms to prune the isomorphism search space. *Bliss* [Junttila 07] extends *nauty* by introducing some extra heuristics that boost its performance on difficult graphs. Such algorithms can compute canonical labels for graphs with thousands of vertices in milliseconds making them ideal to handle even the most complex SPARQL query graphs. However, the most descriptive format that most of the above algorithms work with is the directed vertex-colored graph.

Definition A directed vertex-colored graph is a graph G=(V,E,c), where $V = \{1, 2, ..., n\}$ is a vertex set, $E \subseteq V \times V$ is a set of directed edges, and $c : V \rightarrow N$ is a function that associates to each vertex a non-negative integer(color).

In order to use the existing graph canonical labelling algorithms we present a transformation of SPARQL queries to directed vertex-colored graphs. The transformed SPARQL graph can be then labelled by one of the previously mentioned graph canonical labelling algorithms. The proposed transformation guarantees no lose of query information. It avoids introducing false positives or false negatives, i.e., non-isomorphic SPARQL queries having the same label or isomorphic queries having different labels.

3.3.1 Labelling SPARQL join graphs

In this section, we present our SPARQL join graph canonical labelling algorithm. As a running example, let us assume that we need to get a canonical label for the following SPARQL query:

```
?prof teacherOf ?gcourse . (1)
?prof teacherOf ?ugcourse . (2)
?ugcourse type UndergraduateCourse . (3)
?gcourse type GraduateCourse . (4)
```

The first task is to transform SPARQL queries to directed vertex-and-edge-colored graphs.

Definition A directed vertex-and-edge-colored graph is a graph G=(V,E,c_v,c_e), where $V = \{1, 2, ..., n\}$ is a finite set of vertices, $E \subseteq V \times V$ is a set of directed edges, $c_v : V \rightarrow N$ and $c_e : E \rightarrow N$ are color functions for vertices and edges.

The join graph of the query is depicted on the left part of Fig. 3.5, where the vertex IDs correspond to the triple pattern IDs presented above. Initially, we remove all information related to the variable names. To do so, for each triple query we create a label that consists of three IDs, one for each position inside the triple. Bound nodes are translated to integer IDs using a String to ID dictionary while variables are translated to zero. For the current example, let us assume that the String-ID dictionary contains: {teacherOf→15, type→3, UndergraduateCourse→52, GraduateCourse→35}. The label of the first triple query would be, for example, "0_15_0". *OPTIONAL* triple patterns are handled by appending a special character '|' at the beginning of their label. We also direct and color the edges according to the type of join that they represent. All possible types are {SS, SP, SO, PS, PP, PO, OS, OP, OO}. To reduce the number of edge colors needed, a position ordering (S<P<O) is introduced and only edges whose source position is lower than or equal to their destination position are added to the transformed graph. Therefore, only the following 6 edge types {SS, SP, SO, PP, PO, OO} are required. In the second graph of Fig. 3.5, we can see both the vertex and edge labels for our query.

In the final step, the vertex and edge labels are translated to non-negative integers(colors). To do so, a sort of the vertex labels is performed and the position of the label in the sorted set is used as the final label. For edge labels the following translation {SS→1, SP→2, SO→3, PP→4, PO→5, OO→6} is used. The final graph is a directed vertex-and-edge-colored graph and can be seen in the first graph of Fig. 3.6. SPARQL grouping, ordering, filtering and projection

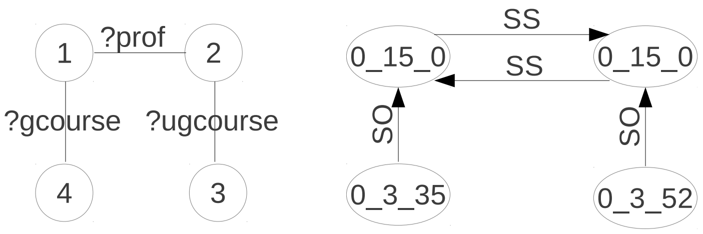


Figure 3.5: *SPARQL query transformation*

information is not used for labelling SPARQL queries. Queries that are isomorphic but differ for example in result ordering will get the same label but are later handled as different versions of the same result by the dynamic programming planner, Section 3.4. The above transformation removes all information relevant to variable naming while managing to maintain all structural information of the query using the directed join edges [Papailiou 15].

In [McKay 90], a practical way to transform directed edge-colored graphs to simple directed graphs without changing their automorphism group is presented. More specifically, if there are v vertex colors and all available edge colors are integers in $\{1, 2, \ldots, 2^d - 1\}$, a graph with d layers is constructed. Each of the layers contains n vertices, where n is the number of vertices of the initial graph. As mentioned in the previous paragraph only 6 edge colors are required, one for each possible type of triple pattern join, and thus d equals to 3 leading to a new graph that contains $3n$ vertices. The vertices of the different layers (each corresponding to one vertex of the original graph) are vertically connected using paths. The vertex colors of the first layer remain the same as in the original graph and they get propagated to higher layers by adding v to the corresponding color of the lower layer. For each edge of the original graph the binary expansion of its color number tells us which layers should contain the horizontal edge. For example, an edge with color 3, whose binary expansion is 011, will be placed both in the first and the second layer of the new graph. The transformation for the above example is depicted in the second graph of Figure 3.6, where the vertex IDs represent the assigned colors.

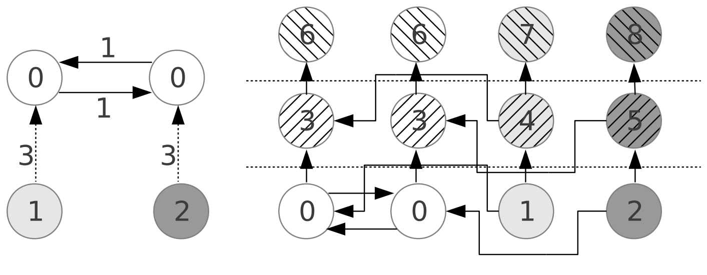


Figure 3.6: *Transformation to directed vertex-colored graph*

At this point the *Bliss* algorithm is used to produce a canonical label for the above graph. The canonization process returns a canonical order of the graph's vertices. As stated in [McKay 90], the order by which a canonical labelling of the new graph labels the vertices of the first layer can be taken to be a canonical labelling order of the original graph. Thus, after executing *Bliss* we can get a canonical ordering for our initial query. For our example the canonical ordering is {1,2,4,3}, where the ids are the SPARQL triple query ids used in the initial graph. Using this ordering, a string canonical label of the SPARQL query graph can be generated. To do so, we use this canonical ordering to iterate over the triple patterns, for each pattern we append its signature at the end of the label string. While iterating, a canonical ordering of the variables is also generated, i.e. variable ?prof is the first variable that we find and thus it gets the canonical ID ?1. In our example, the canonical variable mapping is {?prof→?1, ?gcourse→?2, ?ugcourse→?3} and the generated string label is ?1_15_?2&?1_15_?3&?2_3_35&?3_3_52. The presented algorithm produces the exact same string label for any isomorphic SPARQL query and thus is a canonical labelling algorithm for SPARQL query graphs. This label can be used as key for a SPARQL result caching implementation.

3.3.2 Labelling simplified SPARQL join graphs

As a running example, let us assume that we need to get a canonical label for the simplified join graph presented in Fig. 3.4. As before, let us also assume that the String-ID dictionary contains: {author→407, advisor→85, type→3, fullName→176, emailAddress→243, worksFor→401, studiesIn→545, Professor→211, GraduateStudent→102, MIT→1623, Harvard→1874}. Initially, we transform the simplified join graph of Fig. 3.4 to a labelled graph presented in Fig. 3.7. The labelling procedure of the skeleton part of the graph is the same as the one presented in the previous section. For each of the star nodes, we create a list of labels containing all their triple patterns. The only difference, while generating the triple pattern labels, is that, in this case, we do not replace the star join variable with '0' but with "?s" in order to maintain the

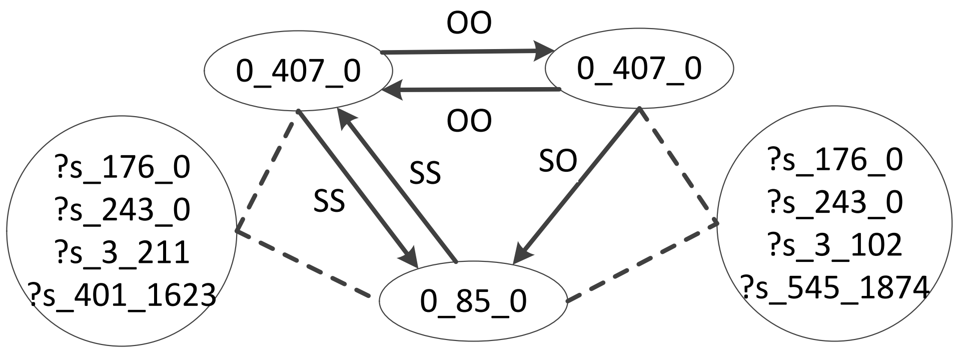


Figure 3.7: *Labelled simplified join graph for Q_e*

information of the star variable position. After creating the list of pattern labels we also lexicographically sort them in order to generate a unique string label for the star node. The labelling of simplified join graphs is split into the following steps:

1. Generate a canonical label for the skeleton subgraph of the labelled join graph. This is done using the canonical labelling algorithm presented in the previous section.

2. Use the canonical variable mapping generated from the labelling of the skeleton graph to generate canonical labels for the star subgraphs.

3. Concatenate the skeleton and the star canonical labels in order to produce a canonical label for the entire query described by the simplified join graph.

In our running example, the canonical label of the skeleton graph is ?1_407_?2&?3_407_?2& ?3_85_?1 and the generated canonical variable mapping is {?st→?1, ?paper→?2, ?prof→?3}. The canonical variable mapping of the skeleton graph generates canonical IDs for all the star join variables in a connected join graph. In the extreme case of a star only query graph, where the skeleton subgraph is empty the star join variable gets the first canonical ID: ?1. Having generated canonical variable IDs for all the star join variables we use them to iterate over the star subgraphs. Each star subgraph, is labelled using the lexicographically sorted list of its pattern labels, as depicted in Fig. 3.7. In our example, we first label the star subgraph around ?st→?1 and the generated canonical label is (?1,[?s_176_0,?s_243_0,?s_3_102,?s_545_1874]). The label of the second star subgraph around ?prof→?3 is (?3,[?s_176_0,?s_243_0,?s_3_211,?s_401_1623]). When creating the star labels, we also generate canonical variable IDs for the rest of the star variables {?stName→?4, ?stEmail→?5, ?pName→?6, ?pEmail→?7}. However, those IDs are not replaced in the final canonical label. As we will discuss in the next section, this provides the ability for our labels to be used, apart from exact isomorphism testing, for subgraph isomorphism testing. To generate the canonical label of the entire query we just concatenate the skeleton label with the star labels using the canonical star join variable order. Therefore the label of the entire query is: ?1_407_?2&?3_407_?2&?3_85_?1&{(?1,[?s_176_0,?s_243_0,?s_3_102,?s_545_1874]), (?3, [?s_176_0,?s_243_0,?s_3_211,?s_401_1623])}.

Proof The presented algorithm is a canonical labelling algorithm for SPARQL query graphs. The canonical labelling process of the skeleton query subgraph is the same as the one mentioned in section 3.3.1. Therefore, the label generated for the skeleton query subgraph is a canonical label. Concerning star subgraphs, they can be described as a set of pattern labels E, representing triple patterns that are only connected by their common join variable and are also disconnected from the rest of the query graph. Our algorithm bases the labelling of star graphs on: i) removing variable naming information from the labels of the set E, ii) canonical labelling the set of edges using its lexicographic order, iii) retrieving the canonical ID of the star join variable from the skeleton graph label and propagating canonical IDs to the star graph variables.

We will prove that the lexicographic ordering of the pattern label set E provides a canonical label for a star query graph. Initially, all isomorphic star queries generate the same set E, independent of the utilized variable names as well as the triple pattern ordering used. Respectively, a pattern set E can be used to generate an isomorphic star query graph. This can be done by giving unique IDs to all non-join variables presented as "0" in the pattern labels. The join variable presented as "?s" can also be assigned to a unique variable ID. Therefore, canonical labelling of the pattern label set E provides a canonical label for the star graph. Due to the fact that there are no dependences between the labels contained in E, its lexicographic order can be used as a canonical order. Thus, the lexicographic order of the pattern set E provides a canonical label for a star query graph.

Additionally, concatenating the skeleton label with the star labels using the canonical ordering of the star join variables provides a canonical label for the entire query. To prove this, we note that isomorphic queries have the same skeleton subgraph and thus they will generate the exact same canonical ordering for all their star join variables. Therefore for all isomorphic queries, the star labels will be concatenated in the same order and thus the generated labels are canonical for the entire query and we can use exact label matching to performing isomorphism tests.

3.3.3 Using canonical labels for subgraph isomorphism

Using the previously mentioned algorithm to generate canonical labels we also have the capability to perform subgraph isomorphism testing for special graph classes. More formally, the problem of subgraph isomorphism can be defined as:

Definition Given two graphs $G(V, E)$ and $H(V', E')$ we want to test if there exists a subgraph, $G_1(V_1, E_1) : V_1 \subseteq V, E_1 \subseteq E$, of G that is isomorphic to H.

While canonical labels are designed to perform exact graph isomorphism testing, our proposed skeleton-star canonical labels can be efficiently used to also perform subgraph isomorphism for queries that share the same skeleton graph but have deviations in their star subgraphs. More formally, we introduce the problem of skeleton-star subgraph isomorphism testing.

Definition Given two graphs $G(V, E)$ and $H(V', E')$ and their respective simplified graphs (Sk, St), (Sk', St'), where $Sk \subseteq (V, E)$, $Sk' \subseteq (V', E')$ and $St = \langle s, L(s) \subseteq V \rangle$, $St' = \langle s, L'(s) \subseteq V' \rangle$ are the skeleton and star subgraphs of G and H respectively. H is skeleton-star subgraph isomorphic to G iff there exists a subgraph, $G_1(V_1, E_1) : V_1 \subseteq V, E_1 \subseteq E$, of G with $Sk_1 = Sk$ that is isomorphic to H. Additionally, if H is a skeleton-star subgraph of G, it is true that their skeleton subgrahs are isomorphic $Sk = Sk_1 = Sk'$. Concerning their star subgraphs, it is true that $St' \subseteq St : \{\forall(s, L'(s)) \in St' : (s, L(s)) \in St, L'(s) \subseteq L(s)\}$

To be skeleton-star subgraph isomorphic, two queries must share isomorphic skeleton structures and therefore the same skeleton canonical labels. Furthermore, all the star subgraphs of the one query must be subgraphs of the star subgraphs of the other query. For example, the query depicted in Fig. 3.8 is skeleton-star subgraph isomorphic to the query depicted in Fig. 3.3.

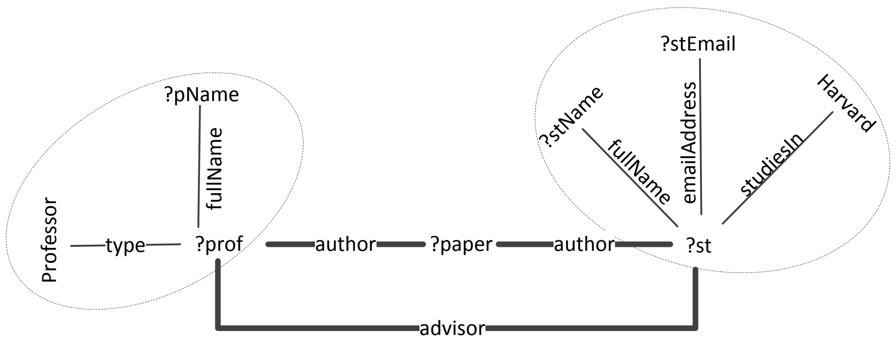


Figure 3.8: *Query that is skeleton-star subgraph isomorphic to Q_e*

The algorithm that we can use to perform skeleton-star subgraph isomorphism tests, using our canonical labels, is presented in Algorithm 4. The algorithm starts by checking the equality of the skeleton canonical labels. If the skeleton subgraphs of the tested queries are isomorphic, we continue to the star subgraph test, otherwise we return *false* because the two queries are by definition not skeleton-star subgraph isomorphic. To test the star subgraph isomorphism we need to ensure that *all* the star subgraphs of H are present in G. To do this efficiently, we take into account the canonical ordering of the star join variables as well as the lexicographic order of the patterns inside the star labels.

Due to the fact that the skeleton graphs of the two queries are isomorphic, our canonical labelling algorithm will generate, for both, the same canonical variable IDs and therefore their star subgraphs will be labeled in the same canonical order. The only difference is that G can have more star subgraphs than H. To find if *all* star subgraphs of H exist in G we just iterate over the ordered star subgraphs of H and try to match them with the star subgraphs of G. To check if two star subgraphs match, we first check if they have the same join variable canonical ID. If they do, we continue the check by iterating over the lexicographically ordered lists of their patterns. Again, the list of patterns for a star subgraph of G is allowed to have more patterns than the

Algorithm 4: Skeleton-star subgraph isomorphism testing

```
Input: Two graphs $G, H$ and $(G_{sk}, G_{st}), (H_{sk}, H_{st})$, their canonical skeleton and star labels
Output: True/false if $H$ is skeleton-star subgraph isomorphic to $G$
if $G_{sk} \neq H_{sk}$ then
    return false;
else
    $sit \leftarrow H_{st}.iterator()$; $hst = sit.nextStar()$; $starMatches = false$;
    for $gst \in G_{st}$ do
        if $gst.getStarVar() = hst.getStarVar()$ then
            $pit = hst.iterator()$; $hp = pit.nextPattern()$; $patternMatches = false$;
            for $gp \in gst$ do
                if $gp = hp$ then
                    $patternMatches = true$;
                    if $pit.hasMore()$ then
                        $hp = pit.nextPattern()$;
                    else
                        break;
                    end
                end
            end
            if $pit.hasMore() || patternMatches = false$ then
                return false;
            end
            $starMatches = true$;
        end
    end
    if $sit.hasMore() || starMatches = false$ then
        return false;
    end
    return true;
end
```

respective list of H, but *all* patterns of H's list must be present in G's list. The use of the generic variable IDs "0" and "?s" for the star pattern labels, described in the previous section, ensures that the lexicographic order of the star patterns will be the same in both queries. Therefore, the proposed algorithm can perform the skeleton-star subgraph isomorphism test with just one pass over the canonical labels of the tested queries.

3.3.4 Canonical labelling complexity

In this section, we formally examine the time and space complexities of the proposed canonical labelling algorithms. The first examined algorithm is the canonical labelling algorithm of Section 3.3.1. The complexity of this algorithm is directly associated with the complexity of the *Bliss* algorithm that attempts to solve the graph isomorphism (GI) problem. GI is known to have time complexity at most $O(2^{\sqrt{n \log n}})$ for graphs with n vertices [Arvind 00]. However, this is not a representative bound for *Bliss* because its complexity mainly depends on the amount of automorphisms present in the graph structure rather than on the number of its vertices. Indeed, there are graph examples that result in exponential labelling times but for general graphs *Bliss* presents sub-exponential complexity. The extended SPARQL query labelling algorithm introduces a polynomial time, $O(n)$, transformation of the input query which is negligible compared to the worst-case exponential time complexity of *Bliss*. Its major overhead is the fact that it transforms the n vertices of the query graph to $3n$ vertices and thus introduces a polynomial increase to the input size.

Concerning the simplified join graph canonical labelling algorithm, of Section 3.3.2, it again depends on the complexity of the *Bliss* algorithm. The major difference now is that *Bliss* only runs on top of the skeleton subgraph of the query. The complexity of the skeleton subgraph labelling is the same as discussed above but it now depends on sk, the number of skeleton triple patterns of a query, which can be much smaller than n. Furthermore, our simplification procedure removes all star subgraphs from the query graphs and thus removes cliques from the join graph used as input for canonical labelling. Therefore, the skeleton graph not only contains less vertices than the original join graph but is also more loosely connected. As a result, the skeleton graph contains less automorphisms than the initial join graph and reduces the complexity of the automorphism enumeration procedure performed by the *Bliss* algorithm. Labelling star subgraphs has polynomial $O(st \log(st))$, to the number of star patterns st, time complexity because it just requires a lexicographic sort of the pattern labels.

Lastly, having generated canonical labels for query graphs, we can efficiently perform isomorphism and skeleton-star subgraph isomorphism tests in polynomial time. The complexity of the proposed skeleton-star subgraph isomorphism algorithm is linear to the number of the query patterns (label size) n, because in the worst case it needs to iterate once over the query labels. The exact isomorphism test has also linear complexity to the label size n because it needs to perform an equality match between two labels and in the worst case it needs to examine the entire label strings.

3.4 Query planning

Finding the optimal join plan for complex queries has always been a major research challenge in optimizing database systems. In addition, our system needs to effectively discover which of the maintained cached results can be used to provide input for a query's subgraph. Both these tasks have exponential complexity to the size of the query because they need to check all of its subgraphs. While there exist several greedy, heuristic approaches for SPARQL query planning [Tsialiamanis 12, Papailiou 13], they cannot be easily integrated with a cached result discovery algorithm that finds all relevant cached results. In contrast, dynamic programming query planning approaches [Moerkotte 06, Neumann 10a] explore all subgraphs of a query and thus can be easily modified to achieve both optimal query planning and cached result discovery.

One of the oldest and most efficient dynamic programming algorithms for join planning is *DPsize* [Gassner 93], widely used in commercial databases like IBM's DB2. *DPsize* limits the search space to left-deep trees and generates plans in increasing order of size. A more recent approach, *DPccp* [Moerkotte 06] and its variant *DPhyp* [Moerkotte 08] are considered to be the most efficient, state of the art dynamic programming algorithms for query optimization. They reduce the search space by examining connected subgraphs of the query in a bottom-up fashion. In addition, *DPccp* is successfully utilized to generate optimal SPARQL join plans in RDF-3X [Neumann 10a].

SPARQL queries frequently contain many star-shaped pattern joins, making multi-way joins attractive for execution engines. A multi-way join can process a star query that contains an arbitrary number of patterns in a single step. Especially in the case of indexing all ordered permutations of RDF indexes (e.g., [Weiss 08]), a star-shaped query graph can be effectively executed with one multi-way merge join on the common join variable instead of a sequence of two-way merge joins [Papailiou 14]. Since *DPccp* only explores two-way join plans, in this work we extend it by adding support for both cached result discovery and multi-way join exploration. This way, our caching framework can also be integrated with execution engines that support multi-way joins.

3.4.1 Multi-way join exploration

In this section, we describe the changes required for the *DPccp* to efficiently explore multi-way join plans. The input query graph $G = (V, E, c)$ used in our dynamic programming planner is the simplified SPARQL join graph presented in Fig. 3.3 of Section 3.2, where each triple query is represented by a vertex and triple queries that share a common variable are linked with an edge labelled by the variable's name. *DPccp* bases its enumeration procedure on finding all *csg-cmp-pairs* in the query graph [Moerkotte 08]. *Csg-cmp-pairs* are pairs containing a connected subgraph(csg) of the query graph and one of its connected complement subgraphs(cmp).

Definition (csg-cmp-pair). Let $G = (V, E, c)$ be a query graph and S1, S2 two subsets of V such that S1 $\subseteq$ V and S2 $\subseteq$ (V $\setminus$ S1) are a connected subgraph and a connected complement respectively. If there further exists an edge (u, v)$\in$E such that u$\in$S1 and v$\in$S2, we call (S1, S2) a *csg-cmp-pairs*.

The enumeration of *csg-cmp-pairs* restricts *DPccp* to 2-way join plans because each *csg-cmp-pair* corresponds to a 2-way join between the csg and the cmp graphs. To explore multi-way join plans our algorithm enumerates label connected, connected complement subgraph lists(*cmp-lc-list*).

Definition (cmp-lc-list). Let $G = (V, E, c)$ be a query graph, L an edge label and S={S_1,. . . ,S_n} a list of sets where $S_1 \subseteq V$, $S_2 \subseteq (V \setminus S_1)$, ..., $S_n \subseteq (V \setminus S_1, \ldots, S_{n-1})$ are connected subgraphs of G. If for every pair (S_i, S_j) ∈ S there exists an edge (u, v) ∈ E with label L such that u ∈ S_i and v ∈ S_j, we call S a cmp-lc-list.

Each *cmp-lc-list* corresponds to a multi-way join of all subgraphs contained in the list, on the common join variable represented by label L. Note that if S={$S_1, \ldots, S_n$} is a *cmp-lc-list*, then any reordering of the sets in S will result in a *cmp-lc-list* as well. We restrict the enumeration of *cmp-lc-lists* in the same way as presented in [Moerkotte 08]. We enumerate only the lists that satisfy the condition min(S_1) $\prec$ min(S_2) $\prec \cdots \prec$ min(S_n), where $\prec$ is the total ordering relation introduced over the vertices of G by their vertex IDs and

$$min(S_i) = \{v | v \in S_i, \forall v' \in S_i : v' \neq v \implies v \prec v'\} \tag{3.1}$$

Assuming that multi-way join operators are commutative, meaning that the order of the sets inside the list does not affect the cost of the join, the enumeration of all list orderings is superfluous. Thus, the application of this restriction ensures that no duplicate *cmp-lc-lists* are examined by our dynamic programming algorithm. Furthermore, in order to find the optimal join execution plan, a dynamic programming algorithm would need to examine all the *cmp-lc-lists*, each corresponding to a distinct multi-way join, making our algorithm optimal in terms of the amount of examined plans.

3.4.2 Dynamic programming planner algorithm

In this section, we present the pseudocode of our dynamic programming planner. We focus more on the changes made compared to the *DPccp* algorithm but we maintaining the same notation as used in [Moerkotte 08], allowing interested readers to easily point to this paper for the details of the baseline algorithm. One of the major differences of our algorithm compared to *DPccp* is the structure of the dpTable used. To leverage the strengths of maintaining all permutations of RDF indexes, a system needs to promote the use of merge joins [Weiss 08, Neumann 10a]. To achieve that in the presence of multiple indexes and cached orderings of the same graph pattern, we need to carefully preserve result orderings while generating join plans. A result with a certain ordering, while more expensive to generate than another with no ordering, can be more efficiently used in a subsequent join leading to a better join plan for the query. To cover this case, we change the structure of our dynamic programming table. While in *DPccp* only one plan is kept for each query subgraph, our dpTable maintains, for each query subgraph, a list of plans that contains the best join plan for each discovered ordering. The *mergeAll* method merges two join plan lists maintaining only the best plan for each distinct ordering found in both lists.

The proposed dynamic programming planner runs on top of the simplified join graph described in Section 3.2. Its main method is the *solve*(V), where V is the set of nodes of the simplified join graph, presented in Algorithm 5.

Algorithm 5: $solve(V)$

```
1  for $v \in V$ do
2      //initialize dpTable
3      if $v.isStar()$ then
4          $dpTable[\{v\}].mergeAll(getPlanForStar(v))$;
5      else
6          $dpTable[\{v\}].mergeAll(indexScans(v))$;
7      end
8  end
9  for $v \in V$ descending according to $\prec$ do
10     $emitCsg(\{v\})$;
11     $enumerateCsgRec(\{v\}, \mathcal{B}_v)$;
12 end
13 return $dpTable[V]$;
```

Initially, the *solve*(V) method seeds the dpTable with plans for all the vertices of the simplified join graph. The simplified graph contains vertices that represent either skeleton triple patterns or star subgraphs. In the case of skeleton triple patterns, we add to the dpTable all the possible triple query index scans. For example, if a hexastore [Weiss 08] indexing is used, the data of a triple pattern can be retrieved from multiple indexes with different orderings. Concerning star subgraphs, we need to generate join plans and store them in the dpTable. For engines that support multi-way joins, the best plan for a star subgraph consists of one multi-way merge join on the star variable. In the case of engines that use 2-way join plans, efficient methods of generating join plans for star patterns, such as hierarchical characterization and characteristic sets [Gubichev 14], are used.

Consequently, the *solve* method calls the two subroutines *emitCsg* and *enumerateCsgRec* for all vertices in decreasing order according to $\prec$. For readability, we maintain the same naming used in [Moerkotte 08] but for us the *csg* set represents the first set of a *cmp-lc-list*. Thus, the *emitCsg* method bounds the first set of the *cmp-lc-list* and continues by enumerating the rest of its subsets. The *enumerateCsgRec(*S_1, X*)* function is used to extend a given connected subgraph S_1 to a larger connected subgraph while avoiding vertices that belong to the exclusion set X. The exclusion set is used to avoid duplicate subgraph enumerations by prohibiting the subgraph to expand and include nodes that are ordered before v according to $\prec$. This is achieved with the use of $\mathcal{B}_v = \{w : w \prec v\} \cup \{v\}$ which restricts nodes that have IDs smaller than v.

Algorithm 6: $enumerateCsgRec(S_1, X)$

```
1 for $N \subseteq \mathcal{N}(S_1, X) : N \neq \emptyset$ do
2     $emitCsg(S_1 \cup N)$;
3 end
4 for $N \subseteq \mathcal{N}(S_1, X) : N \neq \emptyset$ do
5     $enumerateCsgRec(S_1 \cup N, X \cup \mathcal{N}(S_1, X)$;
6 end
```

To extend a subgraph, the *enumerateCsgRec* uses the concept of a subgraph's neighbourhood $\mathcal{N}(S_1, X)$ containing all nodes reachable from S_1 with one edge that are not in X. For all those sets, we call both *emitCsg* and *enumerateCsgRec*.

To facilitate the better understanding of our algorithm, Fig. 3.9 depicts the trace of its execution for the simplified join graph of Section 3.2. The trace presented on the upper part of the figure depicts the sequence of calls to both *emitCsg* and *emitCmpLcList* functions. The trace of *emitCsg(S_1)* is depicted as $\{S_1\}$ while for *emitCmpLcList* we use $[var|\{S_1\}, \ldots, \{S_n\}]$. The *solve* method will start by emitting the set {12} because this is the largest vertex according to $\prec$. This set will not generate any *cmp-lc-lists* and will not be further extended as its exclusion set contains all its neighbours. The same will happen for the set {11}. The *emitCsg* method will then enumerate all *cmp-lc-lists* that have {6} as their primary set. When this is done the *enumerateCsgRec* will expand {6} and call *emitCsg* for {6,12}. The same will happen for {5} that will be extended to {5, 6}, {5, 11}, {5, 12}, {5, 6, 11}, {5, 6, 12}, {5, 11, 12}, {5, 6, 11, 12}. The set {4} will be extended as, {4, 5}, {4, 6}, {4, 11}, {4, 5, 6}, {4, 5, 11}, {4, 6, 11}, {4, 5, 6, 11} from the first loop of the *enumerateCsgRec* method. The second loop will then recursively add the node 12 in all the above sets calling *emitCsg* for {4, 5, 12}, {4, 6, 12}, {4, 5, 6, 12}, {4, 5, 11, 12}, {4, 6, 11, 12} and {4, 5, 6, 11, 12}.

During the enumeration process, the *emitCsg(S_1)* function is executed only once for every connected subgraph of the query in hand, making it a great place to integrate our check cache mechanism. In addition, *emitCsg(S_1)* is called before every attempt to utilize S_1 in a join and thus the cache results retrieved here will be available for the computation of the cost of any join plan that contains S_1.

The *emitCsg* function proceeds with enumerating the *cmp-lc-lists* presented in Section 3.4.1. We now introduce the concept of the labelled neighbourhood of a subgraph $N_{label} =$

Algorithm 7: $emitCsg(S_1)$

```
if $|S_1| \geq 2$ then
    $dpTable[S_1].mergeAll(checkCache(S_1))$; //check cache and merge plans
end
$X = S_1 \cup \mathcal{B}_{min(S_1)}$;
$N_{label} = \mathcal{N}_{label}(S_1, X)$;
for $(l, S_2) \in N_{label} : S_2 \neq \emptyset$ do
    $X_1 = X \cup S_2$;
    $list.push(S_1)$;
    $enumerateCmpLcList(list, S_2, X_1, l)$;
end
```

$\mathcal{N}_{label}(S_1, X)$. N_{label} is a set of pairs (l, S_2), where l is an edge label that corresponds to a join variable and S_2 is the set of nodes not contained in X and reachable from S_1 with one edge labelled by l. We also want to enforce the enumeration of multi-way joins that join *all* patterns that share a certain variable in one step. To do so, $N_{label}(S_1, X)$ is restricted to contain only pairs (l, S_2), such that $S_1 \cup S_2$ contains *all* the nodes of the join graph that have an edge labeled by l. For each pair of the labelled neighbourhood of S_1 we enumerate all possible *cmp-lc-lists* that contain S_1 as their primary set and enforce the *cmp-lc-list* ordering constraints of Section 3.4.1 using the function *enumerateCmpLcList*.

Algorithm 8: $enumerateCmpLcList(list, S, X, l)$

```
if $S = \emptyset$ then
    $extendCmpLcList(list, 1, X, l)$;
    return;
end
$m = \min_{\prec}\{v \in S\}$;
for $S_1 \subseteq S \setminus m$ do
    $S_1 = S_1 \cup m$; $S_2 = S \setminus S_1$; $list.push(S_1)$;
    $enumerateCmpLcList(list, S_2, X, l)$;
    $list.pop()$;
end
```

The *enumerateCmpLcList(list*, S, X, l*)* function recursively splits the set S in two sets; S_1 containing the minimum node of S according to $\prec$ and $S_2 = S \setminus S_1$. S_1 gets pushed in the *cmp-lc-list* and the enumeration continues until S_2 has no more elements. This procedure ensures that the *cmp-lc-lists* satisfy the ordering property described in Section 3.4.1. Finally, the *enumerateCmpLcList* calls the *extendCmpLcList* function to recursively extend all the subsets of a *cmp-lc-list*.

The *extendCmpLcList(list*, i, X, l*)* function enumerates all possible extensions of the i-th subgraph of the *cmp-lc-list*. Its primary subgraph was extended using the *enumerateCsgRec*

Algorithm 9: $extendCmpLcList(list, i, X, l)$

```
1  if $i < list.size() - 1$ then
2      $extendCmpLcList(list, i+1, X, l)$;
3  else
4      $emitCmpLcList(list, l)$;
5  end
6  $S = list.get(i)$;
7  for $N \subseteq \mathcal{N}(S, X) : N \neq \emptyset$ do
8      $S_1 = S \cup N$; $list.put(i, S_1)$;
9      $extendCmpLcList(list, i, X \cup \mathcal{N}(S, X), l)$;
10 end
11 $list.put(i, S)$;
```

function and thus the extension procedure skips it and recursively extends all subgraphs having index $i = 1, \ldots, list.size() - 1$. The *emitCmpLcList(*$list, l$*)* is the final step of our dynamic programming algorithm and is responsible for computing the cost of the multi-way join described by $(list, l)$, using the query engine's cost model and updating the dpTable entry for the resulting subgraph.

In our example, the *emitCsg*({5}) method will call *enumerateCmpLcList* once for every labelled neighbourhood of {5} ([*st*,{6,12}]). In this case, the corresponding lists cannot be further extended due to the ordering constraints and the *emitCmpLcList* will be called twice: [st|{5}, {6}, {12}], [st|{5}, {6, 12}]. In the case of {4}, the labelled neighbourhood is ([prof,{5,11}],[paper,{6}]). Examining the case of the neighbourhood [prof,{5,11}], the recursive enumeration of the *enumerateCmpLcList* will then call *extendCmpLcList* for the following lists: [{4}, {5}, {11}], [{4}, {5,11}]. Finally, the *extendCmpLcList* will extend all the subsets of those lists calling the *emitCmpLcList* for: i) [prof|{4}, {5}, {11}], [prof|{4}, {5, 6}, {11}], [prof|{4}, {5, 12}, {11}], [prof|{4}, {5, 6, 12}, {11}] ii) [prof|{4}, {5, 11}], [prof|{4}, {5, 6, 11}], [prof|{4}, {5, 11, 12}], [prof|{4}, {5, 6, 11, 12}].

3.5 Caching framework

In this section, we describe our caching framework, used to discover cached results that can enhance the execution of the query in hand. As depicted in Figure 3.10, query resolution starts from the *Dynamic Programming Planner*, described in Section 3.4, that iterates over all possible plans identifying the optimal execution plan. Meta-data about cached results are stored in-memory, indexed using their canonical labels, in the *Result Cache* table. In order to find usable cached results, while the planner examines all the connected subgraphs of the simplified join graph, it issues cache checks using the canonical labelling algorithm presented in Section 3.3.2. The benefit of all discovered cached results is examined by the cost model during the planner's execution and the optimal plan can contain (in part or in whole) cached query results. The

{12}, {11}, {6}, {6, 12}, {5}, [st|{5}, {6}, {12}], [st|{5}, {6, 12}], {5, 6}, [st|{5, 6}, {12}], {5, 11}, [st|{5, 11}, {6}, {12}], [st|{5, 11}, {6, 12}], {5, 12}, [st|{5, 12}, {6}], {5, 6, 11}, [st|{5, 6, 11}, {12}], {5, 6, 12}, {5, 11, 12}, [st|{5, 11, 12}, {6}], {5, 6, 11, 12}, {4}, [prof|{4}, {5}, {11}], [prof|{4}, {5, 6}, {11}], [prof|{4}, {5, 12}, {11}], [prof|{4}, {5, 6, 12}, {11}], [prof|{4}, {5, 11}], [prof|{4}, {5, 6, 11}], [prof|{4}, {5, 11, 12}], [prof|{4}, {5, 6, 11, 12}], [paper|{4}, {6}], [paper|{4}, {5, 6}], [paper|{4}, {5, 6, 11}], [paper|{4}, {6, 12}], [paper|{4}, {5, 6, 12}], [paper|{4}, {5, 6, 11, 12}], {4, 5}, [prof|{4, 5}, {11}], [paper|{4, 5}, {6}], [paper|{4, 5}, {6, 12}], [st|{4, 5}, {6}, {12}], [st|{4, 5}, {6, 12}], {4, 6}, [prof|{4, 6}, {5}, {11}], [prof|{4, 6}, {5, 12}, {11}], [prof|{4, 6}, {5, 11}], [prof|{4, 6}, {5, 11, 12}], [st|{4, 6}, {5}, {12}], [st|{4, 6}, {5, 11}, {12}], [st|{4, 6}, {5, 12}], [st|{4, 6}, {5, 11, 12}], {4, 11}, [prof|{4, 11}, {5}], [prof|{4, 11}, {5, 6}], [prof|{4, 11}, {5, 12}], [prof|{4, 11}, {5, 6, 12}], [paper|{4, 11}, {6}], [paper|{4, 11}, {5, 6}], [paper|{4, 11}, {6, 12}], [paper|{4, 11}, {5, 6, 12}], {4, 5, 6}, [prof|{4, 5, 6}, {11}], [st|{4, 5, 6}, {12}], {4, 5, 11}, [paper|{4, 5, 11}, {6}], [paper|{4, 5, 11}, {6, 12}], [st|{4, 5, 11}, {6}, {12}], [st|{4, 5, 11}, {6, 12}], {4, 6, 11}, [prof|{4, 6, 11}, {5}], [prof|{4, 6, 11}, {5, 12}], [st|{4, 6, 11}, {5}, {12}], [st|{4, 6, 11}, {5, 12}], {4, 5, 6, 11}, [st|{4, 5, 6, 11}, {12}], {4, 5, 12}, [prof|{4, 5, 12}, {11}], [paper|{4, 5, 12}, {6}], [st|{4, 5, 12}, {6}], {4, 6, 12}, [prof|{4, 6, 12}, {5}, {11}], [prof|{4, 6, 12}, {5, 11}], [st|{4, 6, 12}, {5}], [st|{4, 6, 12}, {5, 11}], {4, 5, 6, 12}, [prof|{4, 5, 6, 12}, {11}], {4, 5, 11, 12}, [paper|{4, 5, 11, 12}, {6}], [st|{4, 5, 11, 12}, {6}], {4, 6, 11, 12}, [prof|{4, 6, 11, 12}, {5}], [st|{4, 6, 11, 12}, {5}], {4, 5, 6, 11, 12}

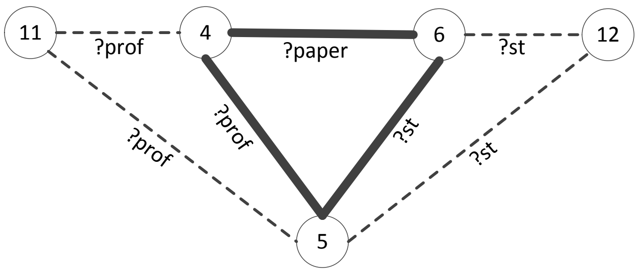


Figure 3.9: *Planner's execution trace for Q_e*

Cache Controller module is responsible for monitoring cache requests and maintaining detailed benefit estimations for possibly usable query patterns as well as for their materialization cost. This information is stored in the *Cache Requests* table and is used to trigger the execution and caching of *profitable* queries (frequently requested but not cached queries) in order to boost the cache utilization.

The above approach introduces two main challenges:

1. Our dynamic programming planner runs on top of the simplified join graph which contains star nodes, representing entire star subgraphs of the query in hand. Therefore, our planner's enumeration of connected query subgraphs is restricted to the enumeration of connected skeleton-star query subgraphs. This restriction lowers the complexity of our query planner but it introduces a challenge of locating *all* possibly usable cached results

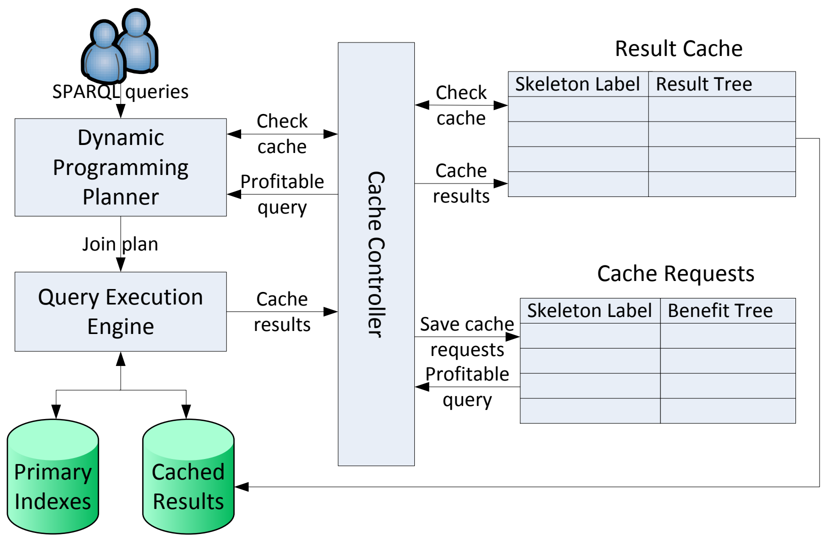


Figure 3.10: *System architecture*

because it avoids the enumeration of the star subgraphs. To alleviate this, we utilize our skeleton-star canonical labels in order to perform skeleton-star subgraph isomorphism tests, presented in Section 3.3.3, when checking for usable cached results. Our planner enumerates *all* skeleton-star query subgraphs. This enumeration, along with our cache check mechanism that examines all skeleton-star subgraph isomorphic cached results, ensures that *all* cached results matching a subgraph of the query in hand are examined.

2. The check cache mechanism should be able to examine not only cached results that exactly match query subgraphs but also more general results that can be used to provide the input of a query subgraph by applying filtering or projection operations on them. In this case, we also need to take into account optional pattern properties and cached result indexing that can help reduce the filtering operation overhead. To tackle this problem, we utilize a query abstraction technique presented in Section 3.5.1.

3.5.1 Query abstraction

In this section, we describe how we can locate and check the usability of results that can be used to provide the input of a query subgraph by applying filtering operations on them. Lets examine the following query:

```
?prof worksFor ”MIT” .
?prof emailAddress ?email .
?prof name ”Mike” .
```

Using exact matching, our planner would only issue cache requests for query subgraphs that contain all the bound literals and URIs of the initial query, depicted below:

?prof worksFor “MIT” . ?prof name “Mike” .	?prof worksFor “MIT” . ?prof emailAddress ?email .
?prof emailAddress ?email . ?prof name “Mike” .	?prof worksFor “MIT” . ?prof emailAddress ?email . ?prof name “Mike” .

However, we notice that the following indexed cached result, while not examined, could also be beneficial for answering the query, transforming it to a simple index lookup:

```
{ ?prof worksFor ?univ .
?prof emailAddress ?email
?prof name ?name } index by ?univ ?name
```

To address this scenario, we need to also examine more general graphs as well as their indexing properties that can help reduce the filtering operation overhead. For example and in the above case, caching the general result without any indexing might be useless, if for instance its size was significantly larger than the size of the filtered results, as we would have to read all its data in order to find the relevant ones. In addition, the usability of a cached result also depends on the join operation that must be applied on it. For example, if we need to join this result with another triple pattern according to variable `?prof`, we would like to have it sorted in order to perform a merge join operation and not a more complex and inefficient hash or sort-merge join operation. Furthermore, we need to take into account the case of cached results or queries that contain more complex filters on variables, projection and group-by clauses. To achieve all these goals, we need to find an efficient way to examine which of the cached results can provide input for a query subgraph, due to filtering, projection and grouping properties, and search them to find the one that incurs the least cost.

To tackle this problem, we *abstract* our query graph. We remove all bound query nodes that reside in the subject or object position of a triple query and replace them with variables along with the respective equality filters. In the above example, the query would be transformed to:

```
{ ?prof ub:worksFor ?univ .
?prof ub:emailAddress ?email .
?prof ub:name ?name } filter(?univ=”MIT”, ?name=”Mike”)
```

We use this abstract query graph as input for our dynamic programming planner and thus check all its subgraphs issuing cache requests. Each cache request is issued having as key the canonical skeleton-star label of the abstract query subgraph and is accompanied by the filter,optional pattern, projection and group-by clauses of the query as well as by a request for a join variable. As mentioned in Section 3.3, filters, optional patterns, projections and groupings are not used to generate the canonical label and thus queries with the same abstract structure will be grouped together using their label.

3.5.2 Result Tree

The use of canonical labels to access the *Result Cache* reduces the results that we need to examine for every query subgraph but we still need an efficient way to select the best result from the list of all existing results that share the same abstract skeleton structure. Each cache record, related to an abstract skeleton label, maintains all cached results that share this label using a tree structure, called *Result Tree.* This structure can then be used: i) to efficiently perform skeleton-star subgraph isomorphism tests and ii) to efficiently search for the best result with respect to the request's auxiliary information(filters, projections, groupings). Fig. 3.11 depicts this structure for the abstract skeleton query graph used in Section 3.2:

```
?3 author ?2 .
?3 advisor ?1 .
?1 author ?2 .
```

For better understanding, we use the canonical variable names, presented in Section 3.3.1. In this example, we assume that our cache contains several cached results with the two following abstract star labels:

```
?1 fullName ?4 .
?1 emailAddress ?5 .
?1 studiesIn ?7 .
?1 type ?6 .
```

```
?1 fullName ?4 .
?1 emailAddress ?5 .
?1 studiesIn ?6 .
```

The first edges of the tree encode the abstract star labels of the existing cached results. Each other edge contains the filtering, optionality and indexing information related to a specific query variable. Each leaf of the tree represents a cached result and, therefore, the path that connects it with the root node encodes all its auxiliary information. For example, an edge with label *?4{"*, Index"}* means that the result contains no filter on this variable and provides an index for it. An edge labeled *?6{="MIT"}* denotes that the result contains the filter *?6="MIT"*. A "Π" edge denotes that the respective variable is projected out in the result while an "*O*" edge denotes the optionality of a specific pattern. If the query contains a group-by clause it is encoded as a last level edge in the cached result tree. To check the usability of cached results for a cache request, we can traverse the tree from the root node and find the results that can be utilized to answer it by following only the edges that provide more generic results than the query request. When crossing star label edges, we perform a skeleton-star subgraph isomorphism test in order to only follow the subtrees that can be used to provide input for the respective result.

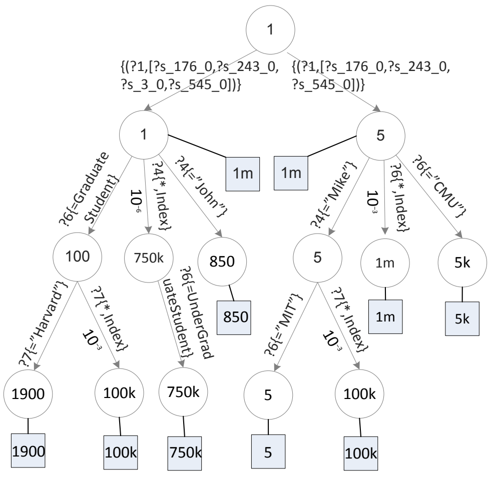


Figure 3.11: *Cached Result Tree*

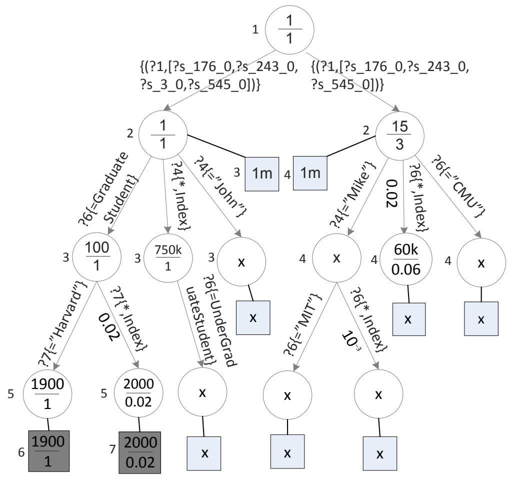


Figure 3.12: *Search Result Tree for Q_r*

Apart from checking the usability of cached results for a cache request, we also need to be able to evaluate their cost and find the result that best matches each cache request. As cost of a cached result, we refer to the amount of records that need to be accessed for using it. To estimate this cost in the presence of several indexed variables and query filters, we need to extend our tree structure with selectivity estimations and result sizes. The procedure of inserting a cached result in the result tree can be seen in Algorithm 10. The *treeInsert* method recursively adds a new leaf node, representing the current result, along with its size in number of records. The *minResults* value of each tree node, visible inside each node of Fig. 3.11, represents an estimation of the lowest cost that can be achieved by the results of its respective subtree. Having guarantees for the lowest cost of a subtree, we can perform the search for a cache request using efficient A* search and prune entire subtrees that do not contain relevant results. To create the minimum cost estimations, each leaf node contains the actual result size; we propagate this value to the parent nodes by keeping each time the minimum value among all children. In the case of indexed edges, we apply the selectivity estimations before we propagate the value from the child to the parent node. If a request contains a filter on a variable that is indexed, we need to reduce the cost of the result by the expected selectivity of the filtering operation on the index.

Edges that contain no indexes do not change the cost of the result because we would need to access all their data to perform the filtering operation.

Algorithm 10: $cacheResult$

```
function $cacheResult(V, E, aux, size)$
    //$V, E$ : vertices and edges of the abstract query graph
    //$aux$ : auxiliary query info(filters, projections, etc)
    //$size$ : the size of the result in records
    $(skeletonLabel, starLabel) \leftarrow canonicalLabel(V, E)$;
    $resultTree \leftarrow ResultCache.get(skeletonLabel)$;
    $aux.addStarLabel(starLabel)$;
    $treeInsert(resultTree.root, aux, size)$;
function $treeInsert(node, aux, size)$
    if $aux.isEmpty()$ then
        $createLeaf(size)$;
    else
        $aInfo \leftarrow aux.getNext()$;
        if $((edge, child) = node.getEdge(aInfo)) = null)$ then
            //Edge does not exist, create new
            $(edge, child) = newEdge(aInfo)$;
        end
        $treeInsert(child, aux, size)$;
        if $aInfo.isIndexed()$ then
            //compute the maximum selectivity of the index
            $maxSel \leftarrow maxSelectivity(edge)$;
            $results = child.minResults * maxSel$;
        else
            $results = child.minResults$;
        end
        if $results < node.minResults$ then
            if $results \leq 1$ then
                $results \leftarrow 1$;
            end
            $node.minResults = results$;
        end
    end
```

Our cache implementation does not depend on the specific way that an RDF execution engine handles selectivity estimations. To create the cached result tree we only require the maximum selectivity that can be achieved by doing a filtering operation on a certain index. For each indexed edge of a cached result, we generate a selectivity value $s = minRecords/totalRecords$, depicted along the indexed edges of Fig. 3.11. In our running example, an index on the general result(1 million records), on variable `?7 (?univ)` has a minimum amount of 1000 records and thus its maximum selectivity is $s = 1000/1m = 10^{-3}$. To handle the estimation of multiple filtering operations on different variables we follow the independency assumption, i.e., the selectivity of two filtering operations with selectivities s_1 and s_2 is $s = s_1 \cdot s_2$. This property allows us to follow paths of filtering operations on the cached result tree and maintain a total estimation by just multiplying the individual selectivities while crossing indexed edges. The *treeInsert* method (Algorithm 10) starts by adding the respective leaf along with its amount of records. We then move from the leaf to the root node and update the minimum values of the parent nodes. When crossing an indexed edge we apply the maximum selectivity by multiplying it with the child's number of records. We also set a minimum value of 1 for the costs of nodes. The complexity of the add operation is linear to the size of the auxiliary query info, which is bound by the amount of variables contained in the abstract query pattern.

As mentioned before, we use the minimum cost estimations of the tree nodes in order to perform an A* search and prune subtrees with large costs. The search procedure starts from the root node and checks all nodes in a best-first search according to cost estimations. Fig. 3.12 depicts the execution of the *searchTree* method (Algorithm 11) when searching for the following cache request (Q_r):

```
{ ?3 author ?2 .
?3 advisor ?1 .
?1 author ?2 .
?1 fullName ?4 .
?1 emailAddress ?5 .
?1 studiesIn ?7 .
?1 type ?6 .
} filter(?6=GraduateStudent, ?7="Harvard")
```

In Fig. 3.12, each node contains two numbers; its estimated cost (upper number) and its selectivity (lower number). When crossing tree edges our Algorithm checks the following properties:

- Whether the specific edge can be used for answering the request. For example, the edge with label *?4="Mike"* will not be followed because it is more restrictive than the request. We also check the usability of optional query patterns by applying projection and not null filters.

- When crossing indexed edges that match with a filtering operation of the request, we need to apply their selectivity. For example, when crossing the edge *(?7: "*",Indexed")* we need to apply the selectivity of the filter *?7="Harvard"*. To do so, we consult the selectivity estimator of the abstract skeleton-star result for the respective variable. We

Algorithm 11: *CheckCache*

```
function $checkCache(V, E, aux, k)$
    //$V, E$ : vertices and edges of the abstract query subgraph
    //$aux$ : auxiliary query info(filters, projections, etc)
    //$k$ : search for the top-k results
    $(skeletonLabel, starLabel) \leftarrow canonicalLabel(V, E)$;
    $resultTree \leftarrow ResultCache.get(skeletonLabel)$;
    $aux.addStarLabel(starLabel)$;
    return $searchTree(resultTree, aux, k)$;
function $searchTree(resultTree, aux, k)$
    $results \leftarrow \{\}$;
    //$openNodes$ : priority queue with pairs (node, cost)
    $openNodes \leftarrow \{(resultTree.root, 1)\}$;
    while $openNodes \neq \{\}$ do
        //get the open node with the minimum cost
        $n \leftarrow openNodes.removeHead()$;
        if $(results.size = k)$and$(n.cost \geq results.maxCost)$ then
            //We have found k results and all open nodes have greater cost
            return $results$;
        end
        $processNode(n, openNodes, results, aux, k)$;
    end
    return $results$;
function $processNode(n, openNodes, results, aux, k)$
    for $(edge, child) \in n.children()$ do
        if $s = selectivity(edge, aux) > 0$ then
            $child.selectivity \leftarrow s * n.selectivity$;
            $child.cost \leftarrow child.minResults * child.selectivity$;
            if $child.isLeaf()$ then
                //maintain the k best results
                if $results.size < k$ then
                    $results.add(child)$;
                elseif $results.maxCost > child.cost$ then
                    $results.removeMaxAndAdd(child)$;
                end
            else
                $openNodes.add(\{child, child.cost\})$;
            end
        end
    end
```

use the abstract result to estimate selectivities because a tree edge can belong to several results with different sizes and we want to get an estimation of the maximum selectivity. In this case, the selectivity of *?7="Harvard"* is 0.02 because it is expected to return 20 thousand records and the abstract result contains 1 million records.

- When crossing join variable edges, we multiply the selectivity by 2, if the edge is not indexed, due to the fact that the overhead of performing a hash or sort-merge join algorithm instead of a merge join can be approximated by the time to read the input data twice [Neumann 10a].

- For star label edges, we need to perform a subgraph isomorphism test on the star labels in order to check the usability of the existing subtrees of results. If the star label exactly matches the request label we do not change the selectivity estimation. In the case of cached results that are subgraphs of the request, we need to change their selectivity in order to reflect the cost of joining it with the remaining star triple patterns. We also need to change the canonical variable name mapping for the specific subtree because some of the star variables change their canonical names due to the missing patterns. In our example, when following the right subtree the ?univ variable has canonical name ?6 and not ?7. Concerning the selectivity estimation, we set it to 2 plus the number of the unjoined triple patterns. This number estimates the cost of performing a sort-merge join on top of the cached result.

To estimate the selectivity of several consecutive edges we maintain a selectivity estimation for each open node and propagate it to children nodes by multiplying it when crossing indexed edges with filter operations.The minimum cost estimation for each node (upper value) is computed by multiplying its selectivity with the *minResults* estimation depicted in Fig. 3.11. The selectivity estimation of a cached result edge according to the auxiliary query info is described in detail using function $selectivity(edge, aux)$, Algorithm 12.

Our cost estimations can deviate from the actual costs due to the use of the abstract result selectivity estimator and our assumption of independency for filtering operations. Therefore, we perform a top-k search for results and then further examine the cost of each result, inside *DPccp*, using the detailed cost model of the execution engine. In Fig. 3.12, we perform a top-2 search, depicting on the side of each node the step in which it was opened. Nodes marked with a '$\times$' were not opened; grey coloured nodes depict the results. We observe that we required 7 steps to find the top-2 cached results and our search managed to prune a large part of the search tree due to auxiliary info mismatches and minimum cost estimations.

The irregularity of the A* search does not allow us to set a useful upper bound for this algorithm. Of course, an upper bound for the algorithm is the maximum size of the tree stored inside each cache record, but this bound does not take into account the intelligent pruning of the search space performed. In the worst case, the tree search will have complexity O(r), where r is the maximum amount of cached results that share the same abstract skeleton structure. We note here that r is also bound by the cache size constraints, as discussed in the next Section, and we can therefore expect that it will not grow limitless. Our *checkCache* (Algorithm 11),

Algorithm 12: *Selectivity estimation*

```
function $selectivity(edge, aux)$
    //compute edge selectivity for $aux$ info
    $selectivity \leftarrow 1$;
    if $edge.isStarLabel()$ then
        if $edge.skeletonStarSubgraphIsomorphic(aux)$ then
            $aux.setNewCanonicalVariableMapping()$;
            $selectivity \leftarrow 2 + patternsToBeJoined()$;
        else
            return $0$;
        end
    else
        for $a \in aux.get(edge.variable)$ do
            //check edge usability
            if $edge.subsumes(a)$ then
                if $a.isFilter$ then
                    if $edge.isIndexed$ then
                        //filter selectivity estimation using the abstract result estimator
                        $selectivity \leftarrow selectivity * filterSelectivity(a)$;
                    end
                else if $a.isJoinVariable$ and $edge.isNotIndexed$ then
                    $selectivity \leftarrow selectivity * 2$;
                end
            else
                return $0$;
            end
        end
    end
    return $selectivity$;
```

generates a canonical label and then performs a tree search in the respective result tree. Thus, the canonical labelling time will, in the worst case, dominate the checkCache mechanism due to its exponential time complexity and due to the fact that the tree search has bounded worst-case performance. However, as shown in our experimental evaluation, our checkCache mechanism is practical and presents acceptable time complexity for general and complex graphs.

3.5.3 Containment of SPARQL Queries

In this section, we present the query containment properties of our caching framework. We focus on conjunctive SELECT queries that contain filters, projections, groupings, orderings and optional patterns. Nested SPARQL queries, OWL reasoning as well as more complex functionalities such as property paths are not supported. Specifically:

- Our execution plans can utilize multiple cached results, each covering different non-overlapping subgraphs of the query. This feature extends related SPARQL query containment approaches that check the usability of a single cached pattern [Shu 13, Fard 14].
- A usable cached result can provide input for a subgraph of the query in hand. In current approaches, the new query has to be entirely answered using a cached result [Harbi 16, Shu 13, Fard 14].
- We extend the usability of cached results by checking for filter, projection, ordering, grouping and optional pattern usability. While filter usability is introduced in [Shu 13], addressing the rest of these features is a novelty of this work.
- Our dynamic programming planner, along with the skeleton-star isomorphism tests performed in our cache checks, ensure that *all* results that match a subgraph of the query are examined.

3.6 Cache Controller

In this section, we describe the functionality of our Cache Controller module. Its tasks are the generation and caching of *profitable* query patterns and the cache replacement strategy. As *profitable* query patterns we define queries that, even if not exactly issued by users or executed as intermediate results, could benefit the execution of the workload if cached. For example, consider a workload of queries having the same abstract query structure but random bound selective IDs. Caching only the intermediate results of those queries would not offer a great benefit to the workload because it would achieve cache hits only for queries that share the exact same IDs. In contrast, caching the abstract query pattern indexed according to the variable that contains the selective IDs would introduce benefits for all queries, in this special type of workload, transforming them to index scans. Apart from identifying abstract results and their indexing, our cache controller can intelligently identify cross-query frequent subgraphs and index them according to: 1) their most common filtering variables and 2) their most common join variables.

3.6.1 Generation of profitable cached results

As discussed in the previous section, our dynamic programming planner issues cache requests for all subgraphs of the abstract query. In addition, these cache requests are issued while optimizing the query plan and can thus be recorded along with estimations about their effect to the execution time of the respective query. Maintaining such a detailed log of cache requests provides valuable information about which query patterns can provide the most benefit to the

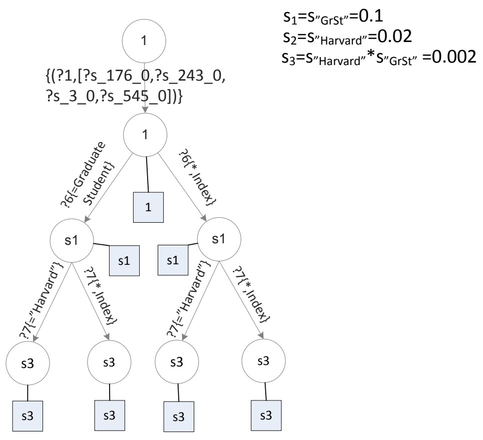


Figure 3.13: *Benefit Estimation Tree for the cache request (Q_r)*

workload. The exact benefit computation of each cached request $Q_i = (V_i, E_i)$ to the execution of a query $Q = (V, E)$ would require the execution of *DPccp* for each one of them. To avoid this, we use the following heuristic function, that multiplies the query's cost by the fraction of triple patterns covered by the subgraph, to compute a benefit estimation.

$$B(Q_i|Q) = \frac{|V_i|}{|V|} \cdot cost(Q) \tag{3.2}$$

This benefit estimation requires the optimal cost of the query Q and can be computed at the end of our planning process. Therefore, during the planning process our planner records all non satisfied cache requests (Q_i) and at the end compiles a list of requests along with their respective benefit and sends it to the Cache Controller for further processing. This means that the complexity cost of attributing benefits to possibly usable query patterns does not affect the execution and planning time of queries because it is done in an offline manner by the separate thread of the Cache Controller module. The Controller maintains a Cache Requests structure containing request benefit estimations indexed by their abstract skeleton canonical label. Each record holds a tree structure encoding benefit estimations for requests sharing the same label.

Fig. 3.13 depicts this benefit estimation tree, generated by the *addBenefit* method (Algorithm 14), for the cache request (Q_r) of the previous Section with benefit $B = 3sec$. Each leaf of the tree represents a query pattern that can be possibly utilized for the cache request. To attribute benefits to all possibly usable results, for all query filters we at least create the *"*, Index"* and the edge that contains the respective filter. Furthermore, if the record of the Cache Requests table already contained benefits for results with filters or optional patterns that can be

Algorithm 13: $executeQuery$

```
function $executeQuery(query)$
    $(q, aux) \leftarrow abstractQuery(query)$;
    //q: abstract query, aux: auxiliary query info
    $cacheRequests \leftarrow \{\}$;
    $plan \leftarrow DPccp(q, aux, cacheRequests)$;
    $results \leftarrow execute(plan)$; //RDF engine
    //handled offline by the CacheController thread
    $CacheController.cacheResults(results)$;
    $CacheController.addRequestBenefits(cacheRequests, qID)$;
    $CacheController.addResultBenefit(q, plan)$;
function $cacheResults(results)$
    //cache computed results
    for $result \in results$ do
        //get the respective benefit from the cache requests
        $request \leftarrow CacheRequests.get(result)$;
        $result.benefit \leftarrow request.benefit$;
        $result.queryIDs \leftarrow request.queryIDs$;
        $cache(result, result.benefit)$;
    end
function $addRequestBenefits(cacheRequests, qID)$
    for $(req, benefit) \in cacheRequests$ do
        $addBenefit(req.skeletonLabel, req.starLabel, req.aux, benefit, qID)$;
    end
function $addResultBenefit(q, plan)$
    //update the benefit of utilized cached results
    for $result \in plan.usedCachedResults$ do
        $newPlan \leftarrow DPccpWithoutResult(q, result)$;
        $result.benefit+ = (newPlan.cost - plan.cost)$;
    end
```

used for the current request, they are also followed. For example, if a previous query had a regular expression filter $?univ = “H*”$ we would also attribute benefits to its subtree. Concerning star label edges, if the existing tree contains a skeleton-star subgraph isomorphic subtree, then it is also followed in order to attribute benefit to its leafs. When existing star labels can be utilized, we also generate more generic labels that contain both the existing and the new star patterns. In this merge process, we use OPTIONAL patterns to handle mismatches. This approach is used to create more generic cached results by recording and augmenting star patterns that are frequently accessed. The values inside the nodes represent the selectivity estimations for the respective pattern. To generate these estimations we only require a selectivity estimator of the abstract result and the total amount of records of the abstract result. We again use the independency property to estimate the selectivities for multiple filters on different variables. Therefore, we just need to propagate the selectivity estimation from the parent to the child

Algorithm 14: $addBenefit$

```
function $addBenefit(skeletonLabel, starLabel, aux, benefit, qID)$
    $//skeletonLabel, starLabel$ : the canonical labels of the abstract query graph
    $//aux$ : auxiliary query info(filters, projections, etc)
    $//benefit$ : the estimated benefit for the query
    $//qID$ : the query ID
    $aux.addStarLabel(starLabel)$;
    $benefitTree \leftarrow CacheRequests.get(skeletonLabel)$;
    $treeAddBenefit(benefitTree.root, aux, benefit, 1, qID)$;
function $treeAddBenefit(node, aux, benefit, s, qID)$
    $//s$ : parent node selectivity
    if $aux.isEmpty()$ then
        $addChild(benefit, s, qID)$;
    else
        $aInfo \leftarrow aux.next()$;
        if $aInfo.isStarLabel()$ then
            $newEdgeIfNotExists(aInfo.getLabel())$;
        else
            $addChild(benefit, s, qID)$;
            $newEdgeIfNotExists(``*, Index")$;
            $newEdgeIfNotExists(aInfo)$;
        end
        for $(edge, child) \in node.children()$ do
            //check usability, selectivity of existing edges
            $selectivity \leftarrow s * selectivity(edge, aInfo)$;
            //prune subtrees with $benefit \leq 0$
            if $(benefit - selectivity * R/thr) > 0$ then
                $treeAddBenefit(child, aux, benefit, selectivity)$;
            end
        end
    end
```

node by multiplying with the selectivity of the respective edge. The benefit for each usable result, leaf node, is:

$$b = B - s \cdot R/thr \tag{3.3}$$

where B is the total benefit of the request, s is the selectivity of the result, R is the number of records of the abstract query, and thr is the engine's read throughput (e.g. 100k records/sec). The second part of the equation represents the cost to read the result. In Fig. 3.13, the benefit for a result with selectivity s_3 is $b_3 = 3sec - s_3 * 1m/100k \simeq 3sec$. We also ignore benefits that are less than 0. For example, the non indexed abstract result, higher leaf, has selectivity 1 and benefit $b = 3sec - 1 * 1m/100k = -7sec$. When the benefits of all patterns are estimated, the Cache Controller sums the previously existing benefit values with the new ones and stores the

Algorithm 15: $CacheControllerPeriodicProcess$

```
// methods that run in configurable time or query intervals;
function $updateBenefits()$
    //$OrderedResults$ : benefit ordered list of cached results
    for $result \in ResultCache$ do
        //decrease benefit with time using decay paramater $0 < a < 1$
        $result.benefit = result.benefit * a$;
        $OrderedResults.insert(result)$;
    end
    //$OrderedRequests$ : benefit ordered list with max size
    for $request \in CacheRequests$ do
        //decrease benefit with time
        $request.benefit = request.benefit * a$;
        $OrderedRequests.insert(request)$;
    end
    //remove cache requests that are not in OrderedRequests
    $removeFromCacheRequests(OrderedRequests)$;
    for $request \in OrderedRequests$ do
        //estimate cost for best requests using $DPccp$
        $request.benefit = request.benefit/estimateCost()$;
    end
function $profitableQueryGeneration()$
    //iterate in decreasing order of benefit/cost
    for $req \in OrderedRequests$ do
        //proactively check cache replacement
        $evict \leftarrow evictions(estimateSize(req), req.benefit)$;
        if $evict.satisfied$ then
            $result \leftarrow executeQuery(req)$;
            return;
        end
    end
```

tree inside the record of the Cache Results table. The Controller, running our planner, estimates the execution cost of the most prominent requests and maintains an ordered list of (request, benefit/cost) pairs (Algorithm 15) used by the *profitableQueryGeneration* method to trigger the caching of the most *profitable* queries.

The query pattern tree grows exponentially to the size of the auxiliary info of the query request and there are various computed and maintained benefits for results that will never be the most profitable. However, this exponential complexity does not affect our query response times because it is done independently of the query execution. To alleviate the problem of maintaining every benefit estimation, we utilize an offline process (Algorithm 15) that runs in configurable timeframes (e.g., every 10sec or every 10 queries) and maintains only the top-k leaf nodes of the Cache Requests table. Furthermore, to avoid promoting only queries with large costs, we normalize the benefit estimation of a query pattern using its execution cost. Thus,

Algorithm 16: $cachingPolicy$

```
function $cache(result, benefit)$
    $evict \leftarrow evictions(result.size, benefit))$;
    if $evict.satisfied$ then
        $removeCachedResults(evict)$;
        $cacheResult(result.V, result.E, result.aux, result.size)$;
        $decreaseBenefits(result.queryIDs)$;
        return;
    end
function $evictions(size, benefit)$
    //cache replacement policy
    $evict \leftarrow \{\}$;
    if $availableCacheSize \geq size$ then
        $evict.satisfied = true$;
        return $evict$;
    end
    //iterate cached results in decreasing benefit order
    for $result \in OrderedResults$ do
        if $result.benefit \leq benefit$ then
            $evict.add(result)$;
            if $evict.totalSize \geq size$ then
                break;
            end
        end
    end
    if $evict.totalSize \geq size$ then
        $evict.satisfied = true$;
    else
        $evict.satisfied = false$;
    end
    return $evict$;
function $evictions(size, benefit)$
    for $request \in CacheRequests$ do
        $oldSize \leftarrow request.queryIDs.size$;
        //remove common query IDs
        $request.queryIDs = request.queryIDs \setminus queryIDs$;
        $newSize \leftarrow request.queryIDs.size$;
        $request.benefit = request.benefit * newSize/oldSize$;
    end
```

in the previous example, if we had queries that mostly targeted "$Harvard$", both the indexed version of the abstract result and the version that contains only records for "$Harvard$" would gather the same benefit but eventually the more specific result would be cached due to its lower cost of execution.

In addition, queries issue cache requests adding benefit to all their subgraphs. When one query pattern gets selected for caching, the corresponding benefit should be removed from all other patterns that were requested by the same queries as they were *satisfied*. Not reducing benefits for *satisfied* queries can lead to: 1) executing all subgraphs of a frequent query pattern, 2) difficulties in identifying new profitable cache requests due to obsolete benefit estimations of *satisfied* requests. To alleviate this problem, we maintain for each query pattern in the Cache Requests table a list of query IDs that attributed to its benefit. This list is used to reduce the benefit of cache requests after the execution of a *profitable* query. We reduce the benefit of each request by the fraction of its query IDs that belonged to the *profitable*'s query ID list. In the example above, if we decided to execute the abstract query indexed according to variable $?univ$, all query profits would go to zero because the *profitable* query gathered benefit from all workload queries. Thus, no more queries would be executed by the Controller. To avoid maintaining obsolete benefit estimations for cache requests, we additionally decay their benefit with time.

3.6.2 Cache replacement strategy

In this paper, we target disk based cached results keeping only their meta-data in main memory. Thus, the amount of the maintained cached results is only limited by the available disk space. However, the user can set limits on the disk space capacity dedicated for storing cached results. When a new cached result cannot be stored without exceeding those limits, we need to remove some of the existing results. To be able to intelligently select which result should be evicted from the cache, the Controller also maintains a benefit estimation for each cached result. This benefit estimation is updated using the method *addResultBenefit* (Algorithm 13) and differs from the one described in the previous section. After the execution of a query that used a cached result, the Controller executes the planner one more time, restricting the use of this cached result. This gives us the cost of the query without utilizing the respective cached result. The controller adds the difference between this cost and the actual cost to the result's benefit value. Benefits, also, decrease with time in order to avoid maintaining obsolete results. We prioritize cache evictions using Algorithm 16 that utilizes this benefit estimation. New results can only evict cached results that have less benefit. In addition, if the cache size constraints are violated, our controller will not execute new *profitable* queries unless their benefit is larger than the lowest cached result benefit. To evict multiple results, due to its size, a new result must have benefit greater than the sum of benefits of all evicted results.

3.6.3 Complexity analysis

For a more detailed description, Algorithm 13 describes the complete processing of a SPARQL query corresponding to Fig. 3.10. It starts with the abstraction of the query graph updating its auxiliary info(aux). It then calls the extended *DPccp* planner that issues cache requests and records their benefit using Equation 3.2. The optimal query plan is executed and then the CacheController thread handles the caching of produced intermediate results, the request and result benefit attribution. The *cacheResults* method tries to cache all computed results consulting our caching policy. The *addRequestBenefits* updates benefits for all recorded cache requests using Algorithm 14. Lastly, the *addResultBenefit* method updates the benefit of all utilized cached results. It computes their contribution to the query execution by checking the optimal execution time computed by the *DPccp* without the use of the respective cached result.

Our caching policy, described in Section 3.6.2, is depicted in Algorithm 16. The *evictions* function, called if the cache constraints are violated, tries to find a set of results that cover the new result size and have cumulative benefit that is lower than the benefit of the new result. Furthermore, the *decreaseBenefits* function is presented that handles the update of benefits after the execution of a profitable result mentioned in the end of Section 3.6.1.

Lastly, Algorithm 15 presents our periodic processes. The *updateBenefits* method is responsible for decreasing the benefit estimations for both cached results and requests through time. To do so it uses a configurable decay factor $0 < a < 1$. It also maintains the ordered lists of results and requests used in our other algorithms. In the case of cache requests, we remove requests that are not in the top-k most profitable ones and also compute their execution cost. The *profitableQueryGeneration* method iterates over all cache requests in decreasing order of benefit/cost and, consulting our caching policy, selects the most profitable request for execution. It proactively checks the caching policy using an estimation of the query size, in order to avoid executing requests that cannot be cached due to their size and benefit constraints.

The *addRequestBenefits* method, presented in Algorithm 13, is responsible for attributing benefits to all possibly usable query patterns for all the cache requests issued during the planning of a query. The number of cache requests for a query graph is bound by the number of its connected subgraphs(csg) s. For every csg the addBenefit method is called (Algorithm 14). The worst-case time and space complexity of the addBenefit method is $O(d^v)$ where d is the maximum degree of the tree nodes (at least 3) and v is the number of variables in the query. In total, the worst-case complexity of the *cacheRequests* method is $O(sd^v)$. This worst-case complexity is dominated by: (i) the offline pruning of the benefit trees that maintains the top-k most profitable tree nodes for all query labels and (ii) the ability of the addBenefit method to prune entire subtrees with benefit ≤ 0.

The *profitableQueryGeneration* method, presented in Algorithm 15, iterates, in a benefit order, over the benefit estimations checking the caching policy in order to select the query that is going to be executed and cached. Our caching policy check has linear complexity, $O(res)$, in the amount of cached results res because, in the worst case, it needs to check the removal of all cached queries. Therefore, its worst-case complexity is $O(req \cdot res)$ where req is the number of maintained request benefit estimations. As stated above, this number is regulated by our offline request pruning process in order not to grow exponcentially. The res parameter is also regulated by the cache size constraints.

Our *updateBenefits* method, presented in Algorithm 15, has complexity $O(res \cdot \log res + req \cdot creq + req \cdot \log req + req \cdot dp)$. It sorts res cached results, it maintains and sorts req of the $creq$ accumulated requests. It also executes req times the *DPccp* algorithm, with complexity dp as presented above. The main memory space complexity of our caching framework is $O(res + req + creq)$ which is regulated using the configurable res and req values. The $creq$ value is regulated by the execution frequency of the *updateBenefits* method.

3.7 Experiments

In this section, we present a detailed performance evaluation of the proposed RDF caching framework.

3.7.1 RDF engines

We have integrated our caching framework on top of three open source RDF engines:

- RDF-3X [Neumann 10a]: we have extended the latest version[3] (0.3.8) of the RDF-3X centralized engine. The respective experiments were conducted using an 8-Core Intel i7-4820K CPU at 3.90GHz, 64GB of RAM server, with 7.5TB of disk space.

- H_2RDF+ [Papailiou 13]: we have extended the latest version[4] (0.2) of H_2RDF+. All experiments on the H_2RDF+ system use a cluster of 10 worker nodes plus a single machine in the role of the Hadoop and HBase master. Each of the workers features a 2 Quad-Core E5405 Intel Xeon®CPUs at 2.00GHz, 8 GB of RAM and a 500GB disk, while the master has similar CPU/disk specs and 4 GB of RAM. Each worker is set to concurrently run 5 mappers and 5 reducers, each consuming 512MB of RAM. In our experiments, we used Hadoop v1.1.2 and HBase v0.94.5.

- TriAD [Gurajada 14]: we obtained and extended the latest version (1.0.1) of the TriAD engine. All TriAD experiments were conducted on the same 10-worker cluster, used for H_2RDF+.

Our caching framework is independent of the query execution engine and thus changes are mainly required inside the query planner module of the engines. RDF-3X and TriAD utilize the same implementation of the DPccp dynamic programming planner, which we extend

[3]`https://code.google.com/archive/p/rdf3x/downloads`

[4]`https://github.com/npapa/h2rdf`

Dataset	Triples	#S	#P	#O	Size
LUBM1k	153 M	23 M	18	17 M	26 GB
LUBM10k	1.4 B	222 M	18	165 M	276 GB
LUBM20k	15.2 B	2.3 B	18	1.7 B	549 GB
WatDiv-100M	109 M	5.2 M	86	18 M	19GB
WatDiv-1B	1.09 B	52 M	86	179 M	149GB
Yago2	188 M	10 M	98	51 M	26 GB
Bio2RDF	4.2 B	552 M	1,714	1.07 B	547GB

Table 3.1: *Dataset statistics*

with our caching functionalities. For H_2RDF+ we replace its planner with our multi-way join enumeration algorithm described in Section 3.4.1. Another important aspect of the implementations is the indexing and storage of cached results. In the case of H_2RDF+ non-indexed results are stored in plain HDFS files while indexed ones are stored in HBASE tables according to H_2RDF+'s indexing scheme. For both RDF-3X and TriAD we maintain all cached results using main-memory structures.

3.7.2 Datasets

To conduct a detailed experimental evaluation we used both real and synthetic RDF datasets of variable sizes. Table 3.1 records for each dataset, its total number of triples, its distinct number of subjects, predicates and objects (#S, #P, #O) and its raw size in GB. First, the LUBM[5] dataset generator creates RDF datasets with academic domain information, enabling a variable number of triples by controlling the number of *university* entities. LUBM is widely used to compare the performance and scalability of triple stores due to its ability to create arbitrarily large datasets. We used 3 LUBM datasets consisting of 1, 10 and 20 thousand universities respectively. However, recent work [Aluç 14] suggests that LUBM generates uniform data distributions while its queries are limited in complexity. WatDiv[6] is a test suite that generates data from skewed and non-uniform distributions and issues queries of varying structural characteristics and selectivity classes. We experimented with two versions of this dataset: WatDiv-100M and WatDiv-1B. In order to test our system with real-life RDF data we also utilized the Yago2[7] and Bio2RDF[8] datasets. Yago2 consists of real data gathered from various resources such as Wikipedia, WordNet, GeoNames, etc. Bio2RDF provides biological linked data connecting 24 different biological datasets.

[5]`http://swat.cse.lehigh.edu/projects/lubm/`
[6]`http://dsg.uwaterloo.ca/watdiv/`
[7]`http://yago-knowledge.org/`
[8]`http://download.bio2rdf.org/release/2/`

3.7.3 SPARQL query canonical labelling

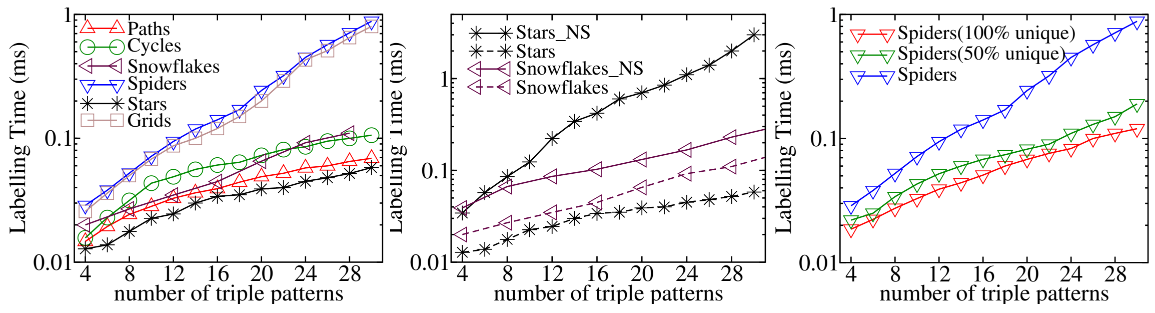


Figure 3.14: *Canonical labelling performance*

In this section, we evaluate our SPARQL query canonical labelling algorithm. We test its performance for various types of queries and compare it with the algorithm presented in [Papailiou 15], which is not using query simplification, denoted by a "_NS" suffix in the experiments. In the following, we report results using the RDF-3X implementation, while a comparison between the H_2RDF+, TriAD and RDF-3X implementations is presented in Section 3.7.5. The performance of the canonization process is tightly coupled to *Bliss*'s, with an added graph transformation overhead. The first graph of Fig. 3.14 depicts the time required to label different query graphs with variable number of triple patterns. We use a set of graph patterns consisting of paths, stars, cycles, snowflakes, grids and spiders, which contains patterns with ranging graph connectivity complexity [Aluç 14, Chen 97]. The utilized SPARQL queries were generated using the WatDiv benchmark. Ranging the size of the query graph, from 4 to 30, we observe that our algorithm can label all query graphs in less than 1 ms. Spider graphs and grids prove to be the most challenging cases because they contain more automorphisms and cannot be simplified. In contrast, star queries are entirely simplified and thus require the least amount of labelling time. The comparison of our skeleton-star simplification technique to the non-simplified algorithm [Papailiou 15] is depicted in the second graph of Fig. 3.14. Both star and snowflake graphs can be sufficiently simplified and their labelling times drop with respect to their simplification. In the case of star graphs, we observe a speedup of almost 2 orders of magnitude for queries with 30 nodes. Our algorithm performs a simple sort operation to label stars while, in the "_NS" case, a clique graph is passed to *Bliss* for labelling. Snowflakes contain multiple star patterns that are connected with each other only by their central nodes. Therefore, all their star patters are simplified, leading to a 2× performance gain.

The third graph of Fig. 3.14 depicts the effect of having different types of triple patterns in the queries. Different triple patterns provide *Bliss* with more information to prune the isomorphism search space. We measure the time needed to label the challenging spider queries while ranging their percentage of unique triple patterns. We examine queries with 0% (used in the

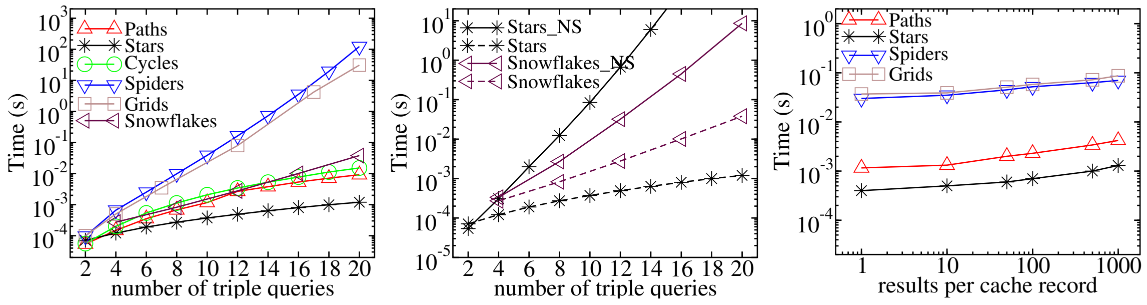


Figure 3.15: *Query planning performance*

previous experiments), 50% and 100% of the query's patterns being unique while the rest of the patterns are duplicates. We notice that the required labelling time decreases when queries contain more unique patterns. This gain is exponential to the number of query patterns, leading to almost one order of magnitude speedup for queries with 100% unique triple patterns.

3.7.4 Dynamic Programming Planner

In this section, we evaluate our dynamic programming planner. We report measurements for the RDF-3X implementation while the comparison of all our implementations is presented in Section 3.7.5. The first graph of Fig. 3.15 depicts the time required for our planner to generate canonical labels for all the query subgraphs, check for existing usable cached results and compute the optimal join plan for the set of query patterns utilized in the previous section. The complexity of our planner depends highly on the amount of connected subgraphs of a query. Path and cycle queries do not contain many subgraphs and therefore our planner is able to optimize queries with up to 20 relations in less than 15ms. Spider and grid query graphs, due to their more complex structure and the fact that they are not simplified, present higher complexity than paths and cycles resulting in sub-second response times for queries with up to 14 triple patterns. Star graphs and snowflakes are sufficiently simplified leading to optimization times of 1ms and 38ms respectively, for queries with 20 patterns. The comparison of our simplification technique ("_S") to the non-simplified version ("_NS) [Papailiou 15] is depicted in the second graph of Fig. 3.15. Clearly, the non simplified join graphs require exponential optimization times for star SPARQL queries. The join graphs of snowflakes are also highly connected and contain many subgraphs that increase their optimization complexity. In both cases, our simplification technique manages to offer gains to the query optimization complexity. We note that we can handle both star and snowflake queries with up to 20 patterns in less than 40 ms, while optimizing over non-simplified graphs required nearly 10 sec for stars with 14 patterns and snowflakes with 20 patterns.

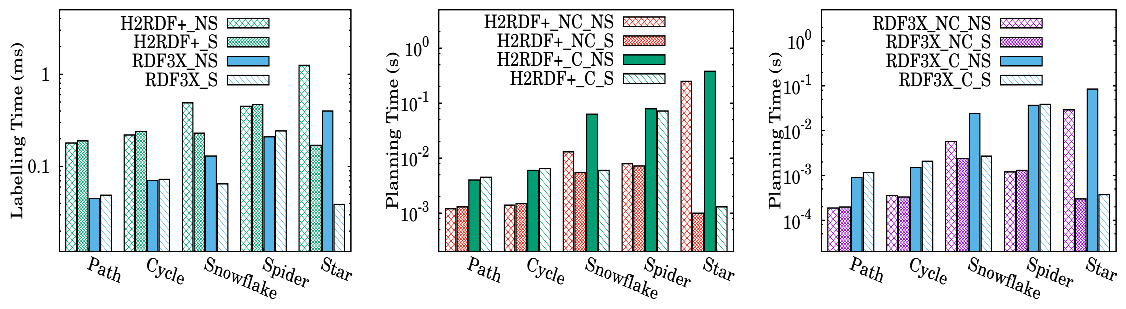


Figure 3.16: *Labelling and planning times for RDF-3X, TriAD and H_2RDF+*

Another major parameter that affects the complexity of our planner is the tree search for cached results inside a cache record. This procedure mainly depends on the amount of maintained cached results. As explained in Section 3.5.1, we perform a top-k A* search to find the best result inside each cache record. The third graph of Fig. 3.15, depicts the effect of the amount of cached results to the planners execution time, using top-3 search. We select a set of query graphs all consisting of 10 triple patterns and range the number of cached results per cache record by randomly generating and loading 1 to 1k cached results. We assume that 1 thousand results per unique abstract query subgraph suffice to evaluate our planner's performance even for the most challenging caching scenario. We note that the performance of our planner scales well to the amount of cached results. This means that our A* search manages to effectively prune large parts of the result trees. The time needed for our planner to evaluate the same query in the presence of 1 and 1000 cached results per cache record only increases by a factor of 3 for all query patterns.

3.7.5 Comparison of different implementations

Our caching algorithms have been integrated with 3 RDF engines. RDF-3X and TriAD are based on the same C++ planner, initially created for RDF-3X. Therefore in the planning and labelling results we omit the TriAD measurements as they are similar to the ones reported for RDF-3X. In contrast, H_2RDF+ is implemented in Java and *Bliss* in C++. In this section, we evaluate the performance deviations among the different caching and planning implementations. The first graph of Fig. 3.16 depicts the labelling time required for different query graphs, all consisting of 10 triple patterns. As we observe, the H_2RDF+ implementation has a stable overhead compared to RDF-3X, which is due to the better integration of the C++ based code of *Bliss*. However, we note that, for all engines, our query simplification achieves great speedups for stars and snowflakes while introducing minor overheads for queries that cannot be simplified (paths, cycles and spiders).

Dataset	Workload Name	Query Types	Distinct Queries	Total Queries	Description
WatDiv	L	5	588	5000	linear queries
WatDiv	S	7	900	5000	star queries
WatDiv	F	5	464	5000	snowflake queries
WatDiv	C	3	27	5000	complex queries
WatDiv	G	20	1979	20000	general, all query types
LUBM	S	6	4621	10000	selective queries
LUBM	C	3	19	10000	complex queries
LUBM	SG	5	897	10000	subgraph pattern queries
LUBM	G	14	3768	10000	general, all query types
Yago2	G	7	8407	20000	general, all query types
Bio2RDF	G	4	7225	10000	general, all query types

Table 3.2: *Workloads*

The second and third graph of Fig. 3.16 depict the planning times for both H_2RDF+ and RDF-3X, for different types of queries, all consisting of 10 triple patterns. We report measurements for 4 different planning implementations: i) no caching and no simplification(_NC_NS), ii) no caching and simplification (_NC_S) [Gubichev 14], iii) caching and no simplification (_C_NS) [Papailiou 15], and iv) caching and simplification (_C_S). We observe that, in both engines, the simplified versions manage to offer speedups for stars and snowflakes while introducing small overheads for the rest of the queries. Lastly, we note that planners can adopt result caching mechanisms with a small overhead (lower than $5\times$ in all cases), which decreases when query simplification is applied.

3.7.6 Query Workloads

To offer a detailed evaluation of our caching platform, we compile a set of SPARQL query workloads that spans various RDF datasets, query patterns and query complexities. Table 3.2, presents the workloads that we use in our evaluation. Starting with WatDiv, we generate for each of its query categories (L, S, F and C) one workload containing 5 thousand queries. We also merge all workloads together to generate the general (G) WatDiv workload consisting of 20 thousand queries. For LUBM, we create 4 workloads each containing 10 thousand queries. The S workload contains queries with selective triple patterns. The C workload consists of non-selective and complex query patterns. The SG workload contains complex queries with common subgraph patterns and the G workload contains a combination of the previous. We randomly select IDs for the bound nodes of each query ensuring high percentage of distinct queries within each workload. For Yago2 and Bio2RDF we generate two general workloads consisting of queries that match all the above categories.

3.7.7 Workload queries

In this section, we present all the patterns of the workload queries.
LUBM-S: <CID>: Course ID, <PID>: Professor ID, <DID>: Department ID, <UID>: University ID

```
select * where {?x type ub:GraduateStudent . ?x ub:takesCourse <CID>}
select * where {?x type ub:Publication . ?x ub:publicationAuthor <PID>}
select * where {?x ub:worksFor <DID> . ?x ub:name ?n . ?x ub:emailAddress ?em .
                ?x ub:telephone ?t . ?x type ub:Professor}
select * where {?x type ub:GraduateStudent . ?x ub:memberOf <DID>}
select * where {?x type ub:Student . ?y type ub:Course . ?x ub:takesCourse ?y . <PID> ub:teacherOf ?y}
select * where {?x type ub:Student . ?y type ub:Department . ?x ub:memberOf ?y .
                ?x ub:emailAddress ?em . ?y ub:subOrganizationOf <UID> }
```

LUBM-C: <DT>: Department type, <ST>: Student type, <PT>: Professor type, <CT>: Course type

```
select * where {?z type <DT> . ?x ub:memberOf ?z . ?x rdf:type <ST> .
                ?z ub:subOrganizationOf ?y . ?y type ub:University . ?x ub:undergraduateDegreeFrom ?y}
select * where {?x type <ST> . ?y rdf:type <PT> . ?z rdf:type <CT> . ?x ub:advisor ?y .
                ?y ub:teacherOf ?z . ?x ub:takesCourse ?z}
select * where {?p type ?tp . ?p ub:worksFor ?d . ?s ub:takesCourse ?c . ?p ub:teacherOf ?c}
```

LUBM-SG: <DT>: Department type, <ST>: Student type,<UID>: University ID

```
select * where {?z type <DT> . ?x ub:memberOf ?z . ?x rdf:type <ST> .
                ?z ub:subOrganizationOf ?y . ?y type ub:University . ?x ub:undergraduateDegreeFrom ?y}
select * where {?x ub:name ?n. ?x ub:emailAddress ?e . ?x ub:telephone ?t .
                ?z ub:subOrganizationOf ?y . ?y type ub:University . ?x ub:undergraduateDegreeFrom ?y}
select * where {?z ub:subOrganizationOf <UID> . ?y type ub:University .
                ?z ub:subOrganizationOf ?y . ?x ub:undergraduateDegreeFrom ?y}
select * where {?x type <ST> . ?y type ub:University .
                ?z ub:subOrganizationOf ?y . ?x ub:undergraduateDegreeFrom ?y}
select * where {?x ub:memberOf <DID> . ?z ub:subOrganizationOf <UID> . ?y type ub:University .
                ?x ub:undergraduateDegreeFrom ?y}
```

Yago2-G:<N>: name, <C>: city

```
select * where {?p y:hasGivenName ?gn . ?p y:hasFamilyName ?fn . ?p y:wasBornIn ?c1 .
                ?p y:hasAcademicAdvisor ?a . ?a y:wasBornIn ?c}
select * where {?p y:hasGivenName ?gn . ?p y:hasFamilyName ?fn . ?p y:wasBornIn ?c .
                ?p y:hasAcademicAdvisor ?a . ?a y:wasBornIn ?c . ?p y:isMarriedTo ?p2 . ?p2 y:wasBornIn ?c1}
```

```
select * where {?a1 y:hasPreferredName ?n1 . ?a2 y:hasPreferredName ?n2 . ?a1 y:actedIn ?m . ?a2 y:actedIn ?m}
select * where {?p1 y:hasPreferredName ?n1 . ?p2 y:hasPreferredName ?n2 . ?p1 y:isMarriedTo ?p2 .
                ?p1 y:wasBornIn ?c . ?p2 y:wasBornIn ?c}
select * where {?p1 y:hasGivenName <N> . ?p y:hasFamilyName ?fn . ?p y:wasBornIn <C> .
                ?p1 y:hasFamilyName ?fn . ?p1 y:hasGivenName ?gn . ?p1 y:wasBornIn ?c .}
select * where {?p1 y:hasGivenName <N> . ?a2 y:hasPreferredName ?n2 . ?a1 y:wasBornIn ?c1 .
                ?a2 y:wasBornIn ?c2 . ?a1 y:actedIn ?m . ?a2 y:actedIn ?m}
select * where {?p1 y:hasGivenName <N> . ?p2 y:hasPreferredName ?n2 . ?p1 y:isMarriedTo ?p2 .
?p1 y:wasBornIn ?c1 . ?p2 y:wasBornIn ?c}
```

Bio2RDF-G: <JID>: journal title, <DID>: description name, <ID>: identifier

```
select * where {?s label ?l . ?s type ?t . ?s identifier <ID> . ?s inDataset ?d . ?s mesh_heading ?h .
                ?h mesh_descriptor_name ?dn . ?h label ?l1 . ?h type ?t1 }
select * where {?s label ?l . ?s type ?t . ?s identifier ?id . ?s inDataset ?d . ?s mesh_heading ?h .
                ?h mesh_descriptor_name <DID> . ?h label ?l1 . ?h type ?t1 }
select * where {?s label ?l . ?s mesh_heading ?h . ?hlable ?l }
select * where {?s label ?l . ?s type ?t . ?s identifier <ID> . ?s inDataset ?d . ?s pb:journal ?j .
                ?j pb:journal_title <JID>. ?j pb:journal_volume ?v }
```

3.7.8 Caching different query patterns

In this section, we evaluate the efficiency of our caching framework using a diverse set of query patterns and caching approaches. We focus on the RDF-3X implementation and the WatDiv workloads. A comparison between all RDF engines and datasets is presented in Section 3.7.11. To evaluate all alternative caching techniques we test the following combinations:
1) *Original*: The original query engine without any caching.
2) *CS*: The discovery of a relational schema using characteristic sets (CS) can be exploited to improve SPARQL query efficiency [Pham 15]. To test the effect of using such approaches, before executing each workload we use the discovered schemas[9] to create the respective cached results. We also index the initial results according to primary and external keys and use optional query patterns for table columns that can receive null values.
3) *EX-NS*: caching with exact cache checks (without abstracting the query graph) and without query simplification [Papailiou 15].
4) *EX-S*: caching with exact cache checks and query simplification.
5) *AB-NS-PR*: caching with abstract cache checks, profitable result execution and without query simplification [Papailiou 15].

[9] `http://homepages.cwi.nl/∼boncz/emergent/`

Linear queries (type L)

	WatDiv-100M					WatDiv-1B				
RDF3X	**Plan**	**Check**	**Execute**	**Profitable**	**Total**	**Plan**	**Check**	**Execute**	**Profitable**	**Total**
Original	1.91	-	20.97	-	22.88	1.70	-	185.89	-	187.60
CS	2.23	0.13	19.03	-	21.40	1.71	0.24	150.44	-	161.91
EX-NS	1.93	0.52	2.14	-	4.59	1.62	0.52	18.37	-	20.52
EX-S	2.20	0.37	1.85	-	4.43	1.76	0.35	15.24	-	17.36
AB-NS-PR	1.95	0.25	2.04	0.08	4.25	1.62	0.50	13.59	0.05	15.78
AB-S-PR	2.21	0.21	1.77	0.04	**4.19**	1.75	0.25	9.05	0.34	**11.41**

Star queries (type S)

	WatDiv-100M					WatDiv-1B				
RDF3X	**Plan**	**Check**	**Execute**	**Profitable**	**Total**	**Plan**	**Check**	**Execute**	**Profitable**	**Total**
Original	5.73	-	6.10	-	11.83	4.65	-	49.12	-	53.76
CS	10.46	9.91	2.57	-	22.94	9.34	9.97	18.80	-	38.12
EX-NS	10.29	9.99	4.63	-	24.91	9.32	10.09	34.68	-	54.08
EX-S	4.48	0.45	4.92	-	9.84	3.46	0.45	38.53	-	42.44
AB-NS-PR	10.41	9.98	0.71	0.61	21.70	10.45	10.19	3.78	0.10	24.51
AB-S-PR	4.20	0.55	0.70	0.57	**6.02**	3.27	0.55	4.41	0.19	**8.42**

Snowflake queries (type F)

	WatDiv-100M					WatDiv-1B				
RDF3X	**Plan**	**Check**	**Execute**	**Profitable**	**Total**	**Plan**	**Check**	**Execute**	**Profitable**	**Total**
Original	5.19	-	74.67	-	79.86	4.69	-	1019.74	-	1024.42
CS	4.82	0.46	60.02	-	65.30	7.16	10.51	812.86	-	830.52
EX-NS	7.42	11.57	16.04	-	35.03	6.92	11.07	188.30	-	206.29
EX-S	4.43	1.22	16.93	-	22.57	3.84	0.71	201.60	-	206.15
AB-NS-PR	7.49	11.48	2.98	0.78	22.73	6.93	11.20	103.58	1.87	123.59
AB-S-PR	4.37	0.66	3.92	0.62	**9.56**	3.83	0.60	57.13	6.35	**67.91**

Complex queries (type C)

	WatDiv-100M					WatDiv-1B				
RDF3X	**Plan**	**Check**	**Execute**	**Profitable**	**Total**	**Plan**	**Check**	**Execute**	**Profitable**	**Total**
Original	1.17	-	7847.97	-	7849.14	2.52	-	54007.70	-	54010.30
CS	3.55	0.69	843.37	-	847.61	3.25	0.70	6278.80	-	6282.75
EX-NS	3.50	8.45	668.17	-	680.11	3.14	8.26	3872.80	-	3884.19
EX-S	2.32	3.96	671.86	-	678.13	2.29	3.62	3888.41	-	3894.32
AB-NS-PR	3.51	8.31	47.45	5.67	64.94	3.26	8.34	424.22	2.29	435.82
AB-S-PR	2.28	3.83	43.38	1.10	**50.59**	2.29	3.61	421.12	2.49	**427.02**

All queries (type G)

	WatDiv-100M					WatDiv-1B				
RDF3X	**Plan**	**Check**	**Execute**	**Profitable**	**Total**	**Plan**	**Check**	**Execute**	**Profitable**	**Total**
Original	14.00	-	7949.72	-	7963.72	13.57	-	55262.45	-	55276.09
CS	21.06	11.19	924.99	-	957.24	21.46	21.43	7260.91	-	7313.31
EX-NS	23.15	30.52	690.97	-	744.64	21.00	29.94	4114.15	-	4165.10
EX-S	13.43	5.99	695.56	-	714.98	11.35	5.13	4143.80	-	4160.27
AB-NS-PR	23.37	30.02	53.18	7.14	113.63	22.27	30.23	545.18	4.31	599.70
AB-S-PR	13.06	5.26	49.78	2.33	**70.38**	11.16	5.01	491.72	9.37	**514.76**

Table 3.3: *WatDiv execution times (sec) for alternative caching techniques*

5) *AB-S-PR*: caching with abstract cache checks, query simplification and profitable result execution.
Table 3.3 reports, for each WatDiv workload, the breakdown of time spent in each phase of the query execution. We measure the total time required for executing the entire workload and the amount of time spent in query planning, checking for cached results, actual query execution and profitable result execution. The results for all the WatDiv-1B and the WatDiv-100M-G workloads are also presented as stacked bar charts in Figures 3.17-3.22.

Starting with linear queries (L), both planning and check cache times are small for all systems. This is expected by their limited amount of subgraphs and the results presented in the previous sections. We note that, in this case, the overhead of checking for cached results is almost negligible and does not largely deviate, regardless of using query simplification or not. We also observe that abstract query caching and profitable results generation can speedup the workload execution by almost an order of magnitude, while building cached results using the relational schema (CS) is not effective providing approximately a 10% speedup. Linear queries are paths involving multiple tables, requiring joins even when using star-based CS indexing.

Inspecting star queries (S), a big percentage of the workload execution time is spent for planning and checking for cached results. This workload type highlights the effects of query simplification. Non-simplified versions spend almost $2\times$ and an order of magnitude more time in planning and checking the cache respectively. Simplifying the query graph can lead to less efficient query plans, this is depicted by the increase of execution time between NS and S versions. However, the reduction of planning times proves larger than this increase. Star queries are also a good match for the CS technique. In most cases, they refer to a specific table transforming the query to a simple scan. This is observed by the large reduction of the execution time between the Original and the CS versions. This workload also includes a high percentage of distinct queries, highlighting the effects of query abstraction which provides around $10\times$ improvement compared to the exact caching versions.

For snowflake queries (F), we observe a similar behavior in planning and caching times caused by query simplification. However, simplification leads to a large increase of actual execution times due to sub-optimal plans. Additionally, the effect of the CS approach is not that profound because snowflake queries often span more than two tables, requiring join execution.

Complex query workloads (C) dedicate most of their time in actual query execution. The CS technique takes advantage of the discovered schema and provides approximately $10\times$ speedup for this workload. However, the execution of such queries can be further improved by maintaining more complex cached results. Exact caching reduces the workload execution time by almost an order of magnitude, while abstract caching offers more than two orders of magnitude speedup.

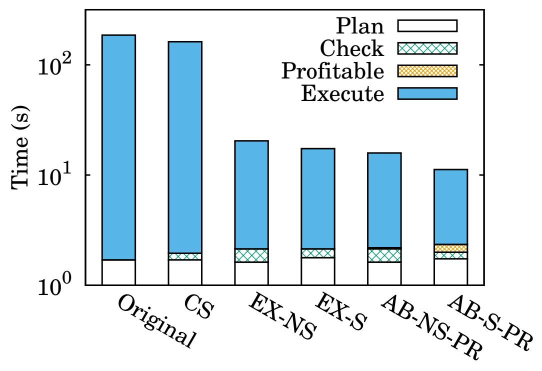


Figure 3.17: *WatDiv-1B-L*

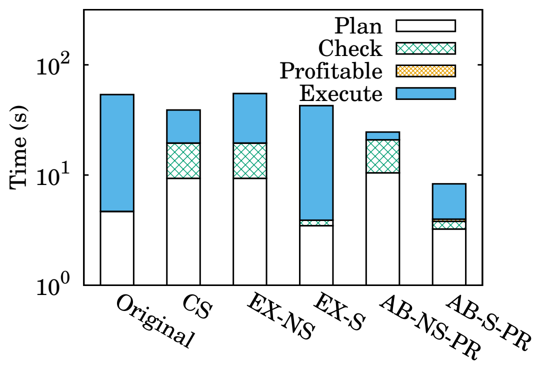


Figure 3.18: *WatDiv-1B-S*

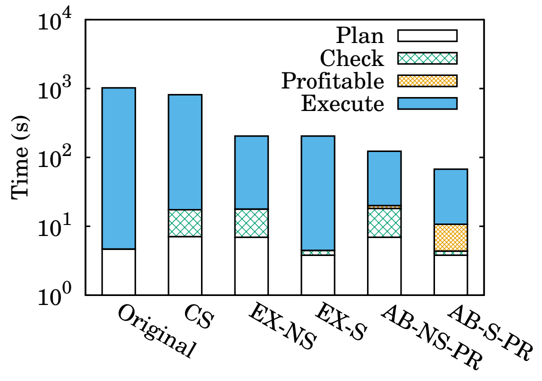


Figure 3.19: *WatDiv-1B-F*

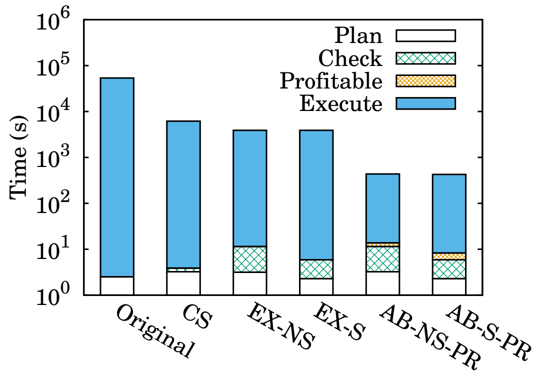


Figure 3.20: *WatDiv-1B-C*

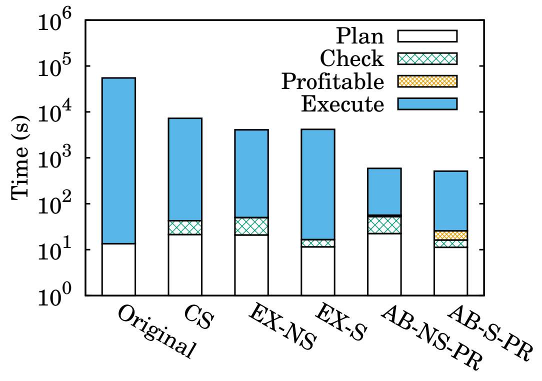


Figure 3.21: *WatDiv-1B-G*

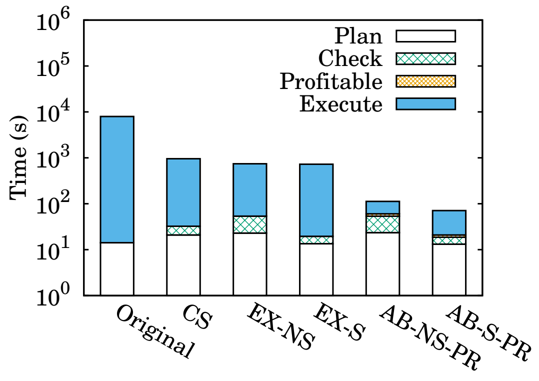


Figure 3.22: *WatDiv-100M-G*

Figure 3.23: *Execution time breakdown for various WatDiv workloads and different caching techniques*

To sum up, the characteristics of all discussed query categories affect the execution of the general workloads (G). As expected, complex queries occupy the majority of the actual execution time. In contrast, star and snowflake queries occupy the majority of the planning time.

However, query simplification effectively reduces planning and cache checking times. In addition, query abstraction is used to effectively trigger the caching of profitable results that improve execution times by almost two orders of magnitude.

3.7.9 Caching effect over time

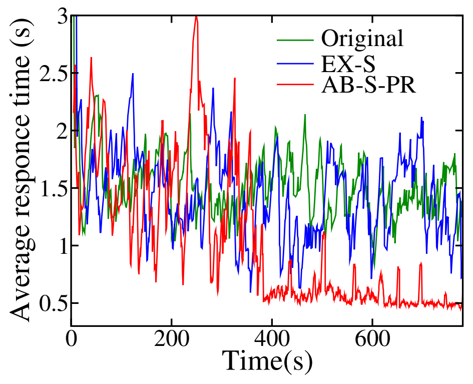


Figure 3.24: *LUBM10k-S*

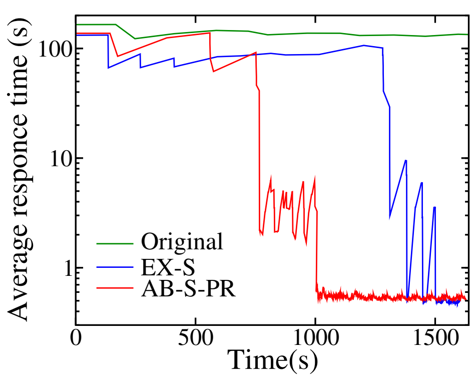


Figure 3.25: *LUBM10k-C*

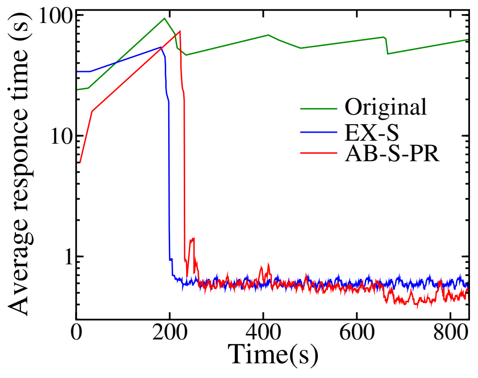


Figure 3.26: *LUBM10k-SG*

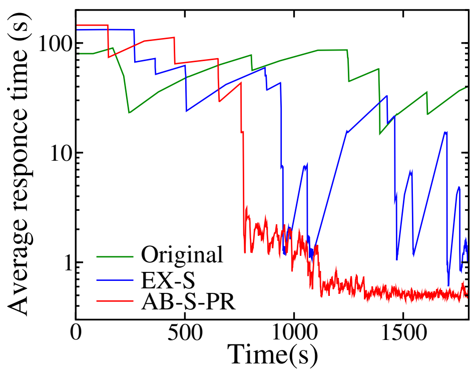


Figure 3.27: *LUBM10k-G*

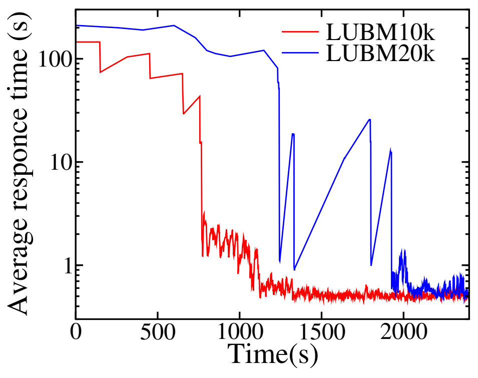


Figure 3.28: *LUBM-G*

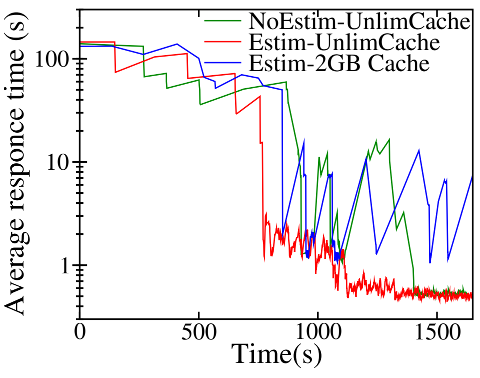


Figure 3.29: *LUBM10k-G*

Figure 3.30: *Caching effects over time for various LUBM workloads and different caching techniques*

In this section, we use all LUBM10k workloads and the H_2RDF+ system to test the caching effect with respect to time. For each workload, we examine the impact of our cache implementation to the average query response time. Figures 3.24-3.27 present the respective results for: 1) the baseline H_2RDF+ system, 2) our caching implementation with exact cache checks and simplification labelled "EX-S" and 3) our fully functional system with query simplification, abstraction and profitable result generation labelled "AB-S-PR". In all cases, we use unlimited cache size. We average query response times using a window of 10 queries in order to avoid large randomness and achieve quick responsiveness to the cache changes. Finally, our Cache Controller is configured to check the Cache Requests table and generate profitable queries every 10 sec.

Fig. 3.24 illustrates the results for the selective workload (S). Initially, we note that our baseline H_2RDF+ system has an average performance that ranges between 1 and 2.5 seconds. This is expected due to the use of centralized joins that take advantage of HBase indexes. This workload contains random selective IDs and high percentage of distinct queries. Therefore exact caching fails to effectively improve performance. In contrast, when using query abstraction we can effectively discover and cache profitable queries, resulting in an average response time of 0.5sec after 400 seconds.

The average response times for the complex query workload (C) are presented in Fig. 3.25. This workload is a lot more execution-intensive than the previous one, leading to an average response time of nearly 130 sec for the plain H_2RDF+ approach. However, our fully functional caching system is able to achieve interactive, millisecond-range response times for this workload in less than 1000 seconds (over 2 orders of magnitude reduction in the mean response time). The abstract caching technique only requires the execution and indexing of 3 results to offer interactive response times for this workload. The time required to execute these queries is around 400 sec. The additional time required to minimize the average response times is introduced by: 1) the fact that the result discovery process will not be very effective until enough entries are gathered in the Cache Requests table and 2) the concurrent execution of workload queries affects the execution time of profitable queries. We notice that the exact cache version can also benefit this workload but it needs nearly 1500 sec to do so. There are only 19 distinct queries in workload C and, thus, only when all of them are executed once the average response time drops.

Fig. 3.26 depicts the results of the subgraph pattern workload (SG). We observe that both caching versions are presenting similar average response time behaviors. This is due to the fact that the common query pattern, despite its execution complexity, is quite selective. Both caching techniques can detect and cache this inter-query dependency, leading to interactive execution times after only 240 sec. After caching the common subgraph, the abstract cache version will also continue caching indexed results for all the query types of the workload. This procedure leads to another small gain, which can be seen after 650 sec.

The caching efficiency for the general workload (G) is depicted in Fig. 3.27. We observe that this is a quite challenging workload for our baseline H_2RDF+ engine as it presents an average response time of around 60 sec. Exact caching is able to reduce the average response time by an order of magnitude resulting in average response times that oscillate around 3 seconds after 2000 seconds. Using query abstraction we manage to reduce the average response time by two orders of magnitude for this challenging workload. As mentioned in Section 3.6.1, our Cache Controller prioritizes the execution of profitable queries according to their expected benefit, resulting in a smooth and gradual query speedup.

3.7.10 Dataset Size and Caching Policy effects

Fig. 3.28 depicts the caching performance for LUBM10k-G and LUBM20k-G workloads. The major differences between the two cases are:

- The average response time in the beginning of the execution is larger for LUBM20k. When increasing the dataset size, the response time for non-selective queries increases respectively while the performance of selective queries remains almost stable. This is why the baseline average response time increases but remains less than double.

- The time needed for our caching framework to give milli-second-range average response times increases. While for LUBM10k our framework requires 1300 sec, 2000 sec are required for fully caching the general workload for LUBM20k. This is due to the fact that most of the *profitable* queries are non-selective, thus their execution time increases along with the dataset size.

To evaluate the effectiveness caching policy heuristics described in Section 3.6, we execute LUBM10k-G three times: 1) with unlimited cache and without using our benefit estimation to discover profitable queries, 2) with unlimited cache and using the proposed benefit estimation, 3) with benefit estimation and 2GB cache size. The experimental results are presented on Fig. 3.29. For the no estimation policy, instead of maintaining benefit estimations we just maintain a counter for each cache request, incremented by 1 each time the query graph is requested. We observe that both policies can eventually minimize the average response time of the workload but the proposed simple benefit-based approach manages to smoothly decrease the response times by caching the profitable queries in the most suitable order. In contrast, the policy that uses no benefit estimations leads to large deviations in the average response time due to the fact that costly queries happen to get cached later than selective queries. When limiting the cache size, we note that the large output of LQ15 cannot be cached, while all the rest cached results fit inside our cache, requiring nearly 1.7GB of disk space. LQ15 requires 182 sec to be executed by H_2RDF+. While all other queries present interactive response times after 800 sec, the execution of LQ15 triggers the oscillations of the average response time.

3.7.11 Caching on different RDF engines

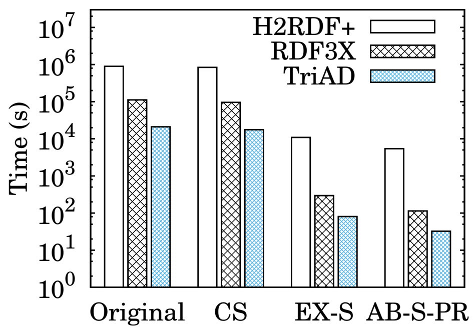


Figure 3.31: *LUBM1k-G*

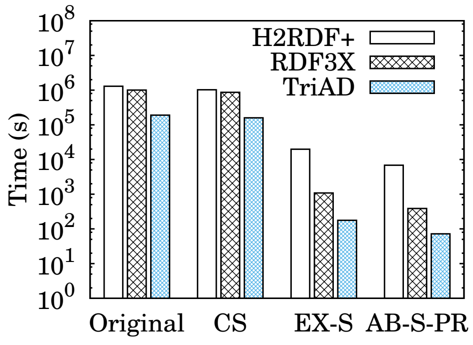


Figure 3.32: *LUBM10k-G*

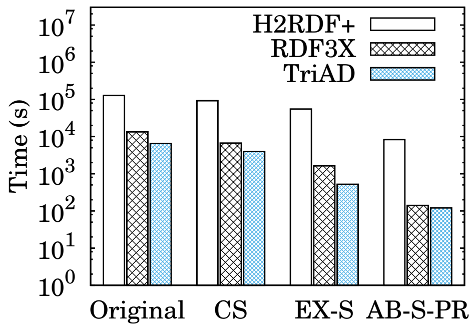


Figure 3.33: *Yago2-G*

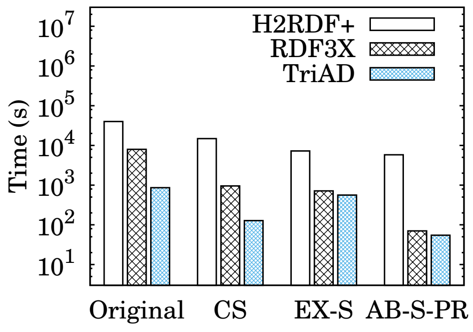


Figure 3.34: *WatDiv-100M-G*

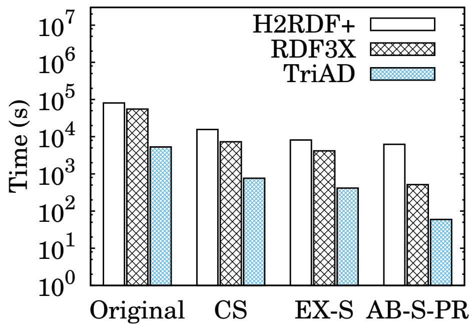


Figure 3.35: *WatDiv-1B-G*

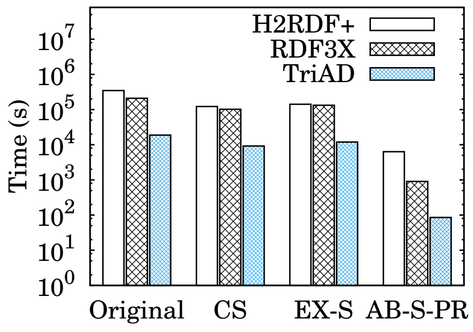


Figure 3.36: *Bio2RDF-G*

Figure 3.37: *Caching on different RDF engines*

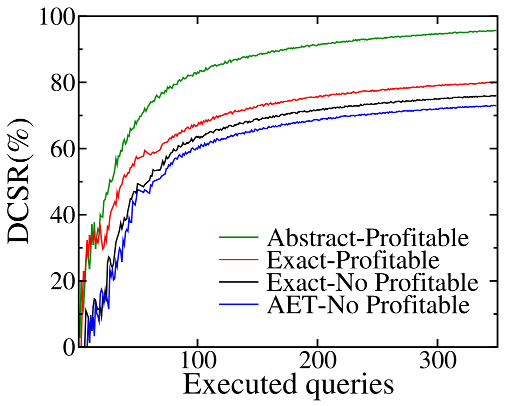


Figure 3.38: *Effectiveness of caching techniques*

In this section, we use the general workloads (G) of Table 3.2 and compare their total execution times on H_2RDF+, RDF-3X and TriAD. Figures 3.31-3.36 present the respective results. We note that TriAD proves faster in all workload cases due to its distributed main-memory query processing. Although H_2RDF+ performs centralized execution for selective queries, it fails to achieve the millisecond range response times offered by RDF-3X and TriAD. However, when the dataset sizes grows larger it gains performance by utilizing multiple resources. Regardless of the query engine, we observe that our caching techniques equally affect all systems and achieve up to 3 orders of magnitude speedup for challenging query workloads and large datasets.

3.7.12 Caching techniques comparisson

To present a more detailed comparison of all different caching techniques for RDF data we use the *Detailed Cost Saving Ratio (DCSR)* metric presented in [Kotidis 99].

$$DCSR = \frac{\sum_{i \in Q} s_i}{\sum_{i \in Q} c_i} \tag{3.4}$$

where c_i is the execution cost for query q_i without utilizing the cache and s_i is the savings provided by using the cache:

$$s_i = \begin{cases} 0, & \text{if } q_i \text{ does not use the cache} \\ c_i, & \text{if there is an exact match for } q_i \\ c_i - cf_i, & \text{if } q_i \text{ uses the cache and has cost } cf_i \end{cases} \tag{3.5}$$

DCSR captures the different effectiveness of the materialized results against the workload queries and can be used to accurately compare different caching techniques.

The $DCSR(\%)$ measurements, for our generic workload LUBM-10k-G, for four different RDF caching techniques are presented in Fig. 3.38. To compare our system to the related RDF caching techniques we choose as baseline the AET-based caching [Yang 11] which caches normalized join execution trees and is the most efficient among the related work systems. We also compare the effectiveness of our techniques for: 1) utilizing more general results using the abstract cache requests, 2) generating *profitable* results using our benefit estimations. Fig. 3.38, shows that our fully functional cache implementation can achieve the best cost saving ratio stabilizing to 96% after 350 queries. We also note that the cache efficiency is reduced by nearly 15% when removing our abstract request functionality and another 4% when removing the discovery of *profitable* results. Lastly, our system outperforms the AET based caching by nearly 24% due to its ability to utilize all possibly usable cached results. The cost savings achieved by the AET based technique are limited to the caching of non-selective queries that do not vary much across the workload. The efficiency difference compared to our cache would increase for workloads with more variable query types and filtering values.

CHAPTER 4

Network Data Analytics

4.1 Introduction

The Internet has become the dominant channel for innovation, commerce, and entertainment. Both Internet traffic and penetration increases at a pace that makes it difficult to track Internet growth and trends in a systematic and scalable way. Indeed, recent studies show that Internet traffic continues to grow by more than 30% annually as it has done the last twenty years and is expected to continue at the same pace in the future [Cisco 13]. Yet, operators and administrators have to perform fast and large scale analytics in huge network datasets to optimize parameters regarding network routing, dimensioning, accountability and security.

Network datasets collected at large Internet Service Providers (ISPs) and Internet Exchange Points (IXPs) have been at the forefront of network analytics. ISPs serve, depending on their footprint, thousands to tens of millions of end-users daily and facilitate billions of network connections [Poese 10]. IXPs consist of physical machines (core switches) to facilitate traffic exchange among different types of networks [Chatzis 13a]. Some of the most successful IXPs, connect more than 600 networks and are handling aggregate traffic that is peaking at multiples of TB per second. To put this traffic into perspective, on an average business day in 2013, one of the largest IXPs, AMS-IX in Amsterdam, exchanged around 25 PB while AT&T and Deutsche Telekom reported carrying 33 PB and 16 PB of data traffic respectively [Chatzis 13a].

To monitor traffic at that scale, specialized technologies such as sFlow[1] or NetFlow[2] are used. An sFlow record contains Ethernet frame samples and captures the first 128 bytes of each sampled frame. This implies that in the case of IPv4 packets the available information consists of the full IP and transport layer headers (i.e., source and destination IPs and ports, protocol information, and byte count) and 74 and 86 bytes of TCP and UDP payload, respectively. These records are then collected in a centralized location for processing. Typically data scientists that analyze these records rely on centralized approaches to execute queries. NetFlow does not capture part of the payload but only source/destination IP and port, interface, protocol, and type of service information. Centralized processing of sFlow or NetFlow is not scalable when it comes to processing multiple TB of data. Moreover, it is challenging to efficiently execute join or filtering queries, e.g., for a particular time period. Other related distributed approaches [Lee 13, Bumgardner 14] have been proposed to tackle the scalability issue, but failed to address the issue of fast execution of join and filtering queries.

To overcome the aforementioned problems related to both scalability and efficient query executions, we design and implement Datix: a scalable, network traffic data analysis system that can efficiently handle data in the form of a classic star schema [Gopalkrishnan 99]. Our system is based on distributed techniques of data analytics, such as MapReduce [Dean 08] and is capable of solving the more general problem of log processing as described in [Blanas 10]. Our goal is to implement efficient distributed join algorithms to combine the information from a main dataset, in our case being the sFlow data collected at an IXP, with additional complementary information provided by secondary datasets, such as the mapping of IP addresses to their corresponding AS, IP geolocation, and IP profile information, e.g., reverse DNS lookup as reported by Internet measurement studies such as ZMap [Durumeric 13].

Our contributions can be summarized as follows:

- We introduce a smart way of pre-partitioning the dataset in files that contain records of a certain range of values, so as to facilitate data processing and query execution.
- Using this particular partitioning scheme we are able to efficiently execute filtering queries, for example across a certain time period, avoiding the need to process the entire dataset but instead accessing only the necessary files.
- We integrate these features into Datix, an open-source[3], SQL compliant network data analysis system by implementing distributed join algorithms, such as map join [Blanas 10] in combination with custom-made user-defined functions that are aware of the underlying data format.

[1]`http://www.sflow.org/`
[2]`http://www.cisco.com/c/en/us/products/ios-nx-os-software/ios-netflow/index.html`
[3]`https://github.com/dsarlis/datix`

4.2 System Description

Datix operates on top of cloud or dedicated computing resources and runs user queries on data that reside in distributed file systems such as HDFS [Shvachko 10] or HBase [Chang 08]. It uses either Hive over Hadoop [Thusoo 09] or Shark over Spark [Zaharia 10] to run distributed MapReduce jobs, generated by translating the user queries from an SQL-like language (HiveQL). Datix divides input datasets in two categories. The first category is the central (*log*) dataset which in our case is the sFlow data collected at the IXP for its operational purposes. This dataset is the central table of our star-schema data formulation and needs to be joined with several other datasets (*meta-datasets*) on one or several of its columns. In our IXP data case, as well as in the general log processing case [Blanas 10], this dataset is expected to be orders of magnitude larger than the rest *meta-datasets*. Apart from the *log* dataset, Datix supports the import of several *meta-datasets* which, in this study, are the IP to AS and IP to Country mappings[4] and the IP to reverse DNS lookup mapping[5]. We choose to use these *meta-datasets* for our analysis because they are publicly available sources. Nevertheless, Datix supports the import of arbitrary *meta-datasets* even proprietary ones depending on each user's preferences. *Meta-dataset* sizes can range vastly according to the information they provide. In order to handle all *meta-dataset* cases, we divide them in two categories: (i) the small sized ones that are in the order of several MBs and can fit in the main memory of mappers (i.e., a single process entity in the MapReduce terminology) and (ii) the large sized ones which are bigger and do not abide to the main memory constraints of mappers. Small sized *meta-datasets* are stored in HDFS files while large sized ones are stored in HBase tables, indexed according to their star-schema join attributes.

By joining raw traffic data with *meta-data* using Datix, it is possible to efficiently answer important operational queries, such as heavy hitter queries, e.g., which are the most popular IPs, AS-pairs, or ports by volume or by frequency of appearance, summary queries, e.g., the aggregated traffic per AS or IP, or range queries, e.g., which are the IPs that are active in two different time periods (when denial of service attacks took place) that are responsible for more than 1% of the overall traffic and more than 3% of the HTTP request traffic.

Figure 4.1 illustrates the architecture of Datix. A graphical interface assists the user to provide custom query parameters. Then, Datix takes over and appropriately rewrites the user input to HiveQL compliant queries (upper part of Figure 4.1) that can be forwarded to the Processing Engine layer comprised by either Hive or Shark. The Processing Engine layer handles the query execution by translating the HiveQL input to a sequence of distributed processing jobs that take

[4]`http://dev.maxmind.com/geoip/legacy/geolite/`
[5]`http://scans.io/`

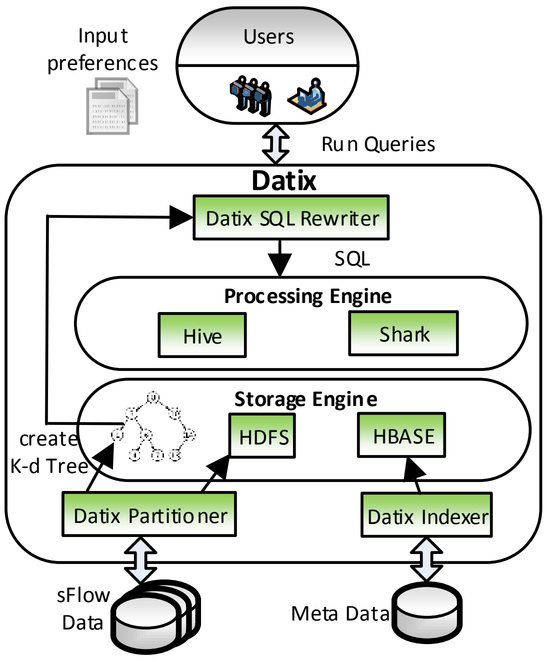


Figure 4.1: *Datix Architecture*

input from the required Datix datasets stored in HDFS or HBase (lower part of Figure 4.1). In detail, Datix consists of the following four layers:

Datix Partitioner/Indexer: This layer is responsible for pre-partitioning the datasets, according to user-specified partitioning attributes while utilizing a K-d Tree [Louis 75] (Section 4.3.3). These attributes can be either join attributes (e.g., source and destination IP) or filtering attributes (e.g., timestamp, protocol, port). Our pre-partitioning aims to achieve a number of optimization objectives:

- Meta-data required for the processing of each partition of the *log* data must fit in a mapper's main memory in order to perform efficient map-joins [Blanas 10].
- Large sized *meta-datasets* should be efficiently indexed in order for mappers to be able to retrieve their respective meta-data with minimum overhead.
- Apart from join attribute partitioning the *log* dataset must be partitioned according to filtering attributes that are used to efficiently run queries on a subset of the *log* dataset.

Furthermore, large *meta-datasets* are indexed using HBase.

Storage Engine: This layer stores and indexes all necessary datasets. Each partition of the *log* dataset is stored in a separate HDFS file while the partitioning information (K-d Tree) is also stored in HDFS. As mentioned before, small *meta-datasets* are stored in HDFS while larger ones are stored and indexed using HBase.

Processing Engine: This layer is responsible for the distributed execution of HiveQL queries that take input from the respective Datix datasets residing in HBase. In particular, either Hive or Shark produces a multi-step plan for executing the query in accordance with the number of actions requested.

Datix SQL Rewriter: This module is responsible for translating the user input preferences into a HiveQL query that both Hive and Shark can interpret and execute. The Datix Rewriter utilizes custom made user-defined functions in order to inject Datix related code inside the HiveQL query execution. It also consults the K-d Tree partitioning scheme, stored in HDFS, in order to apply filtering constraints and reduce the query input to only the required partitions. One of the desired properties of the K-d tree is to locate the portions of a dataset that has to be processed and thus avoid unnecessary processing (Section 4.3.3). Hence, with this scheme it is possible to efficiently execute range queries e.g., for a particular time period.

4.3 Description of Algorithms

In this section, we give an overview of the suite of distributed join algorithms currently implemented in Datix. We present the two major types of algorithms implemented: (i) when the *meta-dataset* is small enough to fit in the main memory of a map task (Sections 4.3.1, 4.3.2), and (ii) when the *meta-dataset* does not fit in memory (Section 4.3.3). Even though Hive and Shark support map-joins, there are some prerequisites that have to be fulfilled. First, map-joins need to be equi-joins and second, one of the two tables should be small enough to fit in a mapper's main memory. In the following sections, we describe how Datix manages to overcome the aforementioned restrictions in each case.

4.3.1 Map equi-join

Problem Statement: For two tables, a large table L and a small table S, the goal is to perform the equi-join $L \bowtie_{L.c=S.c} S$ on a specific characteristic (column) c of the two tables, with $|S| \ll |L|$, so that S can fit in the main memory of a mapper task. This assumption holds true for the case of IP to AS and IP to Country mapping files which are less than 12 Megabytes in size each.

Instead of using the basic, shuffle join, implementation of Hive which requires a lot of disk I/O and data transfer, we utilize a join method based on the map-join technique [Tang 11] and more specifically, on Broadcast Join as described in [Blanas 10]. Our map equi-join performs the join operation during the map phase. At each node, S is retrieved from HDFS and stored locally in the Distributed Cache of each mapper. Each map task then uses an in-memory hash table to join a split of L with the appropriate records of S.

In the beginning of a map task, it is checked whether S has already been stored in memory. If not, the hash table containing the key-value pairs of S is built. Then, while each record of L is processed, the map function extracts the join key and searches the hash table to produce the desirable join output. We note here that this process only transfers the small table S to all cluster nodes and thus avoids the costly data shuffling of the large table L, minimizing any time consuming data transfers. However, a possible drawback is that S has to be loaded several times, since each mapper runs as a separate process. This can be optimized by loading S only once per node, using a shared memory among mappers.

4.3.2 Map theta-join

Problem Statement: Similar to Section 4.3.1 consider the case where we have two tables L and S but now the goal is to perform a compound theta-join on a specific column, i.e., our goal is to compute $L \bowtie_{L.c \geq S.c_1 \wedge L.c \leq S.c_2} S$. The rational is that the files containing the IP to AS and IP to Country mappings consist of IP ranges that are part of an AS or country dataset and not a record for each IP address with its corresponding information. Thus, it is clear that we cannot perform an equi-join without blowing up the size of files from a few MB to some Gigabytes.

To perform the theta-join required, we must now use a different data structure rather than a hash table and so we choose to design our algorithm using an order-preserving data structure, such as a TreeMap. The methodology, in this case, consists of the following steps. We transfer table S to each map task and import it in a TreeMap main memory structure. To produce the join results, for each record of L we extract the join key and perform a range search against the TreeMap structure. We integrate this functionality in Hive and Shark by using custom made user-defined functions that are responsible for the aforementioned operations.

4.3.3 Map equi-join with large meta-dataset

Problem Statement: The definition of the problem is almost identical to this in Section 4.3.1 except for the fact that now S is quite large and cannot fit in a mapper's main memory as a whole. An example of this restriction is the IP to reverse DNS lookup mapping file which is around 57 Gigabytes in size. In this case, we must follow a slightly different approach to implement a map-join and avoid unnecessary data transfers. In particular, the key idea here is that each sFlow data file contains a limited number of unique IP addresses and thus, it is not required to transfer the entire mapping file into memory but only the portion that contains the information about these unique IPs (see semi-join in [Blanas 10]). Our approach is to pre-partition the sFlow data. Knowing beforehand the range of IPs in each file we can retrieve the respective records from HBase. Thus, utilizing the order-preserving storage of HBase we can perform range scans and transfer only the required IP to reverse DNS (IP-DNS) pairs into a mapper's memory.

Pre-partitioning can be performed using various approaches but we turn our attention towards two methods:

Method 1: Static partitioning. The first approach in partitioning is to use a uniform partition scheme across the join fields (IP addresses) of the *log* dataset. The meta-data required for each partition of the *log* (sFlow) dataset should fit in the main memory of a map task. In our IXP use case, we divide the IP-DNS *meta-dataset* in chunks of IP-DNS pairs that can fit in memory, and then apply the same uniform partitioning to the sFlow data. We must take into account that each sFlow file contains a source and a destination IP. Thus, the partitioning needs to be in two dimensions with each one corresponding to one of these IP types. This way of partitioning the dataset is quite straightforward. However, it fails to produce balanced output sFlow data files. This particular partitioning scheme does not consider the distribution of IP addresses in the sFlow files and hence, which IP pairs tend to exchange more traffic than others. Furthermore, the number of output files is quite large without being equally balanced in size (actually a lot of files end up empty) which results in poor performance during the partitioning phase. The implementation consists of two steps:

(i) Partitioning: A MapReduce job is responsible for performing the actual partitioning after defining the split points from the mapping file and produces sFlow files containing records with IP addresses in a given range as well as files containing the actual unique IPs in every chunk. This process is done off-line before any actual query is issued.

(ii) Query: The second step is to integrate the creation of the hash table containing the unique IP to reverse DNS lookup pairs and the equi-join implementation logic in a user-defined function which is aware of the underlying partitioning scheme.

Method 2: Dynamic partitioning. The second approach overcomes the restrictions and problems of the aforementioned method by using a dynamic data structure for partitioning the dataset, called K-dimensional Tree (K-d Tree) [Louis 75]. This data structure is well suited, among other alternatives (e.g., R-Tree), for space partitioning such as the one we are interested in for the following reasons: (i) it leaves no empty space when partitioning the data, (ii) it ensures that all output files will contain a balanced number of records, and (iii) it allows multiple dimensions in data partitioning. This last feature leads, naturally, in an efficient way of executing multi-dimensional filtering queries. Although the K-d Tree structure does not perform efficiently when the number of dimensions increases beyond a certain limit, it is suitable for our cause since the number of dimensions for the important analytics queries in the sFlow dataset does not exceed 10. For values close to 10 the performance of K-d Tree is slightly reduced but nevertheless it is still quite acceptable. There are three steps:

(i) Sampling: First, a relatively small (about 1%) sampling of the input data is performed to create the K-d Tree that holds the information of the split points used during the partitioning step. During the sampling step, the number of maximum records m that each file will contain after the partitioning has taken place, is also set. This value is chosen to allow the creation of

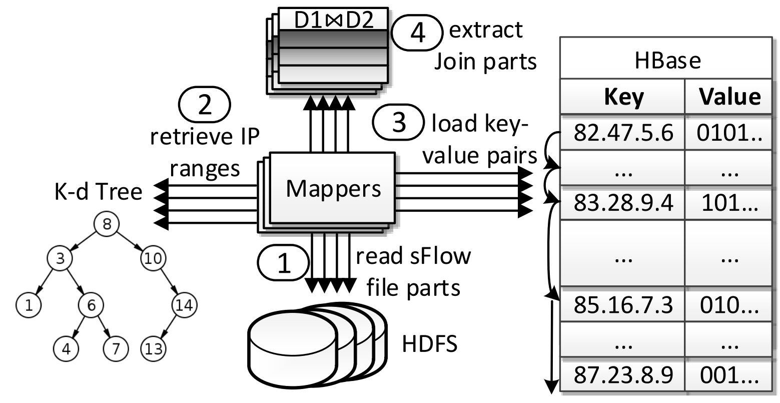


Figure 4.2: *Join Execution using K-d Tree.*

similar in length *log* table partitions while their size is close to the HDFS's predefined block size for well-balanced load distribution among mappers.

(ii) Partitioning: The second step is the use of the K-d Tree for partitioning the dataset according to the split points specified in the previous step. This process is executed in a separate MapReduce job that outputs the *log* table partition files that contain data within a hypercube of the partitioning space attributes and files containing the unique join values present inside each partition, in ascending order. The latter information is used to efficiently retrieve the respective meta-data values from the HBase indexed *meta-datasets*.

(iii) Query: The final step is to construct a user-defined function that integrates the above described functionality. The operations of this function are clearly illustrated in Figure 4.2. The user-defined function takes as input a join attribute (e.g., IP) and the K-d Tree structure. It uses the partition's join range as well as the file that contains the unique join values of the respective partition in order to transfer the relevant meta-data to a mapper's memory. Essentially, it follows a merge join technique between the unique join values and the HBase indexed *meta-dataset*. When the relevant meta-data are retrieved, an equi-join similar to that in Section 4.3.1 produces the join output (see Algorithm 17).

As mentioned before, the required IP-DNS pairs are transferred from HBase, a fact that renders network throughput a determining factor for the system's performance. In order to make the HBase merge-join operation more efficient, we introduce a new scan utility that enhances the performance when reading a range of IPs from HBase. Our scan utilizes a method called `seekTo()` that uses a heuristic to decide whether the next key-value pair will be accessed sequentially using the `next()` method or it will be accessed immediately by jumping directly to it (see Figure 4.2). Essentially, it is best to choose the latter when the number of intermediate key-value pairs is above a certain threshold defined by the cost of initializing a jump to the next pair compared to the cost of sequentially accessing all of the intermediate pairs [Papailiou 13].

Algorithm 17: Join execution using K-d Tree

```
1: Function $evaluate()$
2: if $DnsMap == NULL$ then
3:     $kd = readTreePartitionFile()$
4:     $kd.findBuckets(min, max, l)$
5:     $l.sort()$
6:     $partNum = l.indexOf(partNum)$
7:     $line = readLineFrom(uniqueIPFile)$
8:     $s = HBaseTable.getScanner(scan.setStartRow(line))$
9:     $result = s.next()$
10:    while $line! = NULL\&\&result! = NULL$ do
11:        if $UniqueIP > ScanIP$ then
12:            $result = s.seekTo(line, UniqueIP - ScanIP)$
13:        else
14:            $line = br.seekTo(result.getRowKey())$
15:        end if
16:        if $result.getRowKey().equals(line)$ then
17:            $keyValue.putInHashMap()$
18:        end if
19:    end while
20: end if
21: return $DnsMap.get(ip)$
```

The combination of the aforementioned techniques results in a significant speed up in the execution time of each query. Apart from that, it provides us with a straightforward way of implementing range queries, e.g., across a certain time window, by exploiting the properties of a K-d Tree to find the specific sFlow files that contain records whose target-values lie in the specified range. This way we can limit the number of files that need to be processed and avoid a time consuming scan of the entire dataset.

4.3.4 Dynamic Mapping Files

The mapping files of IPs to ASes, IP geolocation, and reverse DNS lookup are not static and they do change over time. These changes might not be that often, e.g., IP block assignments to ASes, or they may get updated daily, e.g., geolocation information. Datix supports the import of multiple time-varying mapping files. To accomplish this, we use two different storage approaches based on the size of *meta-datasets*. Small ones are stored in an HDFS directory containing all the different timestamp versions (each file's name is appended with the timestamp version). As mentioned before, large mapping files are stored in HBase which natively supports multiple key-values that have different timestamp versions. In each case, during query time the appropriate key-value pairs are loaded into memory according to the *log* partition's timestamp range.

4.4 Experiments

4.4.1 Cluster Configuration

The experimental setup consists of an ∼okeanos IaaS [Koukis 13] cluster of 15 VMs. The HDFS, MapReduce, HBase and Spark master is equipped with 4 virtual CPUs, 4GB of RAM and 10GB of disk space. There are also 14 slave nodes hosting all other required processes each of which has 4 virtual CPUs, 8GB of RAM and 60GB of disk space, summing up to a total of approximately 900GB of disk space. Each worker VM runs up to 4 map tasks and 4 reduce tasks, each consuming 768MB of RAM. We utilized Hadoop v1.2.1, HBase v0.94.5, Hive v0.12.0, Spark v0.9.1 and Shark v.0.9.1 respectively.

4.4.2 Dataset Specification

In our experiments we used a set of sFlow data coming from a national Internet Exchange Point. This dataset spans a period of about six months from 31/07/2013 until 17/02/2014 and is around 1TB in size. In addition, supplementary information of IP to AS and country mappings was retrieved from Geolite[6] . These two files are 12MB and 7MB in size respectively. Finally, IP to reverse DNS lookup mapping had a size of 57GB and was retrieved from the ZMap public research repository that archives Internet-wide scans[7].

4.4.3 System-level Comparison

We compare the performance of Datix when running the same queries using either Hive or Shark as the data analysis tools. We evaluate the performance of these two systems based on the query execution time and we comment on the performance advantages and disadvantages of each system.

Dataset Partitioning: Table 4.1 depicts the overhead of pre-partitioning the dataset for variable dataset sizes. Both sampling and partitioning have been incorporated into the resulting values. As we can see, data loading time scales linearly in respect to dataset size especially for smaller values. For larger datasets, there is an additional slight overhead, mainly due to the fact that more mapper and reducer processes have to be scheduled to complete the job. In addition, pre-partitioning takes roughly 4 times more compared to a simple scan of the respective dataset.

Dataset Size (%)	20%	40%	60%	80%	100%
Loading (min)	24	50	78	102	150

Table 4.1: *Data Loading vs Dataset Size*

[6]http://dev.maxmind.com/geoip/legacy/geolite/
[7]http://scans.io/

Query Type	Python	Hive	Datix-Hive		Datix-Shark	
			2D	3D	2D	3D
topAS	58min	170min	50min	52min	15min	16min
topDNS	>24h	135min	35min	76min	30min	64min
topDNS 1 week	>24h	116min	30min	9min	28min	8min

Table 4.2: *Base Join Implementation vs Datix*

Table 4.2 reports the execution times for two query types using a Python based centralized approach, the default join implementation of Hive and Datix. These queries compute the top-k AS or DNS pairs, i.e., we refer to Fully Qualified Domain Names (FQDN), exchanging network data for the entire time period of the dataset we possess. These queries, although simple, cover the *basic* SQL functionalities and can be used to form more complex or specific ones. In particular, these two queries use: join operators, *GROUP BY* operators, *COUNT* operators and *ORDER BY* operators. By combining some or all of the aforementioned operators, more complex queries can be formed to utilize in full scale Datix capabilities. For example, a useful query would be "Calculate the daily traffic of a specific web-server over time" that uses some of the primitives described above (i.e., filtering in time dimension and *GROUP BY/COUNT* operators to calculate daily traffic). In all tables and graphs, 3D partitioning(i.e., source, destination IP, timestamp) results are presented, unless it's explicitly stated otherwise.

For our Python based experiments, we used a host machine featuring a Core i7-4820K CPU with 8 threads, 48GB of RAM and 8TB disk. For the topAS query, where the *meta-dataset* is small enough and fits in a main memory dictionary, the Python based implementation is nearly 4 times slower than the Datix on top of Shark while it is almost the same for Datix using Hive. There are several reasons why the latter takes place. First, the machine we used for our Python experiments has a CPU clocked 2 times faster than that of the VMs, so, in this case, where all operations are in-memory, the difference in speed is obvious. Second, the performance of VMs running on top of an IaaS that might over-provision its resources cannot be as good as a physical machine. Furthermore, the mapping of virtual CPUs may not be 1 to 1 with the physical CPUs of each host. Lastly, the communication overhead between mappers and reducers when using MapReduce based applications is another thing to consider. On the other hand, the *meta-dataset* for the topDNS query is quite large and thus, it is not feasible to store it in main memory. A possible solution to this problem is to store the *meta-dataset* in a database (e.g., MySQL) and for each sFlow record issue a query to retrieve the required information. Due to high latency when accessing the database, this implementation is inefficient and the query requires more than one day to be executed even in the case of a week's amount of data. In contrast, Datix manages to execute those queries offering scalability and efficiency as discussed in detail in the following paragraphs.

As a second observation, we note that our distributed join algorithms outperform the ones in Hive by nearly 70% both for topAS and topDNS queries because we manage to efficiently exploit in-memory computations and avoid expensive disk I/O operations compared to the simple join algorithm used by Hive. In the case of Shark, the base join implementation fails during execution due to lack of memory, while our implementation is more robust and is executed successfully in short time as shown in Table 4.2. Another point of interest is the behavior of our system when using a 2D (i.e., source and destination IPs) or 3D (including timestamp) partitioning scheme. In the case of 3D partitioning, a considerable increase in the execution time occurs for the topDNS query. To explain this behavior, we note that when more dimensions are used, the sFlow files produced contain IPs in a wider range and therefore, more IP-reverse DNS lookup pairs have to be transferred from HBase. This fact is verified, since the execution time of the topAS query, where all operations are performed in-memory, is not affected. Despite this noticeable overhead, using 3D partitioning results in much better performance when it comes to filtering queries as shown in the third row of Table 4.2. To explain this, we note that adding another dimension (i.e., timestamp of sFlow records) results in much faster query execution since the processing is limited only to the sFlow files that contain the appropriate records.

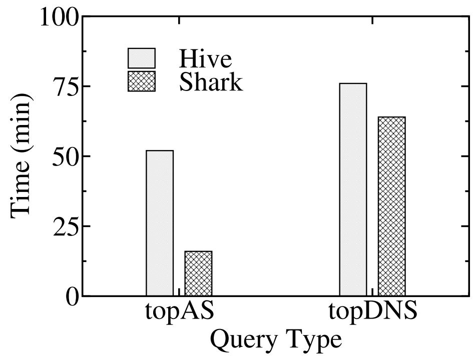


Figure 4.3: *Hive vs Shark*

In order to provide a direct comparison of the two different big data analysis tools, we then test the performance of Hive versus Shark. Figure 4.3 shows the execution time for both systems for two different queries. For a fair comparison we used the exact same HiveQL query for Hive and Shark. We observe that for the topDNS query the execution time is similar for both systems compared to the topAS query where Shark is significantly faster. This behavior is somewhat expected since the former query requires a large amount of data to be transferred from HBase, a process that is limited by the available network throughput and therefore is independent of the characteristics of each tool.

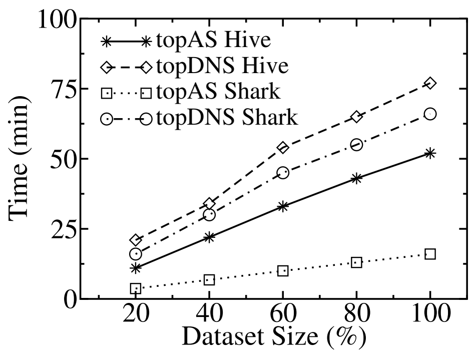


Figure 4.4: *Dataset Scalability*

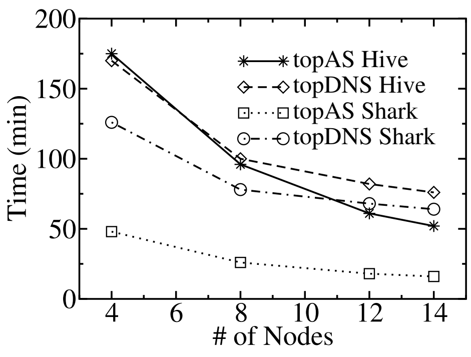


Figure 4.5: *Nodes Scalability*

Figures 4.4 and 4.5 show the scalability of our system in respect of dataset size and number of available nodes respectively. In Figure 4.4, we vary the volume of data processed while we keep the number of nodes at 14. In contrast, in Figure 4.5 we vary the number of nodes while processing the entire dataset. A first observation is that our system scales linearly with the dataset size regardless of the tool used. Shark is faster in all cases when compared to the corresponding implementation in Hive. This is mainly due to the following reasons. First, Shark avoids data spilling to disk for all the intermediate results in a multi-staged MapReduce job, thus, it avoids time consuming I/O operations. Second, it utilizes an efficient task scheduling algorithm and hence, overcomes the expensive launch procedure of mapper or reducer tasks that takes place in traditional MapReduce engines, like Hadoop. Moreover, we observe that for small number of nodes the system's scalability is close to linear while experiencing a slight degradation in performance as this number increases. Lastly, the difference in execution time when using Hive and Shark is quite significant for the topAS query regardless of the cluster or dataset size.

Figure 4.6 shows the speedup in execution time for both query types when using the ORC file format [Huai 14] instead of a plain text format on top of Hive. ORC file format uses a column based approach when storing a Hive table data and is able to retrieve only the required column data. Therefore, in queries that combine information only from a few columns we observe a significant speedup in total execution time.

As an overall comment, Datix's power lies in the cases where the *meta-dataset* is quite large and the available RAM is not sufficient to perform a simple map-join. In particular, our partitioning scheme is designed to overcome such limitations and enables us to efficiently answer various queries. However, when the *meta-dataset* is small enough, a Python-based approach

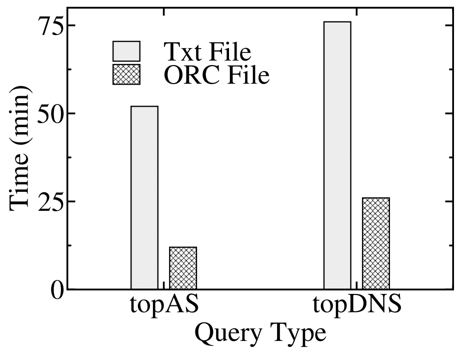


Figure 4.6: *Text vs ORC file format*

running on a single node with lots of resources (CPUs and RAM) is expected to yield comparable performance to Datix. Nevertheless, our system still overpowers such methods (even by a small percentage) and is by far better than the baseline join of Hive and Shark.

CHAPTER 5

Related Work

This chapter surveys the related work to this thesis. We organize it in two components: work related to RDF datastores (Section 5.1), containing references to both RDF engines and SPARQL result caching, and work related to Network data analytics (Section 5.2)

5.1 RDF Datastores

Since its introduction, RDF data management has been studied in a wide variety of contexts. Driven by diverse data sources and query patterns, a rich and constantly expanding collection of RDF engines has emerged [Luo 12, Kaoudi 15].

A popular approach for RDF engines is to leverage relational databases by introducing SQL schemas to describe RDF data. Statically defined schemas contain a single (S,P,O) table where all triples are stored in one relational table, property tables [Carroll 04], clustered property tables [Broekstra 03], optimized schemas for column-oriented databases [Abadi 09], etc. Recent research suggests the discovery of an SQL schema using data mining techniques on RDF data. Lattice structures are used in [Wang 10] to derive a schema evolution algorithm. Hashing and graph coloring are employed in [Bornea 13], while characteristic sets [Gubichev 14] are used in [Pham 15].

Apart from relational-based approaches, a wide variety of native indexing techniques have been proposed for RDF data. One of the most widely used ideas is Hexastore [Weiss 08] which sugests the materialization of all six *permutations* of subject-predicate-object indices. Following this technique, engines can retrieve any triple pattern at minimal cost and extensively use efficient merge joins. A similar approach is followed in RDF-3X [Neumann 10a] which maintains a total of 15 RDF indices. BitMat [Atre 08] is an alternative approach for storing RDF triples via one- and 2-dimensional bit matrices and using efficient matrix operations for query processing.

Viewing RDF data from a graph perspective, a wide variety of indexing and query processing techniques have been proposed. The GRIN index [Udrea 07] can be used to reduce the query search space leveraging information about central RDF vertices. To achieve data locality, the TripleT index groups RDF nodes according to their distance [Fletcher 09]. In [Tran 10, Hawash 10], a RDF graph summary is extracted and used for data partitioning and for early pruning of query results. The RG-index [Kim 14] can also be used to filter out RDF triples when processing SPARQL queries. Recently, state-of-the-art subgraph isomorphism algorithms were adapted to handle SPARQL processing [Kim 15].

To tackle the "Big-Data" challenge, research has also moved onward to distributed RDF data management systems. H_2RDF+ [Papailiou 13] stores RDF data on HBase tables while adaptively processing queries using centralized or MapReduced-based distributed execution. Huang et al. [Huang 11] propose graph partitioning techniques to distribute the RDF graph into a cluster of nodes, each running a local RDF-3X instance. Replicating triples that reside within *n hops* from each partition allows for unobstructed parallel processing of queries satisfying the *n hop* guarantee. Trinity-RDF [Zeng 13] targets distributed, main-memory indexing for RDF data. Utilizing graph exploration techniques it avoids join execution overheads and takes advantage of the quick random access provided by the memory resident data. TriAD [Gurajada 14] depends on distributed main-memory indices, partitioning RDF data using both hash and graph based partitioning techniques. TriAD's asynchronous MPI-based execution takes into account location information to avoid data transfers and synchronization during join execution.

In this section, we also present in more detail some of the most relevant RDF indexing and querying systems, distinguishing them in two categories: centralized and distributed systems.

5.1.1 Centralized Systems

Hexastore [Weiss 08] is a centralized solution that materializes six different indexes, one for each possible *permutation* of subject-predicate-object values; these permutations are *spo, pso, pos, ops, osp* and *sop*. The *spo* index, for instance, contains a list of predicates for each subject, while each predicate *p* in the list points to a table that contains all objects associated with *s* by *p*. These indexes allow the retrieval of any simple triple pattern at minimal cost.

A similar approach is followed in RDF-3X [Neumann 10a] along with query optimization strategies. RDF-3X employs six *lexicographic* indexes (similar to [Weiss 08]) as well as additional indexes that collect statistical information for pairs and stand-alone entities, amounting to a total of 15 indices. It extensively uses merge joins in order to achieve good performance. However, query execution highly depends on the amount of main memory required to perform joins, presenting problems with joins with small selectivity and large input. The use of single threaded query execution limits RDF-3X's scalability on modern multi-core server architectures. RDF-3X is regarded as a state-of-the-art solution in centralized RDF data stores.

In BitMat [Atre 08], RDF triples are indexed via a 3-dimensional $\langle s, p, o \rangle$ bit matrix. Each matrix element is a bit denoting the presence or absence of the corresponding triple. This matrix is flattened to 2-d matrices creating multiple indexes for all possible combinations of subject-predicate-object. However, this approach is effective only in a main-memory environment.

Other frequently-used, efficient centralized systems include Virtuoso [Erling 09], Jena [Carroll 04] and OWLIM [Kiryakov 05]. Still, all aforementioned approaches run on a single machine, limiting their storage and processing capacity.

5.1.2 Distributed Systems

In order to tackle the big-data challenge, research has recently moved onward to distributed RDF data management systems. A first attempt in this direction, 4store [Harris 09], distributes a single *pos* index over the nodes of a cluster, and employs distributed join algorithms to execute SRARQL queries. However, apart from the deficiency ensuing from having a single index, 4store does not adapt its performance for multiple join queries of various selectivity.

HadoopRDF [Husain 11] uses Hadoop Distributed File System (HDFS) files named after predicate values to partition the input RDF data, thereby creating a *pos* index. It is not a fully functional index though, as it can only retrieve subject-object combinations for a given predicate, but not, for instance, subjects for a given predicate-object combination. HadoopRDF performs SPARQL joins in the MapReduce framework, employing an algorithm that greedily reduces the total number of remaining MapReduce joins at each step. Remarkably, this greedy planner does not take into consideration the join selectivity. Finally, joins are executed only with MapReduce jobs, inducing large overheads for selective queries.

Efforts have also been made towards optimizing distributed joins using MapReduce [Blanas 10]. In this work, the authors compare different algorithms for joining big log tables, stored in raw HDFS files. The main difference with H_2RDF+ is that we index our data using HBase. This means that we always process only the amount of data required for each join without having to process the whole dataset. The join algorithms presented in [Blanas 10] do not take into account any data preprocessing and indexing. We also use a multi-way join scheme that differs from the two-way joins implemented in [Blanas 10].

H_2RDF [Papailiou 12] uses a three-index scheme and depends on the *Partial Input Hash-join*. This algorithm exploits HBase indexing and checks whether the join contains small input patterns. If this is the case, only those are read from the indexes during the map phase. The remaining patterns are joined using `get` operations on the reduce phase of the join. H_2RDF also uses adaptive centralized and distributed execution. The main differences with H_2RDF+, can be found in the join algorithms, the number of maintained indexes (three versus six), the more detailed statistics and the type and size of IDs

An alternative proposal is presented by Huang et al. [Huang 11]; this method starts out by partitioning the RDF graph into distinct subgraphs, each stored in a single node running a local RDF-3X instance. Moreover, in a replication scheme, each node keeps information on the graph contents within n *hops* from the contents it owns; this provision allows for unobstructed parallel processing of SPARQL queries satisfying an n *hop guarantee*. In case this guarantee is not satisfied, Hadoop is invoked for distributed join processing. The proposed system suffers from the following drawbacks:(1) Slow import: apart from the centralized graph processing, it also needs a large amount of time to load the corresponding data to individual RDF-3X instances.(2) Its MapReduce joins implement a non optimized, 2-way hash-join scheme.(3) The n-hop guarantee requires size of replication data *exponential* to n.

Zeng et al. [Zeng 13] introduce Trinity.RDF, a distributed, in-memory system. They propose a query execution model based on graph exploration that can be viewed as a sequence of semi-joins similar to the approach followed in BitMat. The main drawback of this system is that its performance is bound by the main memory capacity of the cluster, as the whole set of triples needs to be loaded in main memory. This is not a scalable approach, especially given that clusters are comprised by commodity nodes. Moreover, local semi-join results are gathered at a central node responsible for producing the final results. This server can be the bottleneck of the query execution when: 1) Handling query graphs that contain cycles. The semi-join query execution engine employed cannot fully reduce the result size for these graphs [Bernstein 81], thus overloading the last step of the execution. 2) The query output is really large. In this case the last server will need to generate and write the whole output. This process is limited by the sequential iteration over the result set and the large write I/O requirements.

5.1.3 Workload-adaptive RDF engines

Workload-adaptivity is also emerging as a prevalent technique to handle the diversity of RDF data and SPARQL queries. In [MahmoudiNasab 10], the workload queries are monitored to discover frequent co-existing predicate groups. An automated adjustment phase is responsible to materialize n-ary relational tables based on the discovered groups. This approach resembles schema discovery based on characteristic sets [Pham 15] and is limited to flat grouping of predicate values without any regard to the query graph structure.

Partout [Galarraga 14] introduces a distributed RDF engine which, upon examining a query workload, makes decisions on both triple partitioning and replication to reduce network traffic and increase parallelization. Similarly, WARP [Hose 13] extends basic graph partitioning with query pattern partitioning, discovered from a set of predefined workload queries. However, both WARP and Partout assume that the representative workload queries are given upfront. Expensive re-partitioning of the entire data is applied before the actual query execution. Additionally, both systems use location information to generate query plans. In the case of WARP, a heuristics-based query containment algorithm is used to find which parts of a query can be executed in parallel. For Partout, flat triple pattern location is used to avoid communication.

In contrast, AdPart [Harbi 16], initially applies subject-based hash partitioning of RDF triples. It constantly monitors the query workload making dynamic decisions on redistribution and replication of triples. Furthermore, triples transferred during query execution are also replicated. As a result, consecutive queries can be executed in parallel without data communication. However, AdPart only applies flat triple replication without maintaining materialized query results. This approach can boost parallel execution and avoid communication but offers no speedup to the actual join execution. To showcase this, let us consider the case of the exact same query being executed twice. While our system will just point to the first computed result, AdPart will need to perform the same join operations on top of the redistributed data. Furthermore, AdPart can take advantage of pattern redistribution only when a consecutive query is a complete subgraph of a redistributed pattern. Therefore, queries that are partially redistributed will still be executed over the primitive hash-based partitioning scheme, offering no speedups.

Lastly, Chameleon-db [Aluc 15] uses graph partitioning to store RDF data and subgraph matching for query processing. It also monitors the query workload, periodically adjusting its storage partitions and creating partitions based on query patterns. Query rewriting techniques are used to process queries using the most appropriate layouts. However, this technique mainly targets the adaptive partitioning of the RDF graph, while our system targets result re-usability and materialized view caching.

5.1.4 SPARQL Result Caching

Relative to relational data management, a lot of research has been conducted in the fields of automatic view selection [Yu 03, Aouiche 09], query rewriting using views [Levy 95], multi-query optimization [Roy 00, Diwan 06] and result caching [Shu 13, Fard 14]. To be effectively used, such techniques restrict RDF engines to the relational perspective, discarding the potential benefits of native or graph indexing. In contrast, we present algorithms that utilize the SPARQL interface and can thus be shared by all RDF engines.

A first attempt to introduce SPARQL caching was made in [Martin 10], where a meta-data relational database is responsible for storing information about cached query results. However, this approach cannot tackle the isomorphism problem introduced when the same SPARQL graph pattern is requested from different queries with small deviations such as pattern reordering, variable renaming, etc. A more sophisticated approach was presented in [Yang 11], where the cache keys consisted of normalized Algebra Expression Trees (AETs) that correspond to cached join plans. A cached AET is only used in join plans that exactly contain it as a subtree limiting the usability of cached results. In [Lorey 13], the authors introduce a similarity-based matching algorithm that can detect cached queries that resemble the query in hand. Yet, this greedy technique cannot find all candidate subgraph matches for a SPARQL query. They also propose some heuristics to augment the workload queries and prefetch SPARQL query results, which resembles our *profitable* query execution. Their approach is based on techniques that cannot examine the benefit of all possibly usable results as well as their indexing.

In [Shu 13], query containment and "evaluability" algorithms are described in order to check if a consequent SPARQL query can be entirely evaluated using the computed results of a previous one. In addition, [Fard 14] proposes a tight simulation algorithm that performs a more generic SPARQL query containment. This algorithm is, again, used to check for the usability of computed results. Yet neither system tackles the case of a result matching part of the new query graph and the case of using multiple cached results to answer separate parts of a new query.

Apart from SPARQL result caching, there exist alternative techniques that attempt to tackle parts of the problem that we address in this paper. Many of the state-of-the-art approaches in graph database indexing propose the use of frequent pattern indexing [Zhao 07, Yan 04]. Frequent and discriminative subgraphs are discovered and indexed during the import phase of the dataset and can then be utilized to efficiently answer queries. In addition, approaches for multi-query optimization have also been proposed for SPARQL query processing [Le 12]. Such techniques depend either on knowledge of the workload or on discovering frequent patterns in the dataset. In contrast, our approach assumes no a-priori knowledge on either the dataset or the workload.

5.2 Network Analytics

In this section we present related research on network data analytics as well as on distributed join algorithms, discussing their comparison to the presented Datix system. There is a number of systems that have been proposed for network traffic analysis and each one tackles a particular aspect of this broad research area. In [Lee 13], an approach was presented to analyze network traffic by processing libpcap files in a distributed environment provided by Hadoop's MapReduce in combination with Hive as a data warehousing tool. The authors implemented an intelligent NetFlow reader in Hadoop that was aware of the particular format of libpcap files which could be spread across different nodes in the cluster. Our work is rather orthogonal as our join algorithms can be integrated in their system and by using the same *meta-datasets* can extract additional information about network traffic. Another approach is proposed in [Li 13], where the authors introduce a machine learning paradigm to classify host roles based on network traffic analysis through collecting sFlow records. Essentially, this work tries to extract information like our approach by analyzing sFlow packets and utilizing MapReduce as the execution framework and NoSQL databases as storage. The difference of our proposed approach is that by enabling the use of arbitrary *meta-datasets* we can extract much richer information about network routing, dimensioning and security features rather than only classifying host roles. A system that can perform both streaming and batch processing of network traffic in order to analyze the constantly increasing volume of network traffic data is presented in [Bumgardner 14]. It addresses the scalability problem of existing systems by using distributed methodologies such as MapReduce but it does not support the use of high-level languages over Hadoop framework and thus, cannot be executed over Spark without considerable effort.

The work presented in [Herodotou 11] deals with hierarchical partitioning, optimizing the query plan to prune processing only to required data partitions. For each data dimension there is a different level in the partitioning tree structure. It involves changing the optimizer module by adding extra features to decide on the chain of joins performed. The main focus of the authors is on traditional RDBMSs while they claim that their approach can be extended to parallel databases. Our approach differs in the fact that we use a flat partitioning scheme dictated by the K-d Tree structure and our system design is tailored to using NoSQL storage and distributed system techniques. Furthermore, our partitioning scheme focuses on splitting the volume of data appropriately to fit in a mapper's memory to perform efficient map-side joins.

In [Johnson 15], the authors present TidalRace which builds on data streaming applications and show how to optimize partition-based operations. Datix focuses on log processing (batch operations) and overall optimization of query execution in various cases. In contrast, TidalRace supports incremental updates to partitioning information, partition re-organization, and partition-wise optimizations.

DBStream [Bar 14] is a Data Stream Warehouse solution for Network Traffic Monitoring and Analysis applications. The queries we execute could also be deployed on this system by extending the functionality of DBStream to support the import of various *meta-datasets* like Datix and then evaluate the resulting performance. DBStream supports real-time data analysis and incremental queries apart from batch processing jobs. However, it is deployed in a centralized manner over a traditional RDBMS (PostgreSQL) while our system is fully decentralized, designed to be able to scale to a large number of nodes and gain extra performance when more resources become available. TicketDB [Baer 11], which is the predecessor of DBStream, was compared to vanilla MapReduce jobs performing reduce-side joins, but it does not use a partitioning scheme similar to our K-d Tree approach to enable efficient execution of map-side joins.

5.2.1 Distributed Join Algorithms

Distributed join over MapReduce-like systems is challenging and therefore different approaches have been proposed to address this issue. A first attempt to introduce join algorithms for log processing was presented in [Blanas 10], where the authors compare different algorithms, depending on whether the *meta-dataset* can fit in memory, that can be used to implement equi-joins in MapReduce. However, this work does not tackle the problem of theta-joins which consist of more general join conditions. In fact, our approach modifies the Broadcast Join algorithm presented in this work to effectively deal with theta-joins. In [Yang 07], the authors introduce techniques that support other joins and different implementations, but it is also required to extend the MapReduce model. Furthermore, users have to implement non-trivial functions that can handle the dataflow in the distributed system. Hence, this work cannot support high-level languages that run on top of MapReduce (i.e., Hive) compared to our approach. In [Okcan 11,Afrati 10], the authors propose algorithms that perform partitioning during query time to speed up execution time, whereas our approach focuses on pre-partitioning data in order to efficiently use map-phase joins.

CHAPTER 6

Conclusions

In this thesis, we presented H_2RDF+, a fully distributed RDF store capable of storing and querying arbitrarily large amounts of triples. The main contribution lies in our scalable distributed Merge and Sort-Merge join execution and our adaptive decisions about centralized and distributed join execution. We have also optimized both the compression and retrieval capabilities of our HBase indexes. H2RDF+ greatly. H_2RDF+ is able to achieve great speedups and linear scaling in query processing and data loading tasks as well as high-throughput concurrent operations. These features allows H_2RDF+ to handle non-selective queries in a dataset of size 2.5TB using a 35 small-sized worker node cluster.

We have also proposed a SPARQL query caching framework that is able to effectively cache and utilize query results. We introduced a novel SPARQL query simplification technique that can be used to reduce the complexity of labelling and optimizing complex SPARQL queries. Our simplification technique is integrated in both the canonical query labelling algorithm and the cache architecture that is used to effectively store and retrieve cached results. Furthermore, we extend the *DPccp* dynamic programming optimizer by adding support for multi-way join plan exploration and generation of optimal query plans that consider the utilization of cached query subgraphs. *Profitable* SPARQL queries are discovered and pro-actively cached, in order to reduce the response times for several workloads. Our caching framework was integrated on top of a state-of-the-art distributed RDF datastore, reducing its average response time by up to two orders of magnitude and offering interactive response times for complex workloads and huge RDF datasets.

Another contribution of this thesis is a novel network analytics system that depends on distributed processing techniques and is able to effectively execute filtering queries over state-of-the-art distributed processing engines. We introduced a smart pre-partitioning scheme to speed up the execution time of filtering queries (i.e., over a particular time period or set of IP addresses) and we integrated this functionality into an SQL compliant system by using custom-made user-defined functions that are aware of the data format and implement a custom variation of map-join algorithm. Our approach reduced query execution time compared to the basic Hive and Shark implementation by nearly 70%, while efficiently answering queries that took over a day to be processed with the existing Python-based code. In this work, we used sFlows as a log dataset from which various information was recovered.

Abbreviations

AS	Autonomous System
BGP	Basic Graph Pattern
DNS	Domain Name System
HDFS	Hadoop Distributed File System
HFile	HBase file format
ISP	Internet Service Provider
IXP	Internet Exchange Points
NoSQL	Not only SQL
RAID	Redundant Array of Inexpensive Disks
RDF	Resource Description Framework
RDFS	Resource Description Framework Schema
SPARQL	SPARQL Protocol and RDF Query Language
SQL	Structured Query Language
TPC	Transaction Processing Performance Council
URI	Unique Resource Identifier

VM	Virtual Machine

Publications

Journals

- D. Sarlis, **N. Papailiou**, I. Konstantinou, G. Smaragdakis and N. Koziris: “Datix: A System for Scalable Network Analytics.” ACM SIGCOMM Computer Communication Review, 45(5), October 2015.

- T. Risse, E. Demidova, S. Dietze, W. Peters, **N. Papailiou**, K. Doka, Y. Stavrakas, V. Plachouras, P. Senellart, F. Carpentier, A. Mantrach, B. Cautis, P. Siehndel and D. Spiliotopoulos: “The ARCOMEM Architecture for Social- and Semantic-Driven Web Archiving” Future Internet Journal 2014, 6, 688-716.

- E. Demidova, N. Barbieri, S. Dietze, A. Funk, H. Holzmann, D. Maynard, **N. Papailiou** W. Peters, T. Risse and D. Spiliotopoulos: “Analysing and Enriching Focused Semantic Web Archives for Parliament Applications” Future Internet Journal 2014, 6, 433-456.

Conferences

- **N. Papailiou**, D. Tsoumakos, P.Karras and N. Koziris: “Graph-Aware, Workload-Adaptive SPARQL Query Caching” In Proceedings of the 2015 ACM SIGMOD/PODS International Conference on Management of Data (SIGMOD 2015), Melbourne, Australia

- K. Doka, **N. Papailiou**, D. Tsoumakos, C. Mantas and N. Koziris: “IReS: Intelligent, Multi-Engine Resource Scheduler for Big Data Analytics Workflows.” In Proceedings of the 2015 ACM SIGMOD/PODS International Conference on Management of Data (SIGMOD 2015 Demo Track), Melbourne, Australia

- I. Giannakopoulos, D. Tsoumakos, **N. Papailiou** and N. Koziris: "PANIC: Modeling Application Performance over Virtualized Resources" In Proceedings of the 2015 IEEE International Conference on Cloud Engineering (IC2E 2015), 9-13 March, Tempe, AZ, USA

- I. Giannakopoulos, **N. Papailiou**, C. Mantas, I. Konstantinou, D. Tsoumakos and N. Koziris: "CELAR: Automated application elasticity platform" In proceedings of the 2014 IEEE International Conference on Big Data (BigData 2014), Washington DC, USA

- **N. Papailiou**, D. Tsoumakos, I. Konstantinou, P. Karras and N. Koziris: "Scalable Indexing and Adaptive Querying of RDF Data in the cloud" In proceedings of the 6th International Workshop on Semantic Web Information Management (SWIM 2014), Snowbird, Utah, USA

- **N. Papailiou**, D. Tsoumakos, I. Konstantinou, P.Karras and N. Koziris: "H2RDF+: An Efficient Data Management System for Big RDF Graphs." In Proceedings of the 2014 ACM SIGMOD/PODS International Conference on Management of Data (SIGMOD 2014 Demo Track), Snowbird, Utah, USA

- **N. Papailiou**, I. Konstantinou, D. Tsoumakos, P.Karras and N. Koziris: "H2RDF+: High-performance Distributed Joins over Large-scale RDF Graphs." In proceedings of the 2013 IEEE International Conference on Big Data (BigData 2013), Santa Clara, CA, USA

- E. Demidova, N. Barbieri, S. Dietze, A. Funk, G. Gossen, D. Maynard, **N. Papailiou**, V. Plachouras, W. Peters, T. Risse, Y. Stavrakas and N. Tahmasebi: "Analysing Entities, Topics and Events in Community Memories." In proceedings of the 1st International Workshop on Archiving Community Memories, Lisbon, Portugal

- E. Angelou, **N. Papailiou**, I. Konstantinou, D. Tsoumakos and N. Koziris: "Automatic Scaling of Selective SPARQL Joins Using the TIRAMOLA System" In proceedings of the 4th International Workshop on Semantic Web Information Management (SWIM 2012), Scottsdale, Arizona, USA

- **N. Papailiou**, I. Konstantinou, D. Tsoumakos and N. Koziris: "H2RDF: Adaptive Query Processing on RDF Data in the Cloud" In Proceedings of the 21th International Conference on World Wide Web (WWW 2012 demo track), Lyon, France

Bibliography

[Abadi 09]	Daniel J. Abadi, Adam Marcus, Samuel Madden & Kate Hollenbach. *SW-Store: a Vertically Partitioned DBMS for Semantic Web Data Management.* VLDBJ, vol. 18, no. 2, 2009.
[Afrati 10]	Foto N Afrati & Jeffrey D Ullman. *Optimizing Joins in a Map-Reduce Environment.* In EDBT, 2010.
[Aluç 14]	Güneş Aluç, Olaf Hartig, M Tamer Özsu & Khuzaima Daudjee. *Diversified stress testing of RDF data management systems.* In The Semantic Web–ISWC 2014, pages 197–212. Springer, 2014.
[Aluc 15]	Gunes Aluc, M Tamer Ozsu, Khuzaima Daudjee & Olaf Hartig. *Executing Queries over Schemaless RDF Databases.* In Data Engineering (ICDE), 2015 IEEE 31st International Conference on, 2015.
[Aouiche 09]	Kamel Aouiche & Jérôme Darmont. *Data Mining-based Materialized View and Index Selection in Data Warehouse.* Journal of Intelligent Information Systems, 2009.
[Arvind 00]	Vikraman Arvind & Johannes Köbler. *Graph isomorphism is low for zpp (np) and other lowness results.* In STACS. Springer, 2000.
[Atre 08]	M. Atre, J. Srinivasan & J. Hendler. *BitMat: A Main-memory Bit Matrix of RDF Triples for Conjunctive Triple Pattern Queries.* In ISWC, 2008.

[Baer 11] A. Baer, A. Barbuzzi, P. Michiardi & F. Ricciato. *Two Parallel Approaches to Network Data Analysis.* In LADIS, 2011.

[Bar 14] A. Bar, P. Casas, L. Golab & A. Finamore. *DBStream: an Online Aggregation, Filtering and Processing System for Network Traffic Monitoring.* In IWCMC, 2014.

[Bernstein 81] Philip A Bernstein & Dah-Ming W Chiu. *Using Semi-joins to Solve Relational Queries.* JACM, 1981.

[Blanas 10] Spyros Blanas, Jignesh M. Patel, Vuk Ercegovac, Jun Rao, Eugene J. Shekita & Yuanyuan Tian. *A Comparison of Join Algorithms for Log Processing in MaPreduce.* In Proceedings of the ACM SIGMOD International Conference on Management of Data, 2010.

[Bonstrom 03] Valerie Bonstrom, Annika Hinze & Heinz Schweppe. *Storing RDF as a graph.* In Web Congress, 2003.

[Bornea 13] Mihaela A Bornea, Julian Dolby, Anastasios Kementsietsidis, Kavitha Srinivas, Patrick Dantressangle, Octavian Udrea & Bishwaranjan Bhattacharjee. *Building an efficient RDF store over a relational database.* In ACM SIGMOD, pages 121–132. ACM, 2013.

[Brickley 14] Dan Brickley & R.V. Guha. *RDF Schema 1.1.* W3C recommendation, 2014. http://www.w3.org/TR/rdf-schema/.

[Broekstra 03] Jeen Broekstra, Arjohn Kampman & Frank Van Harmelen. *Sesame: An architecture for storing and querying RDF data and schema information.* Spinning the Semantic Web: Bringing the World Wide Web to Its Full Potential, 2003.

[Bumgardner 14] Vernon KC Bumgardner & Victor W Marek. *Scalable Hybrid Stream and Hadoop Network Analysis System.* In ACM SPEC, 2014.

[Carroll 04] Jeremy J. Carroll, Ian Dickinson, Chris Dollin, Dave Reynolds, Andy Seaborne & Kevin Wilkinson. *Jena: Implementing the Semantic Web Recommendations.* In WWW, 2004.

[Chang 08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes & Robert E Gruber. *Bigtable: A Distributed Storage System for Structured Data.* ACM TOCS 26(2), 2008.

[Chatzis 13a] N. Chatzis, G. Smaragdakis, A. Feldmann & W. Willinger. *There is More to IXPs than Meets the Eye.* CCR 45(5), 2013.

[Chatzis 13b] Nikolaos Chatzis, Georgios Smaragdakis, Jan Böttger, Thomas Krenc & Anja Feldmann. *On the Benefits of Using a Large IXP as an Internet Vantage Point.* In ACM IMC, 2013.

[Chen 97] WC Chen, HI Lu & YN Yeh. *Operations of interlaced trees and graceful trees.* Southeast Asian Bull. Math, vol. 21, no. 4, pages 337–348, 1997.

[Cisco 13] Cisco. *Cisco Visual Networking Index: Forecast and Methodology, 2013 – 2018.* Available at http://www.cisco.com, 2013.

[Codd 70] Edgar F Codd. *A relational model of data for large shared data banks.* Communications of the ACM, vol. 13, no. 6, pages 377–387, 1970.

[Darga 08] Paul T Darga, Karem A Sakallah & Igor L Markov. *Faster symmetry discovery using sparsity of symmetries.* In Proceedings of the 45th annual Design Automation Conference, pages 149–154. ACM, 2008.

[Dean 08] J. Dean & S. Ghemawat. *MapReduce: Simplified Data Processing on Large Clusters.* Comm. of the ACM 51(1), 2008.

[Diwan 06] AA Diwan, S Sudarshan & Dilys Thomas. *Scheduling and caching in multi-query optimization.* In Proceedings of 13th International Conference Management of Data, volume 60, 2006.

[Durumeric 13] Z. Durumeric, E. Wustrow & J. A. Halderman. *ZMap: Fast Internet-Wide Scanning and its Security Applications.* In USENIX Security Symposium, 2013.

[Erling 09] Orri Erling & Ivan Mikhailov. *Virtuoso: RDF Support in a Native RDBMS.* In Semantic Web Information Management. 2009.

[Fard 14] Arash Fard, Satya Manda, Lakshmish Ramaswamy & John A Miller. *Effective caching techniques for accelerating pattern matching queries.* In Big Data 2014, pages 491–499. IEEE, 2014.

[Fletcher 09] George HL Fletcher & Peter W Beck. *Scalable indexing of RDF graphs for efficient join processing.* In Proceedings of the 18th ACM conference on Information and knowledge management, pages 1513–1516. ACM, 2009.

[Galarraga 14] Luis Galarraga, Katja Hose & Ralf Schenkel. *Partout: A distributed engine for efficient rdf processing.* In World wide web 2014, 2014.

[Gallego 11] Mario Arias Gallego, Javier D Fernández, Miguel A Martínez-Prieto & Pablo de la Fuente. *An empirical study of real-world SPARQL queries.* In 1st International Workshop on Usage Analysis and the Web of Data (USEWOD), Hydebarabad, India, 2011.

[Gassner 93] Peter Gassner, Guy M. Lohman, K. Bernhard Schiefer & Yun Wang. *Query optimization in the IBM DB2 family.* IEEE Data Eng. Bull., vol. 16, no. 4, 1993.

[Gopalkrishnan 99] V. Gopalkrishnan, Q. Li & K. Karlapalem. *Star/Snow-Flake Schema Driven Object-Relational Data Warehouse Design and Query Processing Strategies.* In DaWaK. 1999.

[Gubichev 14] Andrey Gubichev & Thomas Neumann. *Exploiting the query structure for efficient join ordering in SPARQL queries.* In EDBT, pages 439–450, 2014.

[Guo 05] Yuanbo Guo, Zhengxiang Pan & Jeff Heflin. *LUBM: A Benchmark for OWL Knowledge Base Systems.* Web Semantics: Science, Services and Agents on the World Wide Web, vol. 3, no. 2, 2005.

[Gurajada 14] Sairam Gurajada, Stephan Seufert, Iris Miliaraki & 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, pages 289–300. ACM, 2014.

[Harbi 16] Razen Harbi, Ibrahim Abdelaziz, Panos Kalnis, Nikos Mamoulis, Yasser Ebrahim & Majed Sahli. *Accelerating SPARQL queries by exploiting hash-based locality and adaptive partitioning.* The VLDB Journal, pages 1–26, 2016.

[Harris 09] S. Harris, N. Lamb & N. Shadbolt. *4store: The Design and Implementation of a Clustered RDF Store.* SSWS, 2009.

[Hartke 09] Stephen G Hartke & AJ Radcliffe. *Mckay's canonical graph labeling algorithm.* Communicating mathematics, vol. 479, 2009.

[Hawash 10] Ala' Hawash, Anton Deik, Bilal Farraj & Mustafa Jarrar. *Towards Query Optimization for the Data Web: Disk-based Algorithms: Trace Equivalence and Bisimilarity.* In ISWSA. ACM, 2010.

[Herodotou 11] Herodotos Herodotou, Nedyalko Borisov & Shivnath Babu. *Query Optimization Techniques for Partitioned Tables.* In ACM SIGMOD, 2011.

[Hoffart 11] Johannes Hoffart, Fabian M. Suchanek, Klaus Berberich, Edwin Lewis-Kelham, Gerard de Melo & Gerhard Weikum. *YAGO2: Exploring and Querying World Knowledge in Time, Space, Context, and Many Languages.* In WWW, 2011.

[Hose 13] Katja Hose & Ralf Schenkel. *WARP: Workload-aware replication and partitioning for RDF.* In Data Engineering Workshops (ICDEW), 2013 IEEE 29th International Conference on, pages 1–6. IEEE, 2013.

[Huai 14] Y. Huai, A. Chauhan, A. Gates, G. Hagleitner, E. Hanson, O. O'Malley, J. Pandey, Y. Yuan, R. Lee & X. Zhang. *Major Technical Advancements in Apache Hive.* In ACM SIGMOD, 2014.

[Huang 11] Jiewen Huang, Daniel J. Abadi & Kun Ren. *Scalable SPARQL Querying of Large RDF Graphs.* PVLDB, vol. 4, no. 11, 2011.

[Husain 11] M. Husain, J. McGlothlin, M.M. Masud, L. Khan & B.M. Thuraisingham. *Heuristics-Based Query Processing for Large RDF Graphs Using Cloud Computing.* TKDE, vol. 23, no. 9, 2011.

[Idreos 11] Stratos Idreos, Ioannis Alagiannis, Ryan Johnson & Anastasia Ailamaki. *Here are my Data Files. Here are my Queries. Where are my Results?* In CIDR, 2011.

[Johnson 15] Theodore Johnson & Vladislav Shkapenyuk. *Data Stream Warehousing in Tidalrace.* In CIDR, 2015.

[Junttila 07] Tommi Junttila & Petteri Kaski. *Engineering an efficient canonical labeling tool for large and sparse graphs.* In David Applegate, Gerth Stølting Brodal, Daniel Panario & Robert Sedgewick, editeurs, Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments and the Fourth Workshop on Analytic Algorithms and Combinatorics, pages 135–149. SIAM, 2007.

[Kaoudi 15] Zoi Kaoudi & Ioana Manolescu. *RDF in the Clouds: A Survey.* The VLDB Journal, vol. 24, no. 1, pages 67–91, 2015.

[Kim 14] Kisung Kim, Bongki Moon & Hyoung-Joo Kim. *RG-index: An RDF graph index for efficient SPARQL query processing.* Expert Systems with Applications, vol. 41, no. 10, pages 4596–4607, 2014.

[Kim 15] Jinha Kim, Hyungyu Shin, Wook-Shin Han, Sungpack Hong & Hassan Chafi. *Taming subgraph isomorphism for RDF query processing.* Proceedings of the VLDB Endowment, vol. 8, no. 11, pages 1238–1249, 2015.

[Kiryakov 05] Atanas Kiryakov, Damyan Ognyanov & Dimitar Manov. *OWLIM - A Pragmatic Semantic Repository for OWL.* In WISE, 2005.

[Köbler 94] Johannes Köbler, Uwe Schöning & Jacobo Torán. The graph isomorphism problem: its structural complexity. 1994.

[Kotidis 99] Yannis Kotidis & Nick Roussopoulos. *DynaMat: a dynamic view management system for data warehouses.* In ACM SIGMOD Record, 1999.

[Koukis 13] V. Koukis, C. Venetsanopoulos & N. Koziris. *∼okeanos: Building a Cloud, Cluster by Cluster.* IEEE Internet Computing 17(3), 2013.

[Le 12] Wangchao Le, Anastasios Kementsietsidis, Songyun Duan & Feifei Li. *Scalable multi-query optimization for SPARQL.* In ICDE. IEEE, 2012.

[Lee 13] Yeonhee Lee & Youngseok Lee. *Toward Scalable Internet Traffic Measurement and Analysis with Hadoop.* CCR 43(1), 2013.

[Levy 95] Alon Y Levy, Alberto O Mendelzon & Yehoshua Sagiv. *Answering queries using views.* In ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems, pages 95–104. ACM, 1995.

[Li 13] Bingdong Li, Mehmet Hadi Gunes, George Bebis & Jeff Springer. *A Supervised Machine Learning Approach to Classify Host Roles On Line Using sFlow.* In ACM HPDC, 2013.

[Lohr 12] Steve Lohr. *The age of big data.* New York Times, vol. 11, 2012.

[Lorey 13] Johannes Lorey & Felix Naumann. *Caching and Prefetching Strategies for SPARQL Queries.* In ESWC. 2013.

[Louis 75] Bentley Jon Louis. *Multidimensional Binary Search Trees Used for Associative Searching.* Comm. of the ACM 18(9), 1975.

[Luo 12] Yongming Luo, François Picalausa, George HL Fletcher, Jan Hidders & Stijn Vansummeren. *Storing and indexing massive RDF datasets.* In Semantic search over the web, pages 31–60. Springer, 2012.

[MahmoudiNasab 10] Hooran MahmoudiNasab & Sherif Sakr. *Efficient and adaptable query workload-aware management for RDF data.* In Web Information Systems Engineering–WISE 2010, pages 390–399. Springer, 2010.

[Manola 04] Frank Manola, Eric Miller, Brian McBride*et al.* *RDF primer.* W3C recommendation, vol. 10, no. 1-107, page 6, 2004.

[Manyika 11] James Manyika, Michael Chui, Brad Brown, Jacques Bughin, Richard Dobbs, Charles Roxburgh & Angela H Byers. *Big data: The next frontier for innovation, competition, and productivity.* 2011.

[Martin 10] Michael Martin, Jörg Unbehauen & Sören Auer. *Improving the performance of semantic web applications with SPARQL query caching.* In The Semantic Web: Research and Applications. 2010.

[McKay 81] Brendan D McKay. Practical Graph Isomorphism. Department of Computer Science, Vanderbilt University, 1981.

[McKay 90] Brendan D McKay & Adolfo Piperno. *Nauty and Traces User's Guide*, 1990.

[McKay 14] Brendan D. McKay & Adolfo Piperno. *Practical graph isomorphism, {II}.* Journal of Symbolic Computation, vol. 60, no. 0, pages 94 – 112, 2014.

[Moerkotte 06] Guido Moerkotte & Thomas Neumann. *Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products.* In VLDB, 2006.

[Moerkotte 08] Guido Moerkotte & Thomas Neumann. *Dynamic programming strikes back.* In In Proceedings of the ACM SIGMOD International Conference on Management of Data, 2008.

[Neumann 10a] Thomas Neumann & Gerhard Weikum. *The RDF-3X Engine for Scalable Management of RDF Data.* VLDBJ, vol. 19, no. 1, 2010.

[Neumann 10b] Thomas Neumann & Gerhard Weikum. *x-RDF-3X: Fast Querying, High Update Rates, and Consistency for RDF Databases.* PVLDB, vol. 3, no. 1-2, 2010.

[Neumann 11] Thomas Neumann & Guido Moerkotte. *Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins.* In IEEE 27th International Conference on Data Engineering (ICDE), pages 984–994. IEEE, 2011.

[Okcan 11] A. Okcan & M. Riedewald. *Processing Theta-Joins using MapReduce.* In ACM SIGMOD, 2011.

[Papailiou 12] Nikolaos Papailiou, Ioannis Konstantinou, Dimitrios Tsoumakos & Nectarios Koziris. *H_2RDF: Adaptive Query Processing on RDF Data in the Cloud.* In WWW, 2012.

[Papailiou 13] Nikolaos Papailiou, Ioannis Konstantinou, Dimitrios Tsoumakos, Panagiotis Karras & Nectarios Koziris. *H2RDF+: High-performance Distributed Joins over Large-scale RDF Graphs.* In IEEE BigData, 2013.

[Papailiou 14] Nikolaos Papailiou, Dimitrios Tsoumakos, Ioannis Konstantinou, Panagiotis Karras & Nectarios Koziris. *H2RDF+: An Efficient Data Management System for Big RDF Graphs.* In Proceedings of the ACM SIGMOD International Conference on Management of Data, 2014.

[Papailiou 15] Nikolaos Papailiou, Dimitrios Tsoumakos, Panagiotis Karras & Nectarios Koziris. *Graph-Aware, Workload-Adaptive SPARQL Query Caching.* In Proceedings of the ACM SIGMOD International Conference on Management of Data, pages 1777–1792, 2015.

[Pham 15] Minh-Duc Pham, Linnea Passing, Orri Erling & Peter Boncz. *Deriving an Emergent Relational Schema from RDF Data.* In Proceedings of the 24th International Conference on World Wide Web, pages 864–874. International World Wide Web Conferences Steering Committee, 2015.

[Poese 10] Ingmar Poese, Benjamin Frank, Bernhard Ager, Georgios Smaragdakis & Anja Feldmann. *Improving Content Delivery using Provider-aided Distance Information.* In IMC, 2010.

[Prud'hommeaux 06] Eric Prud'hommeaux & Andy Seaborne. *SPARQL Query Language for RDF.* W3C recommendation, 2006. http://www.w3.org/TR/rdf-sparql-query/.

[Roy 00] Prasan Roy, Sridhar Seshadri, S Sudarshan & Siddhesh Bhobe. *Efficient and extensible algorithms for multi query optimization*. In ACM SIGMOD Record, volume 29, pages 249–260. ACM, 2000.

[Sahoo 10] Satya Sanket Sahoo. *Semantic Provenance: Modeling, Querying, and Application in Scientific Discovery*. PhD thesis, Wright State University, 2010.

[Shu 13] Yanfeng Shu, Michael Compton, Heiko Müller & Kerry Taylor. *Towards content-aware sparql query caching for semantic web applications*. In WISE 2013. 2013.

[Shvachko 10] Konstantin Shvachko, Hairong Kuang, Sanjay Radia & Robert Chansler. *The Hadoop Distributed File System*. In IEEE MSST, 2010.

[Stuckenschmidt 04] Heiner Stuckenschmidt, Richard Vdovjak, Geert-Jan Houben & Jeen Broekstra. *Index structures and algorithms for querying distributed RDF repositories*. In WWW, 2004.

[Tang 11] Liyin Tang & Namit Jain. *Join Strategies in Hive*. Hive Summit, 2011.

[Thusoo 09] Ashish Thusoo*et al*. *Hive: A Warehousing Solution over a Map-Reduce Framework*. VLDB, 2009.

[Tran 10] Thanh Tran & Günter Ladwig. *Structure index for RDF data*. In SemData, 2010.

[Tsialiamanis 12] Petros Tsialiamanis, Lefteris Sidirourgos, Irini Fundulaki, Vassilis Christophides & Peter Boncz. *Heuristics-based query optimisation for SPARQL*. In ICDT. ACM, 2012.

[Udrea 07] Octavian Udrea, Andrea Pugliese & VS Subrahmanian. *GRIN: A graph based RDF index*. In AAAI, volume 1, pages 1465–1470, 2007.

[W3C 15] W3C. *LargeTripleStores*. http://www.w3.org/wiki/LargeTripleStores, 2015.

[Wang 10] Yan Wang, Xiaoyong Du, Jiaheng Lu & Xiaofang Wang. *FlexTable: using a dynamic relation model to store RDF data*. In Database Systems for Advanced Applications, pages 580–594. Springer, 2010.

[Weiss 08] Cathrin Weiss, Panagiotis Karras & Abraham Bernstein. *Hexastore: Sextuple Indexing for Semantic Web Data Management*. PVLDB, vol. 1, no. 1, 2008.

[Yan 04] Xifeng Yan, Philip S Yu & Jiawei Han. *Graph indexing: a frequent structure-based approach*. In SIGMOD, 2004.

[Yang 07] Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao & D Stott Parker. *Map-Reduce-Merge: Simplified Relational Data Processing on Large Clusters*. In ACM SIGMOD, 2007.

[Yang 11] Mengdong Yang & Gang Wu. *Caching intermediate result of sparql queries*. In WWW, 2011.

[Yu 03] Jeffrey Xu Yu, Xin Yao, Chi-Hon Choi & Gang Gou. *Materialized view selection as constrained evolutionary optimization*. IEEE Transactions on Applications and Reviews, 2003.

[Zaharia 10] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker & Ion Stoica. *Spark: Cluster Computing with Working Sets*. In USENIX conference on Hot topics in cloud computing, 2010.

[Zeng 13] Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao & Zhongyuan Wang. *A Distributed Graph Engine for Web Scale RDF Data*. PVLDB, vol. 6, no. 4, 2013.

[Zhao 07] Peixiang Zhao, Jeffrey Xu Yu & Philip S Yu. *Graph indexing: tree+delta<=graph*. In VLDB, 2007.