SUPPORTING SCALABLE DATA ANALYTICS ON LARGE LINKED DATA

by

Wangchao Le

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computing

School of Computing

The University of Utah

December 2013

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of Wangchao Le

has been approved by the following supervisory committee members:

Feifei Li	, Chair	Aug 28th 2013 Date Approved
Suresh Venkatasubramanian	, Member	July 30th 2013 Date Approved
Piyush Kumar	, Member	Aug 26th 2013 Date Approved
Anastasios Kementsietsidis	, Member	July 30th 2013 Date Approved
Jeff M. Phillips	, Member	July 30th 2013 Date Approved

and by Alan L. Davis , Chair/Dean of

the Department/College/School of Computing

and by David B. Kieda, Dean of The Graduate School.

ABSTRACT

Linked data are the de-facto standard in publishing and sharing data on the web. To date, we have been inundated with large amounts of ever-increasing linked data in constantly evolving structures. The proliferation of the data and the need to access and harvest knowledge from distributed data sources motivate us to revisit several classic problems in query processing and query optimization.

The problem of answering queries over views is commonly encountered in a number of settings, including while enforcing security policies to access linked data, or when integrating data from disparate sources. We approach this problem by efficiently rewriting queries over the views to equivalent queries over the underlying linked data, thus avoiding the costs entailed by view materialization and maintenance. An outstanding problem of query rewriting is the number of rewritten queries is exponential to the size of the query and the views, which motivates us to study problem of multiquery optimization in the context of linked data. Our solutions are declarative and make no assumption for the underlying storage, *i.e.,* being store-independent. Unlike relational and XML data, linked data are schema-less. While tracking the evolution of schema for linked data is hard, keyword search is an ideal tool to perform data integration. Existing works make crippling assumptions for the data and hence fall short in handling massive linked data with tens to hundreds of millions of facts. Our study for keyword search on linked data brought together the classical techniques in the literature and our novel ideas, which leads to much better query efficiency and quality of the results. Linked data also contain rich temporal semantics. To cope with the ever-increasing data, we have investigated how to partition and store large temporal or multiversion linked data for distributed and parallel computation, in an effort to achieve load-balancing to support scalable data analytics for massive linked data.

To My Beloved Family.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

Over the course of my PhD journey, my footprints in pursuit of knowledge have been left across the country, from New York to California, from Gulf Stream waters to Great Rocky mountains. As this journey is now about to end, it seems six years of days and nights, full of joy, upset, and stress have flown away in the blink of an eye. Looking back to this journey, I have been fortunate to have guidance from many people, without whom I might not have reached this stage of my career.

Foremost, I am greatly indebted to my advisor Feifei for his excellent guidance and generous support throughout this dissertation research. I have benefited a lot from his insightful criticism and our countless discussions at various stages of my PhD journey, from choosing a promising topic to correcting a tiny grammatical error. I would like to thank Feifei for the time and effort he has invested on my training. In addition, I would like to express my deepest gratitude to my great mentors Dr. Anastasios Kementsietsidis and Dr. Songyun Duan at IBM T. J. Watson research center. During my internships at IBM, we have together conducted pioneering research on RDF data, which eventually became an important part of this dissertation. I am also very grateful to other members in my supervisory committee. I would like to thank Prof. Suresh Venkatasubramanian and Prof. Jeff Phillips, who have given me valuable feedback for my research and my presentation skills in the group meetings. I would also like to offer my special thanks to Prof. Piyush Kumar, who has been on my committee since I was at Florida State and has given me a lot of support in my career development. I also want to thank all my other collaborators, Dr. Min Wang, Prof. Yufei Tao, and Prof. Ke Yi, whose input has brought this dissertation to a standard higher than what I can expect.

Finally, my sincere thanks goes to the people in the Lab for our friendships. I treasure the joyful moments we have had in the past few years.

CHAPTER 1

INTRODUCTION

The goal of this dissertation research is to design, implement, and evaluate novel query processing and query optimization techniques, in a vision to support scalable data analytics for large linked data. To this end, we have conducted research on four closely related problems: (I) how to efficiently rewrite queries on linked data views; (II) how to synthesize the rewritten queries and perform multiquery optimization; (III) how to summarize the linked data and use the summarization to answer keyword queries; and lastly, (IV) how to partition multiversioned RDF data for supporting parallel and distributed processing.

1.1 Motivation, Background, and Outline

Our world is now awash with rapidly growing data from the world wide web in constantly evolving structures. The overwhelming diversity and amounts of data on the web drive the pressing need to efficiently understand the data out of the chaos. Just as the web has radically changed the way we consume information, so can it revolutionize the way we *access*, *integrate*, and *discover* knowledge from data. To this end, one of the fundamental challenges is to effectively represent knowledge on the web. Linked data is a data model proposed by W3C for publishing and sharing schema-free structure information on the web. The idea is to encode data in a machine-readable format, in an effort to augment the human-readable HTML documents that are hard to interpret by machines. At a larger scale, the linked data model provides a publishing and sharing paradigm where data from different sources on the web can be identified under a global data space (*e.g.,* by URI), thereby enabling interlinking across data sources on the web. Moreover, the linkages across applications, enterprises, and domains allow machines to effectively consume, digest, and reason about the structure and semantics in the data.

The foundation of linked data is built on W3C's standard – the Resource Description Framework (RDF) [8]. Recently, RDF data have been gaining strong momentum in numerous applications, including building a large-scale knowledge base and enabling data sharing on

Web 2.0 platforms. Figure 1.1 shows the state of the Linked Open Data project [5] as of September 2011. Each node in the data cloud stands for a data source, where the respective RDF data are stored in an RDF store and queryable by using the HTTP protocol and the standard query language for RDF data – SPARQL [9]. Thanks to the flexibility of RDF, the Link Open Data project is able to integrate RDF data from more than 300 data sources and applications of different domains and encodes billions of facts to date.

As the size of the data continues to grow, so is the concern for its efficient management and querying evaluation. In an effort to push the vision of linked data, people soon realize that the schema-free and the "pay-as-you-go" natures of RDF data put forward significant challenges to the efficient and scalable querying and storage of the data.

One of the major challenges, which we are going to discuss in Chapter 3, is how to systematically rewrite a SPARQL query over a set of data sources. This problem commonly arises in a number of settings, including while enforcing access control policies over RDF stores and querying RDF data from multiple different data sources. To guarantee equivalent semantics, the rewriting process oftentimes generates an exponential number of rewritten queries, which become extremely inefficient to execute. This motivates us to investigate

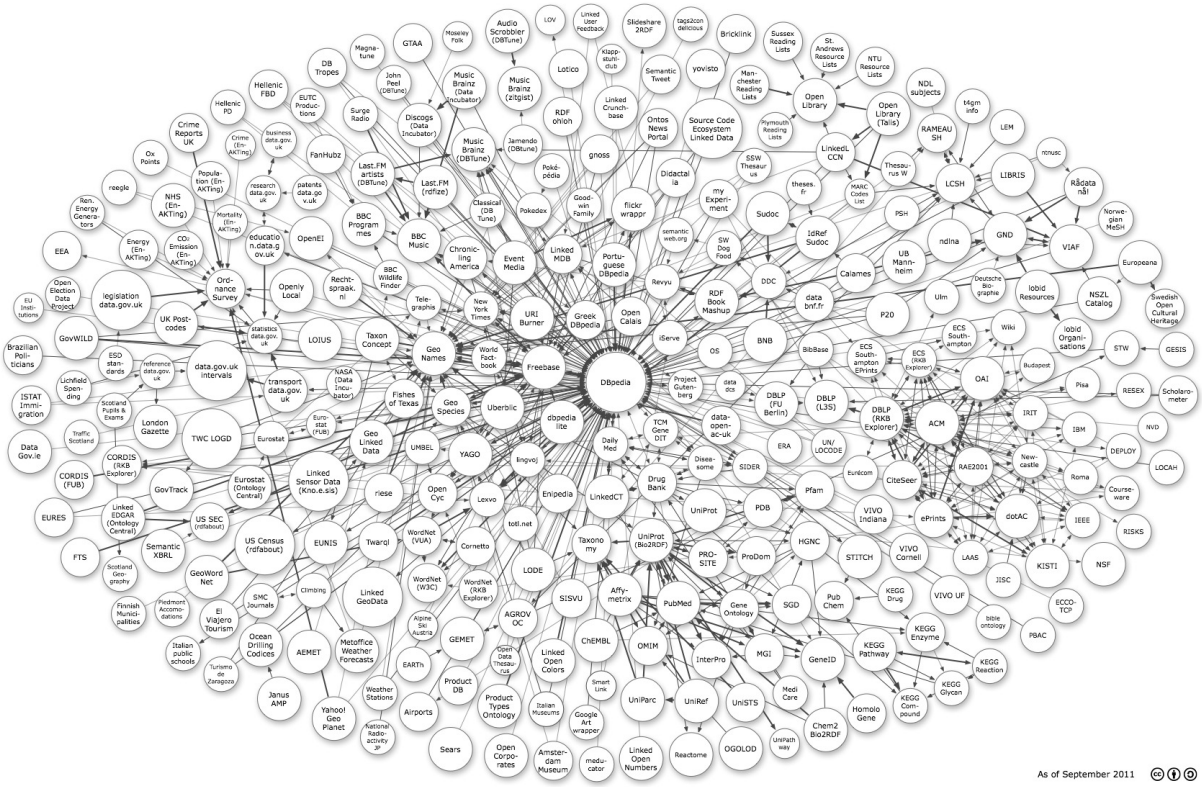


Figure 1.1: Linked Open Data [39]

new rewriting and optimization techniques that can make the query evaluation efficient and tractable. The problems of query rewriting and its optimization have been extensively studied in the literature for relational databases. The goal for this part of our dissertation is to bring together the classic techniques from more than 30 years of research and development in relational databases and our novel ideas to address the problems in the context of RDF and SPARQL. Since the new queries from the rewriting procedure typically share overlapping semantics, a related problem is how to perform multiquery optimization. In Chapter 4, we revisited the classic problem of multiquery optimization, in an effort to achieve practical performance for query rewriting. One design principle we have applied in tackling these challenges is to keep our solutions store-independent. This design choice is very critical for a practitioner to apply the proposed techniques to the off-the-shelf RDF stores, where the implementations took drastically different approaches and architectures [12, 29, 80]. To confirm the effectiveness of the proposed ideas, we have conducted extensive experimental studies with benchmark data sets on *all* popular RDF stores.

Like SQL and other structured query languages, querying with SPARQL assumes users have a good understanding for the schema of the RDF data. However, given the schema-free nature of RDF, a more attractive way is to interact with the data by using keyword search. Keyword search is a useful tool to explore the structure and/or the semantics of linked data, which relieves the users from a steep learning curve of mastering a structured query language like SPARQL, and tracking the constantly evolving schema of the underlining RDF data. In light of its importance, we have extended our studies to keyword search over RDF data. The problem of keyword search has been studied in various similar contexts, including relational, XML, and graph databases. However, when it comes to dealing with massive RDF data sets that have hundreds of millions of facts, the state-of-the-art keyword search techniques have two notable shortcomings. They either rely on building distance matrices for the data, which is hardly scalable nor efficiently updatable, or rely on crippling assumptions for the RDF data model, which easily leads to incorrect interpretation for keyword queries in practice. Existing solutions fall short one way or the other in evaluating keyword queries on real RDF data sets, which motivates us to study efficient and scalable alternatives. In particular, our approach constructs a summarization for the underlying RDF data from its ontology. By leveraging on such an index of the data, the search algorithm can quickly navigate through a large portion of the data that are irrelevant to the keyword query.

In addition to query processing and query optimization, new challenges also come from the management of the constantly evolving linked data. Specially, as RDF data is known to

contain rich temporal semantics, applications often need to capture the changes of ontology or temporal annotation on the RDF data. To see this, consider a simple example[1] from DBpedia [3] – a Linked Open knowledge base built from Wikipedia. In the long and varied history of the city of Istanbul, it has been named Byzantium, New Rome, Constantinople and Stamboul, each of which associates with a period of time. At various times, the city has also been chosen as the capital for the Roman Empire, the Byzantine/Eastern Roman Empire (twice), the Latin Empire, and the Ottoman Empire. These temporal or multiversion properties are part of the RDF encoding for the city, so that temporal queries such as "name of the city in the sixth crusade" can be correctly answered. Such temporal or multiversion RDF data have attracted increasing attention recently [51, 107]. As the amount of temporal or multiversion RDF data continues to grow, it might be no longer feasible to store an entire RDF data set in a centralized database and still expect to query the data efficiently. To improve query performance and scalability, there have been extensive studies on storing and querying RDF data in a distributed and parallel fashion [81, 92, 104]. Therefore, a challenge here is how to partition and store the temporal or multiversion RDF data for distributed and parallel computation. In Chapter 6, we have studied the problem of finding optimal splitters for temporal and multiversion data, in an effort to create partitions of balanced size, and eliminate the potential single bottleneck in a distributed system.

1.2 Dissertation Outline

The rest of this dissertation will be organized as follows. We first introduce the basics of the RDF data and the SPARQL query language in Chapter 2. In Chapter 3, we study the problem of rewriting (SPARQL) queries over SPARQL views and its optimization techniques. One outstanding problem for the rewriting is that the number of rewritten queries could be potentially exponential to the size of the query and the number of views. We investigated the common semantics among the rewritten queries and propose an effective and efficient multiquery optimization framework in Chapter 4. In Chapter 5, we study the problem of answering keyword search in RDF, which is an indispensable tool to explore the structure and semantics of large, real RDF data sets. Many large RDF data sets have rich temporal semantics. This motivates us to study the problem of finding optimal splitters on temporal and multiversion RDF data in Chapter 6, in an effort to support efficient and scalable distributed and parallel processing of such data. Finally, we conclude and discuss some of the promising open problems in Chapter 8.

[1] http://dbpedia.org/page/Istanbulhttp://dbpedia.org/page/Istanbul.

CHAPTER 2

RDF-SPARQL PRELIMINARIES

In this chapter, we introduce the basics of the RDF data and SPARQL query language. More details for the data and the query language will be introduced in the upcoming chapters as we proceed.

2.1 RDF Basics

The Resource Description Framework (RDF hereafter) [8] is the W3C's recommendation for expressing and exchanging semantic metadata and information on the web. It provides mechanisms to uniquely describe logical or physical resources and model their relationships in a machine-friendly way.

The fundamental building block for RDF data is a triple – (subject, predicate, object). In order to give a formal definition for a triple, we first introduce three disjoint infinite sets – the set of URIs $\mathbb{U}$; the set of blank nodes $\mathbb{B}$, and lastly, the set of literals $\mathbb{L}$.

- The set of $\mathbb{U}$ contains an infinite number of Uniform Resource Identifiers (URIs). Each URI is an ASCII string and uniquely identifies a resource. For instance, the URI http://dbpedia.org/page/Utahhttp://dbpedia.org/page/Utah identifies the state of Utah in the dbpedia data set.
- A blank node in $\mathbb{B}$ can be used to identify anonymous resources. In a nutshell, a blank node usually serves as a wide card in representing a group of data sources.
- A literal in $\mathbb{L}$ represents strings, Boolean, or numerical values. Usually, a literal is used for encoding a property of the resource that is described.

Definition 1 *Formally, an* RDF *triple* (s, p, o) *is defined as*

$$(s, p, o) \in \{\mathbb{B} \cup \mathbb{U}\} \times \{\mathbb{U}\} \times \{\mathbb{B} \cup \mathbb{U} \cup \mathbb{L}\}$$

By convention, we also refer to the *predicate* as the *property* of the subject being described. For the ease of presentation, we will ignore blank nodes in the rest the dissertation,

though the techniques we discuss can gracefully handle such an extension. Implicitly, a set of triples forms a graph by joining on the subjects and/or objects. Therefore, an RDF data set is also commonly referred to as an RDF data graph. In what follows, we use the two terms interchangeably.

There is a lot of ongoing research studying how to store and index RDF data to support efficient querying [12, 29, 80]. Popular RDF stores can be classified as either *generic/native* stores or *relational* stores. Native RDF stores view RDF data as graphs and recent studies [23, 80] have proposed a full spectrum of techniques to efficiently index RDF data, for instance, Jena TDB [4], 4store [1], Sesame native [30], and RDF3X [80]. On the other hand, relational RDF stores use relational databases to process and store RDF data, to tap the power of relational engines, for instance, OpenLink Virtuoso [11], Jena SDB [4], and IBM DB2 [29]. To the best of our knowledge, both generic and relational RDF stores are widely deployed as a medium to access RDF data.

2.2 SPARQL Basics

Regardless of underlying implementation of the RDF stores, the standard media to interact with RDF data is to express a user's query in SPARQL, which is the recommendation by W3C to query RDF data. SPARQL is a pattern matching language. The most common SPARQL queries have the following form: Q := (SELECT |CONSTRUCT) RD WHERE GP [OPTIONAL $\mathsf{GP}_{\mathsf{OPT}}$]*, where GP are *triple patterns*, *i.e.,* triples involving variables and/or constants, and RD is the *result description*. Given an RDF graph G, a triple pattern on G searches for a set of subgraphs of G, each of which matches the pattern (by binding pattern variables to values in the subgraph). For SELECT queries, RD is a subset of variables in the graph pattern, similar to a projection in SQL. An example for the selection query is shown in Figure 2.1(a), where the variables are starting with '?'. The query asks for all the predicates and objects for the state of Utah, *i.e.,* listing its properties. Evaluating the query on the DBpedia data set will result in the (partial) answer as shown in Figure 2.1(b), where each row represents a distinct pair of valid variable bindings. On the other hand, for CONSTRUCT queries, RD is a set of triple templates that construct a *new* RDF graph by replacing variables in GP with matched values. By doing so, a user can define new graphs from the matchings of GP in the data.

```
SELECT * WHERE{ <http://dbpedia.org/page/Utah>  ?p  ?o }
```

(a)

?p	?o
name	Utah
nickname	Beehive State
motoo	Industry
mineral	Copper

(b)

Figure 2.1: An example of SPARQL query: (a) A SPARQL query, and (b) a part of the variable bindings

A useful extension to specify the search pattern in the WHERE clause is to combine a GP with one or more OPTIONAL clause(s), each of which is a graph pattern $\mathsf{GP}_{\mathsf{OPT}}$. A subgraph in the RDF data might match not only the pattern in GP but also the pattern (combination) of GP and $\mathsf{GP}_{\mathsf{OPT}}$. While more than one OPTIONAL clauses are allowed, the evaluation process in the engine independently considers the combination of pattern GP with each of the OPTIONAL clauses. Therefore, with n OPTIONAL clauses in query, the query returns two sets of results. In more detail, the first set of the results contains the subgraphs that match any of the n ($\mathsf{GP} + \mathsf{GP}_{\mathsf{OPT}}$) pattern combinations. The second set of the results contains the subgraphs that match just the GP pattern. Finally, we consider Boolean SPARQL queries of the form $\mathsf{Q} := \mathsf{ASK}\ \mathsf{GP}$ which indicate whether GP exists, or not, in G. Similar to SQL where research considered set before bag semantics, for our non-Boolean SPARQL queries, we assume set semantics, whose importance for SPARQL has already been noted [87].

CHAPTER 3

REWRITING QUERIES ON SPARQL VIEWS

3.1 Introduction

In a number of settings, including access control [44, 45, 94, 113] or data integration [74, 111], users can only access data that are *visible* through a set of views. The views are typically defined using a standard query language (SQL for relational data, XPath/XQuery for XML, SPARQL for RDF) and commonly the same language is used by the users to express the queries over the views. The process of answering these user queries is determined on whether the views are *virtual* or *materialized*. For materialized views, evaluating the user queries is straightforward, but the simplicity in query evaluation comes at a cost, both in terms of the space required to save the views, and in terms of the time needed to maintain the views. Therefore, view materialization is a viable alternative only when (i) there are a small number of views; (ii) the views expose small fragments of base data; and (iii) the base data are infrequently updated. Since most practical scenarios do not meet these requirements, the other alternative is to use virtual view and *rewrite* the queries over the views to equivalent queries over the underlying data. In relational databases, query rewriting over SQL views is *straightforward* as it only requires *view expansion*, *i.e.,* the view mentioned in the user SQL query is replaced by its definition. However, in the case of RDF and SPARQL, view expansion is not possible since expansion requires query nesting, a feature not currently supported by SPARQL. In XML, XPath query rewriting is rather involved and the rewriting is exponential to the size of the query and the view [45]. Query rewriting for RDF/SPARQL is inherently more complex since (i) whereas XML/XPath is used for representing and querying trees, RDF/SPARQL considers generic graphs; and (ii) in SPARQL, the query and view definitions may use different variables to refer to the same entity, thus requiring *variable mappings* when synthesizing multiple views to rewrite a given query. Therefore, query rewriting in RDF/SPARQL raises distinct challenges from those in the relational or XML.

To illustrate these challenges, we use a Facebook-inspired example, and in Figure 3.1(a)

we consider RDF triples modeling common *acquaintances* (e.g., friend, related, and works). In such a setting, we can use views to express access control (privacy) policies over Facebook profiles. For instance, for each person (*e.g.,* person0 with name "Eric") we might have a default policy that exposes from the social network only the person's immediate friends (*e.g.,* for person0, person1, and person2), and relatives (*e.g.,* for person0, person3), along with friends-of-friends (FoF), and relatives-of-relatives (RoR), while not exposing the relatives-of-friends, or the friends-of-relatives. Figure 3.1(b) shows four views to enforce this policy (variables are prefixed by '?' and constructed view predicates are prefixed with the letter 'v'). The views hide any distinction between immediate friends (or relatives) and those at a distance of two. Like [94], a parameter $\langle P_i \rangle$ specifies the name of the person for whom the policies are enforced. Figure 3.1(c) shows the result $\mathsf{V}_{\mathsf{Eric}}$ of materializing all four views for "Eric", with each triple annotated by the generating view(s).

Consider the query Q_{U} in Figure 3.1(b) over the triples for "Eric" (shown in Figure 3.1(c)). Q_{U} identifies "Eric"'s friends and relatives who live in the same city. Instead of materializing $\mathsf{V}_{\mathsf{Eric}}$ just to evaluate Q_{U}, we would like to use the views to rewrite Q_{U} into a query over the base data in Figure 3.1(a). *The first challenge is to determine which views can be used in this rewriting.* Finding relevant views requires computing *(variable) mappings* between the body of Q_{U} (its WHERE clause) and the return values (CONSTRUCT clause) of the views. An example of a mapping between triples ($?f_0$, vfriend, $?f_1$) in V_{F} and (person0, vfriend, $?f_5$) in Q_{U}, maps $?f_0$ to person0 and $?f_1$ to $?f_5$. The mapping indicates that V_{F} *can be used* for rewriting Q_{U}. *How* it will be used, is our next challenge.

In more detail, *the second challenge is to determine how the views can be combined into a sound and complete rewriting.* Soundness guarantees that the rewritten query only returns results that would have been retrieved should the user query have been executed over the materialized view. Completeness guarantees that the rewritten query returns all these results. Addressing the second challenge requires algorithms that (i) meaningfully combine the views identified in the first step of the rewriting; and (ii) consider all such possible combinations of the views. In our example, a sound and complete rewriting results in a union of *64 queries*, with each query being a result of one unique combination of views. In more detail, each view combination is a result of combining 2 possible var. mappings for each instance of vfriend and vrelated, and 4 possible var. mappings for each instance of lives. Clearly, there is an exponential blowup in the size of the rewritten query, with respect to the size of the input query and the number of views. Combining view directly generates rewritings that have empty results, which provides optimization opportunities by removing

(person0, name, Eric)
(person1, name, Kenny)
(person2, name, Stan)
(person3, name, Kyle)
(person5, name, Jimmy)
(person6, name, Timmy)
(person9, name, Danny)
(person0, lives, NYC)
(person1, lives, LA)
(person2, lives, NYC)
(person3, lives, NYC)
(person5, lives, NYC)
(person6, lives, CHI)
(person9, lives, LA)
(person0, friend, person1)
(person0, friend, person2)
(person1, friend, person2)
(person1, friend, person5)
(person2, friend, person6)
(person3, friend, person8)
(person0, related, person3)
(person3, related, person9)
(person0, works, person4)
(person2, works, person7)

(a)

View V_F

```
CONSTRUCT {
 ① ?f0 vfriend ?f1,
 ② ?f1 vname ?n1, ③ ?f1 vlives ?l1 }
WHERE {
  ?f0 name ⟨P1⟩, ?f0 friend ?f1,
  ?f1 name ?n1, ?f1 lives ?l1 }
```

View V_{FoF}

```
CONSTRUCT {
 ① ?f2 vfriend ?f4,
 ② ?f4 vname ?n4, ③ ?f4 vlives ?l4 }
WHERE {
  ?f2 name ⟨P2⟩,
  ?f2 friend ?f3, ?f3 friend ?f4,
  ?f4 name ?n4, ?f4 lives ?l4 }
```

View V_R

```
CONSTRUCT {
 ① ?r0 vrelated ?r1,
 ② ?r1 vname ?n1, ③ ?r1 vlives ?l1 }
WHERE {
  ?r0 name ⟨P3⟩, ?r0 related ?r1,
  ?r1 name ?n1, ?r1 lives ?l1 }
```

View V_{RoR}

```
CONSTRUCT {
 ① ?r2 vrelated ?r4,
 ② ?r4 vname ?n4, ③ ?r4 vlives ?l4 }
WHERE {
  ?r2 name ⟨P4⟩,
  ?r2 related ?r3, ?r3 related ?r4,
  ?r4 name ?n4, ?r4 lives ?l4 }
```

Query Q_U

```
SELECT { ?f5, ?r5, ?l5 }
WHERE {
① person0 vfriend ?f5, ② ?f5 vlives ?l5,
③ person0 vrelated ?r5, ④ ?r5 lives ?l5 }
```

(b)

Figure 3.1: Motivating example (a) Base triples, (b) Views and user query, and (c) Materialized triples in V_{Eric}

(person1, vname, Kenny) $[V_{\mathsf{F}}]$
(person2, vname, Stan) $[V_{\mathsf{F}}, V_{\mathsf{FoF}}]$
(person3, vname, Kyle) $[V_{\mathsf{R}}]$
(person5, vname, Jimmy) $[V_{\mathsf{FoF}}]$
(person6, vname, Timmy) $[V_{\mathsf{FoF}}]$
(person9, vname, Danny) $[V_{\mathsf{RoR}}]$
(person1, vlives, LA) $[V_{\mathsf{F}}]$
(person2, vlives, NYC) $[V_{\mathsf{F}}, V_{\mathsf{FoF}}]$
(person3, vlives, NYC) $[V_{\mathsf{R}}]$
(person5, vlives, NYC) $[V_{\mathsf{FoF}}]$
(person6, vlives, CHI) $[V_{\mathsf{FoF}}]$
(person9, vlives, LA) $[V_{\mathsf{RoR}}]$
(person0, vfriend, person1) $[V_{\mathsf{F}}]$
(person0, vfriend, person2) $[V_{\mathsf{F}}, V_{\mathsf{FoF}}]$
(person0, vrelated, person3) $[V_{\mathsf{R}}]$
(person0, vfriend, person5) $[V_{\mathsf{FoF}}]$
(person0, vfriend, person6) $[V_{\mathsf{FoF}}]$
(person0, vrelated, person9) $[V_{\mathsf{RoR}}]$

(c)

Figure 3.1: Continued

the empty rewritings from evaluation. For this particular example, *only four of these combinations* need to be evaluated (the others are either subsumed by these four, or return no results). Therefore, *our third challenge is to optimize the rewriting and evaluate only a subset of the view combinations without sacrificing soundness or completeness.*

Given that relational algebra (and the corresponding SQL fragment) has the same expressive power as SPARQL [20], one might be tempted to address the SPARQL rewriting problem by considering the corresponding SQL setting and applying the solutions in SQL. Although this seems promising since some RDF stores do use a relational back-end (*e.g.,* Jena SDB [4], Virtuoso [11], C-store [12], *etc.*), we show here that for a number of reasons, such an approach does *not* reduce the complexity. To translate our setting to the relational case, we use one of the most efficient relational storage strategies for RDF, namely, predicate tables [12] (column-store style storage); our observations are independent of this choice. So, we have a database with five tables: name(s, o), lives(s, o), friend(s, o), related(s, o), and works(s, o), whose contents are easily inferred by the corresponding triples in Figure 3.1(a). In Figures 3.2(a) and (c), we show the SQL translations of the views and query of Figure 3.1(b). During this translation, we need to create the corresponding *view* predicate tables of the base database tables. So, as shown in Figure 3.2(b), we need to create the vfriend table which contains the friend subjects and objects returned by the $V_{\text{F-SQL}}$ and $V_{\text{FoF-SQL}}$ views (similarly for vrelated and vlives). How can we rewrite $Q_{\text{U-SQL}}$ to a query over the base five tables? Since view expansion is supported in SQL, we can replace in $Q_{\text{U-SQL}}$ the vfriend, vrelated, and vlives tables with their definitions in Figure 3.2(b), and in turn replace $V_{\text{F-SQL}}$, $V_{\text{FoF-SQL}}$, $V_{\text{R-SQL}}$, and $V_{\text{RoR-SQL}}$ with their definitions in Figure 3.2(a). Finally, it is not hard to see that the rewriting of $Q_{\text{U-SQL}}$ results in a union of 64 queries, the same blow-up in size as the one observed in SPARQL. So, moving from SPARQL to SQL does not reduce the complexity of the problem (more exposition in Section 3.5); we will validate this observation in Section 3.4. Such a move is also prohibitive as there is an increasing number of stores (*e.g.,* Jena TDB [4], 4store [1]) using native RDF storage. For these stores, translation to SQL does not work. Therefore, it is necessary to have a *native* and *efficient* SPARQL rewriting algorithm, which has the advantage of being *generic* since it works on any existing RDF store irrespectively of the storage model used. Our contributions can be summarized as follows:

1. We study the rewriting of SPARQL queries over virtual SPARQL views, and propose a *native* SPARQL rewriting algorithm (Section 3.2), and prove that it generates *sound* and *complete* rewritings.

$V_{F\text{-}SQL}$
SELECT F.s, F.o, N'.s, N'.o, L.s, L.o
FROM name N, friend F, name N', lives L
WHERE N.s =F.s AND N.o =$\langle P_1 \rangle$ AND
N'.s =F.o AND L.s =F.o

$V_{R\text{-}SQL}$
SELECT R.s, R.o, N'.s, N'.o, L.s, L.o
FROM name N, related R, name N', lives L
WHERE N.s =R.s AND N.o =$\langle P_3 \rangle$ AND
N'.s =R.o AND L.s =R.o

$V_{FoF\text{-}SQL}$
SELECT F.s, F'.o, N'.s, N'.o, L.s, L.o
FROM name N, friend F,
friend F', name N', lives L
WHERE N.s =F.s AND N.o =$\langle P_2 \rangle$ AND
F.o =F'.s AND N'.s =F'.o AND
L.s =F'.o

$V_{RoR\text{-}SQL}$
SELECT R.s, R'.o, N'.s, N'.o, L.s, L.o
FROM name N, related R,
related R', name N', lives L
WHERE N.s =R.s AND N.o =$\langle P_4 \rangle$ AND
R.o =R'.s AND N'.s =R'.o AND
L.s =R'.o

(a)

vfriend:
SELECT fs, fo FROM $V_{F\text{-}SQL}$
UNION
SELECT fs, fo FROM $V_{FoF\text{-}SQL}$

vrelated:
SELECT rs, ro FROM $V_{R\text{-}SQL}$
UNION
SELECT rs, ro FROM $V_{RoR\text{-}SQL}$

vlives:
SELECT ls, lo FROM $V_{F\text{-}SQL}$
UNION
SELECT ls, lo FROM $V_{FoF\text{-}SQL}$
UNION
SELECT ls, lo FROM $V_{R\text{-}SQL}$
UNION
SELECT ls, lo FROM $V_{RoR\text{-}SQL}$

(b)

$Q_{U\text{-}SQL}$:
SELECT F.o, R.o, L.o
FROM vfriend F, vlives L, vlives L', vrelated R
WHERE F.s = person0 AND F.o = L.s AND R.s = person0 AND R.o = L'.s AND L.o = L'.o

(c)

Figure 3.2: Attempting a relational/SQL rewriting: (a) SQL translation of V_F, V_R, V_{FoF}, V_{RoR}; (b) Secure predicate tables definitions; (c) SQL translation of query Q_U

2. We propose several optimizations of the basic rewriting algorithm to reduce the complexity (Section 3.3.1) and size of the rewritten queries (Sections 3.3.2 and 3.3.3), while employing novel optimization techniques customized for our needs.

3. We present extensive experiments on two RDF stores (Section 3.4) on the scalability and portability of our algorithms. The optimizations result in order of magnitude improvements in rewritten query sizes and evaluation times over our basic rewriting algorithm in SPARQL;

the latter is comparable to applying rewriting techniques in SQL after translating SPARQL queries into SQL queries.

We survey the related work in Section 3.5 and conclude the chapter in Section 3.6.

3.2 Query Rewriting in SPARQL

SPARQL is a pattern-matching query language. In this part of the dissertation, we are going to focus on two types of SPARQL queries. The most common SPARQL queries have the following form: Q := (SELECT | CONSTRUCT) RD (WHERE GP), where GP are *triple patterns*, *i.e.,* triples involving variables and/or constants, and RD is the *result description*. Given an RDF graph G, a triple pattern on G searches for a set of subgraphs of G, each of which *matches* the pattern (by binding pattern variables to values in the subgraph). For SELECT queries, RD is a subset of variables in the graph pattern, similar to a projection in SQL. This is the case for query $\mathsf{Q_U}$ in Figure 3.1(b). For CONSTRUCT queries, RD is a set of triple templates that construct a *new* RDF graph by replacing variables in GP with matched values. This is the case for the views in Figure 3.1(b). Finally, we consider boolean SPARQL queries of the form ASK GP which indicate whether GP exists, or not, in G. Similar to SQL where research considered set before bag semantics, for our non-Boolean SPARQL queries, we assume set semantics whose importance for SPARQL has already been noted [87].

The central technical problem in this chapter is the *rewriting problem* as follows: given a set of views $\mathcal{V} = \{\mathsf{V}_1, \mathsf{V}_2, \ldots, \mathsf{V}_l\}$ over an RDF graph G, and a SPARQL query Q over the vocabulary of the views, compute a SPARQL query Q′ over G such that $\mathsf{Q}'(\mathsf{G}) = \mathsf{Q}(\mathcal{V}(\mathsf{G}))$. Like [113], we consider two criteria on the correctness of a rewriting, namely, *soundness* and *completeness*.

1. The rewriting is sound iff $\mathsf{Q}'(\mathsf{G})$ is contained in $\mathsf{Q}(\mathcal{V}(\mathsf{G}))$,
   i.e., $\mathsf{Q}'(\mathsf{G}) \subseteq \mathsf{Q}(\mathcal{V}(\mathsf{G}))$
2. The rewriting is complete iff $\mathsf{Q}(\mathcal{V}(\mathsf{G}))$ is contained in
   $\mathsf{Q}'(\mathsf{G})$, *i.e.,* $\mathsf{Q}(\mathcal{V}(\mathsf{G})) \subseteq \mathsf{Q}'(\mathsf{G})$

Soundness and completeness suffice to show that $\mathsf{Q}(\mathcal{V}(\mathsf{G})) = \mathsf{Q}'(\mathsf{G})$. We will prove our rewriting meet the two criteria.

3.2.1 Rewriting Algorithm

The first challenge in query rewriting (as mentioned in the introduction) is to determine which views can be used for the rewriting. In SPARQL, the crucial observation to address this challenge is that *if a variable mapping exists between a triple pattern* $(\mathsf{s_v}, \mathsf{p_v}, \mathsf{o_v})$ *in the*

result description $\mathsf{RD}(\mathsf{V}_j)$ *of a view* V_j *and one of the triple patterns* $(\mathsf{s_q}, \mathsf{p_q}, \mathsf{o_q})$ *in the graph pattern* $\mathsf{GP}(\mathsf{Q})$ *of query* Q*, then view* V_j *can be used to rewrite* Q . Using this observation, we present Algorithm 1 (SQR) to perform the rewriting in two steps. In the first step (lines 3-18), the algorithm determines, for each triple pattern $p_i(\bar{X}_i)$ in the user query, the set CandV_i of *candidate views* that have a variable mapping to this triple pattern. For ease of presentation, we assume that in our SPARQL queries, the predicate in each triple pattern is a constant (the subject and object can either be variables or constants). Even if a triple has a variable in its predicate, we can simply substitute such a triple by a set of triple patterns, each triple in the set binding the predicate variable to a constant predicate from the active domain of predicates in the RDF store.

Algorithm 1: SPARQL Query Rewriting (SQR) Algorithm

Input: Views $\mathcal{V}$, query Q with $\mathsf{GP}(\mathsf{Q})=(\mathsf{s}_1^{\mathsf{Q}}, \mathsf{p}_1^{\mathsf{Q}}, \mathsf{o}_1^{\mathsf{Q}}), \ldots, (\mathsf{s}_n^{\mathsf{Q}}, \mathsf{p}_n^{\mathsf{Q}}, \mathsf{o}_n^{\mathsf{Q}})$
Output: a rewriting Q' as a union of conjunctive queries

1 **for** *each* $(\mathsf{s}_i^{\mathsf{Q}}, \mathsf{p}_i^{\mathsf{Q}}, \mathsf{o}_i^{\mathsf{Q}})$, $1 \leq i \leq n$ **do**
2 Set CandV_i to $\emptyset$.
3 **for** *each view* $\mathsf{V}_j \in \mathcal{V}$ **do**
4 Let $\mathsf{RD}(\mathsf{V}_j)=(\mathsf{s}_1^{\mathsf{V_j}}, \mathsf{p}_1^{\mathsf{V_j}}, \mathsf{o}_1^{\mathsf{V_j}}), \ldots, (\mathsf{s}_m^{\mathsf{V_j}}, \mathsf{p}_m^{\mathsf{V_j}}, \mathsf{o}_m^{\mathsf{V_j}})$
5 **for** *each* $(\mathsf{s}_k^{\mathsf{V_j}}, \mathsf{p}_k^{\mathsf{V_j}}, \mathsf{o}_k^{\mathsf{V_j}})$, $1 \leq k \leq m$ **do**
6 **if** $\mathsf{p}_i^{\mathsf{Q}} = \mathsf{p}_k^{\mathsf{V_j}}$ **then**
7 Set variable mapping Φ_{ijk} to *undefined*
8 **for** *the pair* $(\mathsf{s}_i^{\mathsf{Q}}, \mathsf{s}_k^{\mathsf{V_j}})$ *of subjects (similarly objects* $(\mathsf{o}_i^{\mathsf{Q}}, \mathsf{o}_k^{\mathsf{V_j}})$*)* **do**
9 **if** *var. mapping* $\phi : \mathsf{s}_i^{\mathsf{Q}} \rightarrow \mathsf{s}_k^{\mathsf{V_j}}$ *exists* **then**
10 **if** ϕ *maps two variables* **then** $\Phi_{ijk}(\mathsf{s}_k^{\mathsf{V_j}}) = s_i^{\mathsf{Q}}$
11 **else** $\Phi_{ijk}(\mathsf{s}_k^{\mathsf{V_j}}) = \mathsf{s}_k^{\mathsf{V_j}}$ ($\mathsf{s}_k^{\mathsf{V_j}}$ is a constant)
12 **if** *var. mapping* $\phi : \mathsf{s}_k^{\mathsf{V_j}} \rightarrow \mathsf{s}_i^{\mathsf{Q}}$ *exists* **then**
13 **if** ϕ *maps a variable to a constant* **then** $\Phi_{ijk}(\mathsf{s}_k^{\mathsf{V_j}}) = \mathsf{s}_i^{\mathsf{Q}}$
14 **if** Φ_{ijk} *is defined* **then**
15 For any variable v' in $\mathsf{RD}(\mathsf{V}_j)$ not in $(\mathsf{s}_k^{\mathsf{V_j}}, \mathsf{p}_k^{\mathsf{V_j}}, \mathsf{o}_k^{\mathsf{V_j}})$, Φ_{ijk} maps v' to a fresh variable (a new variable)
16 Add $(\mathsf{V}_j, \Phi_{ijk})$ to CandV_i
17 Set the query rewriting result Q' to $\emptyset$
18 **for** *each entry in Cartesian product* $\mathsf{CandV}_1 \times \ldots \times \mathsf{CandV}_n$ **do**
19 **if** $\Phi_{1j_1k_1}, \Phi_{2j_2k_2}, \ldots, \Phi_{nj_nk_n}$ *is compatible* **then**
20 $\mathsf{RD}(q') = \mathsf{RD}(\mathsf{Q})$
21 $\mathsf{GP}(q') = \mathsf{GP}(\Phi_{1j_1k_1}(\mathsf{V}_{j_1}), \ldots, \Phi_{nj_nk_n}(\mathsf{V}_{j_n}))$
22 $\mathsf{Q}' = \mathsf{Q}' \cup q'$
23 **return** Q'

Computing variable mappings between triple patterns in SQR is similar to computing *substitutions* between conjunctive queries [14]. Formally, a substitution is a mapping between the corresponding elements (subject, predicate, and object) in a pair of triples that maps: (i) a variable in the first triple to another variable or constant in the second triple; or (ii) a constant in the first triple to the same constant in the second triple. Or, conversely, a substitution cannot map a constant in the first triple to a variable in the second, or map two different constants in the triples. For example, a substitution exists from ($?f_0$, vfriend, $?f_1$) to (person0, vfriend, $?f_5$), which maps the variable $?f_0$ to the constant person0 and the variable $?f_1$ to the variable $?f_5$. There is no substitution from the second to the first triple since we cannot map the constant person0 to the variable $?f_0$.

Unlike substitutions that are *directional*, *i.e.,* the mapping is always from one triple to another, the variable mappings computed here are more complex, since for their creation, we need to compose the (partial) substitutions that exist between the two triples in *both* directions. As an example, consider the triples (person0, vfriend, $?f_5$) and ($?f_6$, vfriend, person1). There is no substitution between the two triples in either of the directions. However, the variable mappings used by our algorithm attempt to compute partial substitutions between the two triples and use those to compute a variable mapping. In our example, our algorithm computes a partial substitution from the first triple to the second that maps $?f_5$ to constant person1. It also computes a partial substitution from the second triple to the first that maps $?f_6$ to constant person0. The combination of the two partial substitutions constitutes a variable mapping. Eventually, this is used to compute a new triple of the form (person0, friend, person1). The computed triple is such that a substitution exists from each of the initial triples to it.

After the var. mapping computation, Algorithm SQR (lines 19-23) constructs in its second step the rewriting as a union of conjunctive queries. Each query in the union is generated by considering one combination from the Cartesian product of the sets CandV_i ($i \in [1, n]$). While considering each combination, we need to make sure that the corresponding variable mappings from individual predicates are *compatible*, *i.e.,* they do not map one variable in the query Q to two different constants (from the views). For the variables only appearing in GP of the views, they are mapped to fresh (*i.e.,* new) variables by default. For each compatible combination, we generate one query in the union.

To illustrate this, consider triples $\mathsf{t}_1^{\mathsf{Q_U}} = (\mathsf{person0}, \mathsf{vfriend}, ?f_5)$ and $\mathsf{t}_2^{\mathsf{Q_U}} = (?f_5, \mathsf{vlives}, ?l_5)$, from $\mathsf{Q_U}$ of Figure 3.1. For $\mathsf{t}_1^{\mathsf{Q_U}}$, $\mathsf{CandV}_1 = \{(\mathsf{V_F}, \Phi_{111}), (\mathsf{V_{FoF}}, \Phi_{121})\}$, where both Φ_{111} and Φ_{121} are shown in Table 3.1(a) (the subscripts of Φs are defined in Algorithm SQR and

labelled in Figure 3.1(b)). Similarly, Table 3.1(b) shows CandV_2 for $\mathsf{t}_2^{\mathsf{Q_U}}$. To get Φ_{111}, SQR first considers $\mathsf{t}_1^{\mathsf{Q_U}}$ with $\mathsf{t}_1^{\mathsf{V_F}} = (?\mathsf{f}_0, \mathsf{vfriend}, ?\mathsf{f}_1)$ from $\mathsf{V_F}$ (lines 3-8). Then, for the pair of subjects $(\mathsf{person0}, ?\mathsf{f}_0)$ (line 10), a var. mapping ϕ exists (line 14) from $?\mathsf{f}_0$ to person0. Therefore, Φ_{111} assigns the constant to the variable (line 15). Next, the pair of objects $(?\mathsf{f}_5, ?\mathsf{f}_1)$ is considered (line 10) and as a result, Φ_{111} assigns $?\mathsf{f}_1$ to $?\mathsf{f}_5$ (lines 11-12). The remaining variables ($?\mathsf{n}_1$ and $?\mathsf{l}_1$) in $\mathsf{V_F}$ are assigned to fresh/new variables, respectively ($?\mathsf{v}_0$ and $?\mathsf{v}_1$) (line 17). This concludes the computation of Φ_{111}. Other Φ's are computed accordingly. To illustrate, we consider the (partial) query $\mathsf{Q_U}^{\text{part}}$ of $\mathsf{Q_U}$ consisting only of triples $\mathsf{t}_1^{\mathsf{Q_U}}$ and $\mathsf{t}_2^{\mathsf{Q_U}}$. Then, there are 8 rewritings of $\mathsf{Q_U}^{\text{part}}$ (lines 20-24), one for each combination of Φ's in CandV_1 and CandV_2. Table 3.1(c) shows the rewriting for $\mathsf{Q_U}^{\text{part}}$, using $(\mathsf{V_{FoF}}, \Phi_{121})$ in CandV_1 and $(\mathsf{V_R}, \Phi_{233})$ in CandV_2.

Theorem 1 *The rewriting* Q' *of* SQR *is sound and complete.*

Proof. We first show the soundness and then prove the completeness.

Algorithm 1 generates a rewriting Q' of the input query $q(\bar{X})$:- $p_1(\bar{X}_1), \ldots, p_n(\bar{X}_n)$ as a union of conjunctive queries, *i.e.,* $\mathsf{Q}' = \cup q'$, with one query q' for each entry in the Cartesian product of $\mathsf{CandV}_1 \times \ldots \times \mathsf{CandV}_n$ (lines 20-24 of the algorithm). To prove soundness, it suffices to show that that $q'(\mathsf{G}) \subseteq q(\mathcal{V}(\mathsf{G}))$, for each q' of Q'.

Let q' be the query corresponding to the $(\Phi_{1j_1k_1}(\mathsf{V}_{j_1}), \ldots \Phi_{nj_nk_n}(\mathsf{V}_{j_n}))$ entry of the Cartesian product, and let $\mathsf{V}'_{j_i} = \Phi_{ij_ik_i}(\mathsf{V}_{j_i})$, $1 \leq i \leq n$, that is, V'_{j_i} is the view obtained by applying the variable mapping $\Phi_{ij_ik_i}$ to view V_{j_i}. It is not hard to see that $\mathsf{V}'_{j_i}(\mathsf{G}) \subseteq \mathsf{V}_{j_i}(\mathsf{G})$.

Table 3.1: Variable mapping example

$(\mathrm{V_F}, \Phi_{111})$: $\Phi_{111}(?\mathsf{f}_0, ?\mathsf{f}_1, ?\mathsf{n}_1, ?\mathsf{l}_1) = (\mathsf{person0}, ?\mathsf{f}_5, ?\nu_0, ?\nu_1)$
$(\mathrm{V_{FoF}}, \Phi_{121})$: $\Phi_{121}(?\mathsf{f}_2, ?\mathsf{f}_4, ?\mathsf{n}_4, ?\mathsf{l}_4) = (\mathsf{person0}, ?\mathsf{f}_5, ?\nu_2, ?\nu_3)$

(a) CandV_1 for triple $(\mathsf{person0}, \mathsf{vfriend}, ?\mathsf{f}_5)$

$(\mathrm{V_F}, \Phi_{213})$: $\Phi_{213}((?\mathsf{f}_1,?\mathsf{l}_1,?\mathsf{f}_0,?\mathsf{n}_1)) = (?\mathsf{f}_5,?\mathsf{l}_5,?\nu_4,?\nu_5)$
$(\mathrm{V_{FoF}}, \Phi_{223})$: $\Phi_{223}((?\mathsf{f}_4,?\mathsf{l}_4,?\mathsf{f}_2,?\mathsf{n}_4)) = (?\mathsf{f}_5,?\mathsf{l}_5,?\nu_6,?\nu_7)$
$(\mathrm{V_R}, \Phi_{233})$: $\Phi_{233}((?\mathsf{r}_1,?\mathsf{l}_1,?\mathsf{r}_0,?\mathsf{n}_1)) = (?\mathsf{f}_5,?\mathsf{l}_5,?\nu_8,?\nu_9)$
$(\mathrm{V_{RoR}}, \Phi_{243})$: $\Phi_{243}((?\mathsf{r}_4,?\mathsf{l}_4,?\mathsf{r}_2,?\mathsf{n}_4)) = (?\mathsf{f}_5,?\mathsf{l}_5,?\nu_{10},?\nu_{11})$

(b) CandV_2 for triple $(?\mathsf{f}_5, \mathsf{vlives}, ?\mathsf{l}_5)$

GP(q')={ person0 name $\langle P \rangle$, person0 friend $?\mathsf{f}'_3$, $?\mathsf{f}'_3$ friend $?\mathsf{f}_5$, $?\mathsf{f}_5$ name $?\nu_2$, $?\mathsf{f}_5$ lives $?\nu_3$, $?\nu_8$ name $\langle P \rangle$, $?\nu_8$ related $?\mathsf{f}_5$, $?\mathsf{f}_5$ name $?\nu_9$, $?\mathsf{f}_5$ lives $?\mathsf{l}_5$ }

(c) Rewritten body of $\mathsf{Q_U}^{\text{part}}$

But then, $\cup_i \mathsf{V}'_{j_i}(\mathsf{G}) \subseteq \cup_i \mathsf{V}_{j_i}(\mathsf{G})$, and therefore, $\cup_i \mathsf{V}'_{j_i}(\mathsf{G}) \subseteq \mathcal{V}(\mathsf{G})$. Applying query q in both sides of the containment relation, we get $q(\cup_i \mathsf{V}'_{j_i}(\mathsf{G})) \subseteq q(\cup_i \mathsf{V}_{j_i}(\mathsf{G})) \subseteq q(\mathcal{V}(\mathsf{G}))$. By construction, query q' considers one possible way of evaluating query q over $\cup_i \mathsf{V}'_{j_i}(\mathsf{G})$, the one that considers the predicates $p_i(\bar{X}_i)$ in each $\mathsf{V}'_{j_i}(\mathsf{G})$ (remember that predicate $p_i(\bar{X}_i)$ appears in the head of V'_{j_i} due to $\Phi_{ij_ik_i}$). Thus, $q'(\mathsf{G}) \subseteq q(\cup_i \mathsf{V}'_{j_i}(\mathsf{G}))$, which implies that $q'(\mathsf{G}) \subseteq q(\mathcal{V}(\mathsf{G}))$.

To prove completeness, it suffices to show that $\mathsf{Q}(\mathcal{V}(\mathsf{G})) \subseteq \mathsf{Q}'(\mathsf{G})$. Consider $q(\bar{X})$:- $p_1(\bar{X}_1), \ldots, p_n(\bar{X}_n)$ and let $\mathcal{A}(p_i(\bar{X}_i), \mathcal{V}(\mathsf{G})) = \{\mathsf{T} = (\mathsf{s}, \mathsf{p}, \mathsf{o}) |\ \mathsf{T} \in \mathcal{V}(\mathsf{G})$ and there exists a valuation ϕ_i such that $\phi_i(p_i(\bar{X}_i)) = \mathsf{T}\}$, that is, $\mathcal{A}(p_i(\bar{X}_i), \mathcal{V}(\mathsf{G}))$ contains all the triples in $\mathcal{V}(\mathsf{G})$ satisfying $p_i(\bar{X}_i)$.

Now, consider a set of triples $\mathsf{T}_1, \ldots, \mathsf{T}_n$ such that (i) $\mathsf{T}_i \in \mathcal{A}(p_i(\bar{X}_i), \mathcal{V}(\mathsf{G}))$; and (ii) there exists a valuation ϕ (the composition of valuations ϕ_i, for each i) that maps the body $\mathsf{BD}(q)$ of query q to the set of T_i triples. If no such valuation exists for any set of triples, then $\mathsf{Q}(\mathcal{V}(\mathsf{G})) = \emptyset$ and completeness trivially holds. Now, each triple T_i is generated by a predicate t_{k_i} of some view V_{j_i}. This implies the existence of a valuation Ψ_{j_i} of view V_{j_i} over G such that $\Psi_{j_i}(t_{k_i}) = \mathsf{T}_i$. It is not hard to see that due to the existence of valuations ϕ_i between $p_i(\bar{X}_i)$ and T_i, and Ψ_{j_i} between V_{j_i} and T_i, Algorithm 1 constructs (in lines 9-17) a variable mapping $\Phi_{ij_ik_i}$ for V_{j_i}. We show that $\mathsf{T}_i \in (\Phi_{ij_ik_i}(\mathsf{V}_{j_i}))(G)$. This is proven by contradiction. If $\mathsf{T}_i \notin (\Phi_{ij_ik_i}(\mathsf{V}_{j_i}))(G)$, then variable mapping $\Phi_{ij_ik_i}$ assigns a variable v in V_{j_i} (and in particular in t_{k_i}) to a constant c (mappings between variables do not affect the evaluation of V_{j_i}), which causes the exclusion of T_i from the results (c is not one of the constants in T_i), a contradiction. Notice that in Algorithm 1 (lines 10-15), any binding of variables in $\Phi_{ij_ik_i}$ is using constants from query q (and therefore from T_i).

Consider now query q' constructed by considering all the $\Phi_{ij_ik_i}(\mathsf{V}_{j_i})$ corresponding to triples T_i. Query q' is constructed by Algorithm 1 since (a) the $\Phi_{ij_ik_i}(\mathsf{V}_{j_i})$ are compatible, due to the existence of valuation ϕ; and (b) Algorithm 1 is exhaustive and considers all possible combinations of variable mappings, and therefore, it will consider the above combination. It is not hard to see then that for the triples T_i, evaluating q over the triples is equivalent to evaluating $q'(\mathsf{G})$. That is, for any set of triples satisfying q, there exists a corresponding query q' that produces the same result. By considering (the union of) all possible sets of triples that satisfy q, we can infer that $q(\mathcal{V}(\mathsf{G})) \subseteq \cup q'(\mathsf{G})$, *i.e.*, $\mathsf{Q}(\mathcal{V}(\mathsf{G})) \subseteq \mathsf{Q}'(\mathsf{G})$. ■

The cost of Algorithm SQR is influenced by the cost of computing variable mappings $\mathcal{O}(|\mathsf{Q}| \times \sum_j |\mathsf{RD}(\mathsf{V}_j)|)$, but is dominated by the generation of rewritings and is thus equal

to $\mathcal{O}((\sum_j |\mathsf{V}_j|)^{|\mathsf{Q}|})$, where $|\mathsf{Q}|$ (resp. $|\mathsf{V}_j|$) is the size of Q (resp. V_j).

In SQR, as long as a view predicate is mentioned in a query, the view automatically becomes a candidate for rewriting the query (modulo an incompatibility check). The key reason is that the RDF model is *schema-less*. This schema-less nature of the data model is the main reason behind the exponential blow-up of the rewriting. As an example, using SQR to rewrite query Q_U over the views of Figure 3.1 results in a rewriting Q' that is a union of 64 queries, all of which must be evaluated in principle for the rewriting to be sound and complete. However, a number of these queries can either be (i) optimized and replaced by more *succinct* and equivalent queries; or (ii) dropped from consideration altogether because they result in an empty set. Going back to our motivating example, remember that actually, only 4 queries suffice for the rewriting. Therefore, the challenge we address next is to perform such optimizations without sacrificing soundness or completeness.

3.3 Optimizing Rewritings

In this section, we discuss a few optimization techniques that make the rewriting more efficient and practical.

3.3.1 Optimizing Individual Rewritings

In the rewriting of Q_U, each rewriting q' generated by Algorithm SQR joins four views (one view from the CandV of each of the four predicates vfriend, vlives, vrelated, vlives in Q_U). One such rewriting involves views V_F for vfriend, V_F for vlives, V_R for vrelated, and V_R for vlives. That is, the rewriting uses two copies of both V_F and V_R. Since the join (*e.g.,* vfriend joined with vlives) in Q_U is done in a similar way as that in the view V_F, there is redundancy to have two copies of V_F for this join; and similarly for V_R. The question is whether it is possible to get an *equivalent* rewriting by merging view copies, and thus generate a simpler query to evaluate. Indeed, one copy from each view suffices: the two copies of V_F are due to predicates vfriend and vlives being joined on variable $?\mathsf{f}_5$ in Q_U. However, in the CONSTRUCT of V_F, these two predicates are joined in a similar way. Therefore, one copy of V_F suffices since it already returns all the triples joinable by the two predicates (*i.e.,* the view self-join is *equivalent* to the view itself).

Algorithm 2 detects such situations by accepting as input two copies of a view V that are used in rewriting a query, one as the candidate view for predicate p_1 and the other for its joinable predicate p_2, with variable mappings Φ_1 and Φ_2, respectively. The algorithm considers the variable mappings between the query and the views and attempts to construct a new mapping Φ_{merge} that merges the two input mappings. If Φ_{merge} exists, the two copies

Algorithm 2: Candidate View Merging

Input: (V, Φ_1) from CandV_1, (V, Φ_2) from CandV_2
Output: $(\mathsf{V}, \Phi_{\text{merge}})$

```
1  Continue_merge = false;
2  for each triple pattern (s, p, o) in Φ1(RD(V)) do
3  |   Let (s', p, o') be the corresponding pattern in Φ2(RD(V))
4  |   if {s, o} ∩ {s', o'} ≠ ∅ then Continue_merge = true;
5  if Continue_merge == false then return (V, ∅);
6  for each triple pattern (s, p, o) in Φ1(RD(V)) do
7  |   Let (s', p, o') be the corresponding pattern in Φ2(RD(V))
8  |   Create corresponding merged pattern (sM, p, oM) for Φmerge
9  |   if s is a fresh variable then sM = s'; goto 14;
10 |   if s' is a fresh variable then sM = s; goto 14;
11 |   if s = s' then sM = s else return (V, ∅)
12 |   if o is a fresh variable then oM = o'; goto 8;
13 |   if o' is a fresh variable then oM = o; goto 8;
14 |   if o = o' then oM = o else return (V, ∅)
15 return (V, Φmerge)
```

of V can be merged to simplify the rewriting. During merging, should multiple occurrences of the same predicate appear in the same V, they are treated as distinct predicates. A key observation during the construction of Φ_{merge} is that *all the variables and constants appearing in the query are treated as constants* (thus, only fresh variables are treated as variables for the purpose of merging the view copies). This ensures that views are merged not only because they are copies of the same view, but also because their predicates are joined in precisely the same way as in the query (lines 4-7). Each time view copies are merged, we must also account for any variable mappings that have been applied to the views, due to their relationships with the views used for rewriting other predicates. Algorithm 2 ensures that the effects of such variable mappings are also merged (lines 8-16). If Φ_{merge} in the output of Algorithm 2 is $\emptyset$, the two copies of V cannot be merged.

To illustrate, consider in the rewriting of $\mathsf{Q_U}$ the var. mapping $(\mathsf{V_F}, \Phi_{111})$ for predicate vfriend and $(\mathsf{V_F}, \Phi_{213})$ for predicate vlives. Applying the two mapping functions respectively on $\mathsf{V_F}$ would result in two copies of $\mathsf{V_F}$ joined on $?\mathsf{f}_5$. Since in $\mathsf{V_F}$ the triple patterns of vfriend and vlives are joined in the same way as that in $\mathsf{Q_U}$, Φ_{111} and Φ_{213} can be merged; $\Phi_{\text{merge}}(?\nu_4, ?\mathsf{f}_5, ?\nu_0, ?\nu_1) = (\text{person0}, ?\mathsf{f}_5, ?\nu_5, ?\mathsf{l}_5)$. Therefore, the rewriting from Algorithm SQR involving two copies of $\mathsf{V_F}$ can be simplified into a rewriting with one copy.

Theorem 2 *Query q'_{merge} resulting from (i) replacing the two copies of view V in query q' with one; and (ii) applying Φ_{merge} computed by Algorithm 2, in place of Φ_1 and Φ_2; is*

equivalent to q'.

Proof. Without loss of generality, suppose user query $\mathsf{Q_U}$ is composed of two triple patterns $p_1(\bar{X}_1)$ and $p_2(\bar{X}_2)$, and one rewriting q from Algorithm 1 is: $\mathsf{HD}(q) = p_1(\bar{X}_1), p_2(\bar{X}_2)$ and $\mathsf{BD}(q) = \mathsf{BD}(\Phi_1(\mathsf{V})) \wedge \mathsf{BD}(\Phi_2(\mathsf{V}))$. Recall that Algorithm 2 tries to simplify q by removing redundant triple patterns in q. If Φ_{merge} returned from Algorithm 2 is *null*, the rewriting q is retained (thus, the soundness and completeness of Algorithm 1 is not affected); otherwise, q is replaced with a simplified rewriting q': $\mathsf{HD}(q') = p_1(\bar{X}_1), p_2(\bar{X}_2)$ and $\mathsf{BD}(q') = \mathsf{BD}(\Phi_{merge}(\mathsf{V}))$. Since both $\Phi_1(\mathsf{V})$ and $\Phi_2(\mathsf{V})$ are conjunctive queries and are joined (lines $4 \sim 6$ of Algorithm 2), q is also a conjunctive query. Note that only fresh variables in q and q' are treated as variables when constructing the variable mapping function Φ_{merge}; constants and variables appearing in the head of q are treated as constants.

To prove that q' is equivalent to q, we first prove q' is contained in q, *i.e.,* $q' \subseteq q$. The way we construct Φ_{merge} (lines $11 \sim 16$) guarantees that there is a homomorphism φ from q to q': $\varphi(s) = s_M$, $\varphi(s') = s_M$, $\varphi(o) = o_M$, and $\varphi(o') = o_M$, as s and s' (correspondingly, o and o') are either fresh variables or identical to s_M (correspondingly, o_M). Since $t(s, o)$ and $t(s', o')$ have the same predicate as $t(s_M, o_M)$, it is easy to see that $\varphi(q) = q'$. Therefore, q' is contained in q, *i.e.,* $q' \subseteq q$.

Next, we prove that $q \subseteq q'$. For any triple pattern $t(s_M, o_M)$ in q', in which both s_M and o_M are not fresh variables (lines 13 and 16), at least one of the triple patterns $t(s, o)$ and $t(s', o')$ is identical to $t(s_M, o_M)$. For other triple patterns in q' that involve fresh variables, there must exist (at least one) isomorphism between these triple patterns and the corresponding triple patterns in q. Therefore, q' is equivalent to a subquery of q. Thus, we have $q \subseteq q'$.

As both $q' \subseteq q$ and $q \subseteq q'$ hold, we have proved that q is equivalent to q'. ■

The cost of Algorithm 2 is $\mathcal{O}(|\mathsf{V}|)$. Since, in the worst case, there can be as many view copies of a view V as the size of the query, optimizing with Algorithm 2 each conjunctive query generated at lines 22-23 of SQR costs $\mathcal{O}(|\mathsf{Q}| \times |\mathsf{V}|)$.

3.3.2 Pruning Rewritings with Empty Results

Due to the schema-less nature of RDF, a sound and complete rewriting of an input query *requires* that we construct rewritings by considering every possible combination of predicates from the input views, which often results in a certain number of rewritings with empty results. (This observation is unique to RDF/SPARQL, in comparison to the query rewriting results in the relational or XML case.) For example, a sound and complete rewriting of query

$\mathsf{Q_U}^{\text{part}}$ (see Section 3.2.1) includes the rewriting q' in Table 3.1(c). Rewriting q' joins triples from $\mathsf{V_{FoF}}$ and $\mathsf{V_R}$ and essentially looks for persons that are relatives of friends-of-friends of person0. Looking at the triples in Figure 3.1, it is clear that no current base triples satisfy the constraints of q'. The question is then how can we detect such empty rewritings, and more importantly, how to do this efficiently.

Consider a simple case where a rewriting involves a join between two predicates $(?y_1, p_1, ?y_2)$ and $(?y_3, p_2, ?y_4)$, where the join equates $?y_2$ and $?y_3$. Denote the value set of a variable $?x$ as $A(?x)$. If we store $A(?x)$ for every variable in any triple pattern, this problem is trivial, *i.e.,* we simply check whether $A(?y_2) \bigcap A(?y_3) = \emptyset$. Unfortunately, this straightforward solution is expensive space-wise. In general, a negative result exists for the *boolean set intersection* problem, *i.e.,* given two sets A_1 and A_2, checking if A_1 and A_2 intersects requires linear space, even if one is willing to settle to a constant success probability [18, 62]. However, we can design a space-efficient heuristic that works well in practice.

The basic idea is to first determine the value set for each distinct variable involved in the rewriting, and then construct a *synopsis* for each value set. In our example, we can estimate the *size of intersection* of $A(?y_2)$ and $A(?y_3)$ based on their synopses. If the intersection size is estimated to be above some preset threshold with a reasonable probability, we consider the predicates as joinable; otherwise, we issue an ASK query to verify if the join is actually empty; if it is, we remove this and other rewritings involving predicates $(?y_1, p_1, ?y_2)$ and $(?y_3, p_2, ?y_4)$. Note that our pruning step does not affect the soundness and completeness of our solution, as before pruning, we always issue an ASK query to make sure that rewriting has an empty result. In general, an ASK query is much cheaper than the corresponding SELECT query, especially when the graph pattern is nonselective, and the synopses are used to avoid issuing unnecessary ASK queries (for those rewritings that are very likely to be nonempty).

The synopses should satisfy two key requirements. First, we should be able to estimate the size of intersection of multiple value sets (not just binary intersection) since a rewriting might include a join of m predicates on m variables. Let $?x_1, ?x_2, \ldots, ?x_m$ denote these variables. To simplify notation, we use A_i to denote $A(?x_i)$. Second, the synopses of each variable should be able to estimate the distinct elements in its value set (as well as support distinct elements estimation under the set intersection operator). This requirement comes from the observation that we can estimate the size of an intersection $|A_1 \bigcap A_2|$ by simply estimating the size of $D(A_1 \bigcap A_2)$ where D is the number of distinct elements in A_1 and A_2, respectively. In what follows, we show that the *KMV-synopsis*[25] meets both requirements.

For a set A_1, we denote its KMV-synopsis as $\sigma(A_1)$. The construction of $\sigma(A_1)$ is as follows. Given a collision-resistant hash function h that generates (roughly) uniformly random hash values in its domain $[1, M]$, $\sigma(A_1)$ simply keeps the k smallest hash values from all elements in A_1, *i.e.,* $\sigma(A_1) = \{h(v_1), \ldots, h(v_k)\}$, where $v_i \in A_1$, and $h(v) \geq \max(\sigma(A_1))$ if $v \in A_1$ and $h(v) \notin \sigma(A_1)$. Then, $\widehat{D}(A_1) = \frac{k-1}{\max(\sigma(A_1))/M}$ is an unbiased estimator for $D(A_1)$ [25]. Furthermore, it is also possible to estimate the distinct number of elements in a general compound set (produced based on $A_1, \ldots, A_m$ with set union, intersection, and difference operators) [25]. In our case, we are only interested in estimating $D(I)$ where $I = A_1 \bigcap A_2 \cdots \bigcap A_m$. Specifically, inspired by the discussion in [25], we can obtain an unbiased estimator $\widehat{D}(I)$ as follows. Define $\sigma(A_i) \oplus \sigma(A_j)$ as the set consisting of the k smallest values in $\sigma(A_i) \bigcup \sigma(A_j)$, and let $\sigma_{1\ldots m} = \sigma(A_1) \oplus \sigma(A_2) \cdots \oplus \sigma(A_m)$. Furthermore, let:

$$\begin{aligned} K_I &= \left| \sigma_{1\ldots m} \bigcap \sigma(A_1) \bigcap \cdots \bigcap \sigma(A_m) \right| \text{ and,} \\ \widehat{D}(I) &= \frac{K_I}{k} \left(\frac{k-1}{\max(\sigma_{1\ldots m})/M} \right). \end{aligned} \tag{3.1}$$

We can show that , by extending similar arguments from [25]:

Lemma 1 *For $k > 1$, $\widehat{D}(I)$ in Equation 3.1 is an unbiased estimator for $D(I)$.*

Proof. First, by Theorem 5 in [25] and discussion therein, we know that $\sigma_{1\ldots m}$ is the size-k KMV-synopsis for the union of sets $A_1, A_2, \ldots, A_m$, *i.e.,* $\sigma_{1\ldots m} = \sigma(A_1 \bigcup A_2 \cdots \bigcup A_m) = \sigma(U)$, where $U = A_1 \bigcup A_2 \cdots \bigcup A_m$. Hence, $\frac{k-1}{\max(\sigma_{1\ldots m})/M}$ is an unbiased estimator for $D(U)$ by the basic property of the KMV-synopsis (see Section 4.1 in [25]).

Next, we simply extend the similar arguments from Section 5.2 in [25] for the intersection over two sets to the intersection over multiple sets. Let V_U be the set of k values from the original value sets A_1 to A_m that correspond to the hash values in $\sigma_{1\ldots m}$, then, by a trivial extension of Lemma 1 in [25], we can show that for each $v \in V_U$, we have $v \in A_i$ if and only if $h(v) \in \sigma(A_i)$. This implies that $v \in I$, where $I = A_1 \bigcap A_2 \cdots \bigcap A_m$, if and only if $h(v) \in \sigma(A_1) \bigcap \sigma(A_2) \cdots \bigcap \sigma(A_m)$. Hence:

$$K_\cap = |v \in V_U : v \in A_1 \bigcap A_2 \cdots \bigcap A_m| = K_I, \tag{3.2}$$

where K_I is given by Equation 3.1.

Clearly, V_U could be viewed as a random sample of k elements from $\mathcal{D}(U)$, where $\mathcal{D}(\cdot)$ represents the set of distinct elements from an input. Hence, $\rho = D(I)/D(U)$ can be estimated by $\widehat{\rho} = K_\cap/k$, the fraction of sample elements in V_U that also belong to $\mathcal{D}(I)$. As a result, $\widehat{\rho}$ gives an unbiased estimator for ρ.

Finally, since $D(I) = \rho D(U)$, then $\mathrm{E}(\widehat{\rho} \cdot \widehat{D}(U)) = D(I)$ if $\widehat{D}(U)$ is an unbiased estimator of $D(U)$. Since $\widehat{\rho} = K_\cap/k = K_I/k$ (by Equation 3.2), it follows that $\widehat{D}(U) = \frac{k-1}{\max(s_{1\ldots m})/M}$ is an unbiased estimator of $D(U)$ (from the first paragraph in this Section). We complete the proof. ■

Lemma 2 *If $D(I) > 0, \epsilon \in (0,1)$ and $k \geq 1$, let $T = kD(I)/j$, it follows:*

$$\Pr\left(\frac{|\widehat{D}(I) - D(I)|}{D(I)} \leq \epsilon \Big| K_I = j\right) = \Delta(kD(I)/j, k, \epsilon) = \delta, \tag{3.3}$$

$$\begin{aligned}\Delta(T,k,\epsilon) = &\sum_{i=k}^{T} \binom{T}{i} \left(\frac{k-1}{(1-\epsilon)T}\right)^i \left(1 - \frac{k-1}{(1-\epsilon)T}\right)^{T-i} \\ &- \sum_{i=k}^{T} \binom{T}{i} \left(\frac{k-1}{(1+\epsilon)T}\right)^i \left(1 - \frac{k-1}{(1+\epsilon)T}\right)^{T-i}\end{aligned}$$

Proof. The key observation is that for the intersection over multiple sets (more than 2), the following arguments made for the intersection over two sets in Section 5.2 from [25] still hold. V_U is a uniform random sample of size k drawn from $\mathcal{D}(U)$, and $K_\cap$ is a random variable representing the number of elements in V_U that also belongs to $\mathcal{D}(I)$. Hence, similar to Equation 9 in [25], for intersection over m sets, we also have:

$$\Pr(K_\cap = j) = \binom{D(I)}{j}\binom{D(U) - D(I)}{k-j} \Big/ \binom{D(U)}{k}, \tag{3.4}$$

where $K_\cap$ is given in Equation 3.2. Hence, $K_\cap$ is a random variable with a hypergeometric distribution, just as the same random variable defined for the intersection over two sets in Section 5.2 from [25]. The rest of the proof for $\Pr\left(\frac{|\widehat{D}(I)-D(I)|}{D(I)} \leq \epsilon \Big| K_I = j\right)$ given in Equation 3.3 of Lemma 2 follows exactly the same fashion as the proof of Theorem 6 in [25]. We omit the detailed derivations (simple substitution of parameters for several functions defined in [25] in the proof of Theorem 6) that eventually lead to our Equation 2. This completes the proof. ■

In practice, given the observation of $\widehat{D}(I)$ and K_I, we can set $T = k\widehat{D}(I)/K_I$ and substitute T in Equation 3.3. Thus, we can obtain the confidence value δ for $\Pr\left(\frac{|\widehat{D}(I)-D(I)|}{D(I)} \leq \epsilon\right)$ for any error value ϵ. That said, our pruning technique works as follows.

We preset a small threshold value $\tau > 1$, a probability threshold $\theta \in [0,1)$, and a relative error value $\epsilon \in (0,1)$. For any m value sets of m variables to be joined in a rewriting, we estimate their intersection size as $\widehat{D}(I)$ by Equation 3.1, and $\delta = \Pr\left(\frac{|\widehat{D}(I)-D(I)|}{D(I)} \leq \epsilon\right)$ as above. Then, we check if $\widehat{D}(I)/(1+\epsilon) > \tau$ and $\delta > \theta$ (*i.e.,* if $D(I)$ is larger than τ with a probability $\geq \theta$). If this check returns false, we issue an ASK query to verify if the

corresponding rewriting is empty; if yes, we can safely prune this rewriting. Otherwise (either the check returns true or the ASK returns nonempty), we consider that I is not empty and keep the current rewriting. In practice, we observe that the above procedure can be simplified by just checking if $\widehat{D}(I) \leq \tau$ for a small threshold value $\tau > 1$ (without using δ, θ, and ϵ), which performs almost equally well.

To illustrate, consider again the rewriting in Table 3.1(c) of query $\mathsf{Q_U}^{\text{part}}$. To detect whether the rewriting is empty, we estimate the intersection size of the join in Table 3.1(c) using Equation 3.1. For the example, the equation indicates that the intersection is not larger than τ, and therefore, we issue an ASK query. The ASK query evaluates the rewriting of Table 3.1(c) over the triples of Figure 3.1(a). Since there are no triples for persons that are relatives of friends-of-friends of person0, the ASK query returns false. Thus, $\mathsf{Q_U}^{\text{part}}$ is pruned.

The KMV-synopsis supports insertions (of a new item to the multiset from which the synopsis was initially built) but not deletions (hence, it does not support the general update, which can be modeled as a deletion followed by an insertion) [25]. However, we can still use the KMV-synopsis to provide a quick estimation for pruning rewritings with empty results in case of updates to RDF stores, by only updating the synopses with the insertions and ignoring the deletions. Clearly, over time, this will lead to an overestimation of the intersection size for multiple sets. However, such an overestimation only gives us false positives but not false negatives, *i.e.,* we will not mistakenly prune any rewritings that do not produce an empty result. Of course, as the number of deletions increases, this approach will lead to too many false positives (rewritings that do produce empty results cannot be detected by checking their synopses) Hence, we can periodically rebuild all synopses after seeing enough number of deletions w.r.t. a user-defined threshold.

3.3.3 Optimizing the Generation of Rewritings

The pruning technique presented in Section 3.3.2 considers rewritings in isolation, to decide if a rewriting is empty or not. One way to integrate Algorithm SQR with the pruning technique will be: generating all the possible rewritings *in one shot* followed by a pruning step to remove empty rewritings from evaluation. However, such an integration ignores some inherent relationships between the rewritings, *i.e.,* that different rewritings share similar subqueries. If we can quickly determine a common subquery (*i.e.,* partial rewriting) is empty, it will save time that otherwise is needed to determine whether the rewritings contained in this subquery are empty or not. In what follows, we show how one can optimize the rewriting by taking advantage of these common subqueries. To illustrate, consider our

running example and the rewriting of $\mathsf{Q_U}$ over the views in Figure 3.1(b). One generated rewriting q_1' for $\mathsf{Q_U}$ involves views $\mathsf{V_F}$, $\mathsf{V_R}$, $\mathsf{V_R}$, $\mathsf{V_R}$ with appropriate variable mappings since each view is in the CandV of predicate friend, lives, related, and lives, respectively. Similarly, another generated rewriting q_2' involves views $\mathsf{V_F}$, $\mathsf{V_R}$, $\mathsf{V_{RoR}}$, $\mathsf{V_R}$. The key observations here are that (i) both rewritings involve a join of views $\mathsf{V_F}$ and $\mathsf{V_R}$; and (ii) from the optimization of the previous section, the join of views $\mathsf{V_F}$ and $\mathsf{V_R}$ is *empty* since the set of friends of "Eric" (see Figure 3.1(c)) is disjoint from his relatives. Therefore, both rewritings q_1' and q_2' can safely be removed (and every other rewriting involving a join of the two views over the corresponding predicates). By detecting with a single check the empty join between views $\mathsf{V_F}$ and $\mathsf{V_R}$, the algorithm optimized SQR (OSQR, see Algorithm 3) terminates immediately the *branch* of rewritings (including q_1' and q_2') involving these two views. To remove them from consideration, Algorithm SQR must check each generated individual rewriting independently. Algorithm OSQR addresses this shortcoming by building individual rewritings in a step-wise fashion. This way, OSQR *detects* and *terminates* early any *branch* of rewritings involving views whose join result is empty.

In a nutshell, Algorithm OSQR works as follows. The algorithm uses a structure STACK where each element in STACK stores a subquery SubQ of Q along with a candidate view combination for rewriting SubQ. Initially, STACK and SubQ are empty. The first subquery considered corresponds to a triple pattern in Q, and we pick the pattern with the smallest size of |CandV| (*i.e.,* the number of views in CandV). Intuitively, this triple pattern is the most *selective* and by considering the most selective predicates in order (in terms of their |CandV|), we maximize the effects of early terminating a branch of rewritings once we detect the rewriting for SubQ results in an empty set (a larger portion of the rewritings for Q that contain this rewriting for SubQ is pruned earlier in this manner). After the first pattern, the algorithm considers one pattern added at each step. The way the pattern is picked (line 14) ensures that it can be joined with the current SubQ at the head of STACK, which increases the chance of optimization with techniques described in Section 3.3.1 and Section 3.3.2. Again, when more than one pattern is under consideration, the most selective one is picked. After a pattern is added and a candidate view for the pattern is picked, the view redundant with the existing view set for SubQ will be merged into the view set (lines 18-19). If the current rewriting for SubQ has an empty result (lines 21-23), the rewriting is not extended further and not pushed back into STACK.

We use $\mathsf{CandV_1}$ and $\mathsf{CandV_2}$ in Tables 3.1(a) and 3.1(b) to illustrate OSQR. Since $|\mathsf{CandV_1}|$ is smaller in size (line 6) , it first initializes STACK = $\{(\{\mathsf{vfriend}\},\{\mathsf{V_F}\}), (\{\mathsf{vfriend}\},\{\mathsf{V_{FoF}}\})\}$

Algorithm 3: The Optimized SQR (OSQR) Algorithm

Input: Views $\mathcal{V}$, query Q with GP(Q)=(s_1^Q, p_1^Q, o_1^Q), ..., (s_n^Q, p_n^Q, o_n^Q)
Output: a rewriting Q′ as a union of conjunctive queries

```
1  Set the query rewriting result Q′ to ∅.
2  Generate CandV_i for each triple pattern (s_i^Q, p_i^Q, o_i^Q), 1 ≤ i ≤ n.
3  Set SubQ to ∅; initialize a stack STACK to store view combinations for SubQ.
4  Pick a triple pattern (s_i^Q, p_i^Q, o_i^Q), with the smallest size of |CandV_i|.
5  Add (s_i^Q, p_i^Q, o_i^Q) into SubQ;
6  push each combination (SubQ, {V, V ∈ CandV_i}) into STACK.
7  while STACK is non-empty do
8      Pop a combination R from STACK; extract SubQ from R.
9      if SubQ contains all triple patterns in user query then
10         Generate a rewriting q from R's view set (lines 21-23 in SQR).
11         Q′ = Q′ ∪ q; goto line 9.
12     Get all triple patterns that can be joined with SubQ but not in SubQ;
13     Pick the triple pattern (s_j^Q, p_j^Q, o_j^Q) with the smallest size of |CandV_j|.
14     for each view V in CandV_j do
15         Create a copy R′ of R and a copy SubQ′ of SubQ.
16         if V is redundant with existing views in R′ then
17             Merge V with the view set of R′ (Sec. 3.3.1).
18         else Add V into the view set of R′.
19         if the estimated result of a rewriting from R′ is empty (Sec. 3.3.2) then
20             Issue an ASK query corresponding to the rewriting.
21             if ASK query confirms the result is empty then goto line 16.
22         Add (s_j^Q, p_j^Q, o_j^Q) in SubQ′ to replace SubQ in R′;
23         Push R′ in STACK.
```

(line 8). OSQR processes CandV_2 next (line 15). It iterates through CandV_2 from $(\mathsf{V}_\mathrm{R}, \Phi_{233})$ and detects that V_F in CandV_1 cannot be merged with V_R in CandV_2 (line 18). Therefore, OSQR adds (vlives,V_R) to R′ (line 20). Assume OSQR detects an empty result (line 21), (*e.g.,* the join of V_F and V_R for "Eric" is actually empty); OSQR issues an ASK query. If ASK returns negative (*i.e.,* empty), OSQR will skip lines 24-25 to avoid pushing ({vfriend, vlives},{V_F, V_R}) into STACK. The above procedure iterates until STACK is empty.

3.4 Experiments

We implemented our rewriting algorithms and optimization components in C++ and evaluated them on two RDF stores, namely, 4store [1] and Jena TDB [4]. Our relational database experiments were conducted using MySQL. For KMV synopsis, we set k=16 and $\tau = 2$ whenever the synopses were used (the simplified version of the checking procedure from Section 3.3.2 was adopted).

Here, we report the experimental results that compare the basic SPARQL query rewriting (SQR) algorithm with the optimized SQR (OSQR) algorithm, with detailed evaluation of the impact of individual optimization components. We used two key performance metrics, *i.e.,* the number of rewritings generated through query rewriting and the end-to-end evaluation time, including query rewriting and execution. Also, we studied the scalability of our algorithms along multiple dimensions, *i.e.,* the size of query $|\mathsf{Q}|$, views $|\mathcal{V}|$, and $|\mathsf{CandV}|$. In experiments, we used the popular RDF benchmark LUBM [50] (which considers a setting in the university domain that involve students, departments, professors, etc.) to generate a dataset of 10M triples as the base data, over which views are defined using SPARQL queries. We ran all experiments on a 64-bit Linux machine with a 2GHz Intel Xeon(R) CPU and 4GB of memory.

3.4.1 Experimental Results with 4Store

In the introduction, we claim that translating SPARQL queries/views to SQL does not resolve the challenges addressed by our work. Here, we illustrate experimentally that this is indeed the case. For the experiment, we use the setup shown in Figure 3.3. In more detail, we use the seven view templates to instantiate 56 different views. Specifically, we create 14 views using template V_1 (each view with a different parameter in P_1), 12 views using template V_2 (using the same first 12 of the 14 parameters used for V_1), 10 views using template V_3 (using the same first 10 of the parameters used for V_1 and V_2), 8 views using template V_4 (using the same first 8 of the parameters used for V_1, V_2, and V_3), 6 views using template V_5 (using the same first 6 of the parameters used for V_1, V_2, V_3, and V_4), 4 views using template V_6 (using the same first 4 of the parameters used for V_1, V_2, V_3, V_4, and V_5), and 2 views using template V_7 (using the same first 2 of the parameters used for all the other views). Each view exposes some aspect of a student's data (*e.g.,* name, email). In terms of the query, we execute a different query in each iteration of the experiment. In iteration i, the query involves all the predicates in Figure 3.3(b) with an annotation $j \leq i$. So, the query initially has 3 predicates, and in each iteration we add one more predicate, up to a size of 7. Given the above setup, it is not hard to see that (i) the CandV for predicate name has 14 views, that for predicate email, it has 12, and finally for predicate worksFor, it has only 2 views; and (ii) for any two predicates p_i and p_j, there are $min(|\mathsf{CandV}_i|, |\mathsf{CandV}_j|)$ nonempty joins between the two candidate views.

We also translate the SPARQL queries/views and the underlying RDF data to SQL and relational data. For the relational representation of RDF data, we use (fully-indexed) predicate tables [12], which provide one of the most efficient representations of RDF in

V_1:CONSTRUCT { ?x_1 name ?n_1 } WHERE { ?x_1 name ?n_1, ?x_1 worksFor $\langle P_1 \rangle$}
V_2:CONSTRUCT { ?x_2 email ?e_2 } WHERE { ?x_2 email ?e_2, ?x_2 worksFor $\langle P_2 \rangle$}
V_3:CONSTRUCT { ?x_3 degreeFrom ?d_3 } WHERE { ?x_3 degreeFrom ?d_3, ?x_3 worksFor $\langle P_3 \rangle$}
V_4:CONSTRUCT { ?x_4 phone ?p_4 } WHERE { ?x_4 phone ?p_4, ?x_4 worksFor $\langle P_4 \rangle$}
V_5:CONSTRUCT { ?x_5 teacherOf ?c_5 } WHERE { ?x_5 teacherOf ?c_5, ?x_5 worksFor $\langle P_5 \rangle$}
V_6:CONSTRUCT { ?x_6 interest ?i_6 } WHERE { ?x_6 teacherOf ?i_6, ?x_6 worksFor $\langle P_6 \rangle$}
V_7:CONSTRUCT { ?x_7 worksFor ?w_7 } WHERE { ?x_7 worksFor ?w_7, ?x_7 worksFor $\langle P_7 \rangle$}

(a)

Q:SELECT { ① ?x, ① ?n, ① ?e, ① ?d, ② ?p, ③ ?c, ④ ?i, ⑤ ?w }
WHERE { ① ?x name ?n, ① ?x email ?e, ① ?x degreeFrom ?d,
② ?x phone ?p, ③ ?x teacherOf ?c, ④ ?x interest ?i, ⑤ ?x worksFor ?w }

(b)

Figure 3.3: Experimental setup 1 (a) Views templates and (b) Query template

terms of query performance. Then, we compare algorithms SQR and OSQR as well as the corresponding relational/SQL-based representation (denoted as SQL in our figures). Figure 3.4 shows the comparison results. As the size of the input query increases, Algorithm OSQR results in between *one and four orders of magnitude* less queries as part of the rewriting process, while both algorithms SQR and the SQL view expansion result in the same number of queries. Meanwhile, Algorithm OSQR is up to *two orders of magnitude* faster than both SQR and SQL, in terms of the evaluation times for query rewriting and execution.

To illustrate that the above result holds for different queries and views, we perform the same experiment with an alternative setup. In this setting, a query has three predicates and retrieves the email, degree, and all the courses taken by each student (see Figure 3.5(b)). The query is evaluated over views that have one of five view templates, denoted by V_i, $1 \leq i \leq 5$ (shown in Figure 3.5(a)). The templates are defined so that $\text{CandV}_{\text{courses}} = \{V_1, V_2, V_3, V_4, V_5\}$, $\text{CandV}_{\text{degree}} = \{V_3\}$, and $\text{CandV}_{\text{email}} = \{V_1, V_4, V_5\}$. Notice that if each template is instantiated only once, SQR results in 15 rewritings. Normally, one expects that only a few of the rewritings are nonempty and hence we make 2 of the 15 rewritings nonempty, those involving templates V_3 and V_4. To do this, we make sure that the same variable P_3 is used for both view templates V_3 and V_4 and thus both templates are instantiated from the same university. Notice that definition-wise, view templates V_1, V_4, and V_5 are identical. However, we make sure that the three templates are instantiated from different universities so that they are nonoverlapping in their contents. We create multiple instances of view templates using students from different departments, and by always populating pairs of instances of templates V_3 and V_4 from the same department, we make sure they join. Figure 3.6 shows the number of rewritings and evaluation times for SQR, OSQR, and

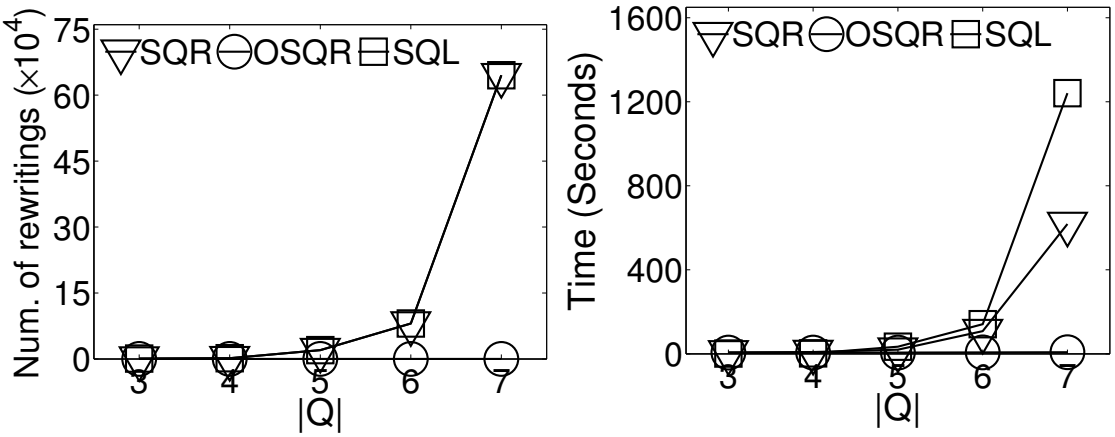


Figure 3.4: SPARQL rewriting vs. SQL expansion (a) Rewritten queries over query size and (b) Eval. time over query size

V_1:CONSTRUCT { ?x_1 email ?e_1, ?x_1 course ?c_1 }
WHERE { ?x_1 email ?e_1, ?x_1 course ?c_1, ?x_1 member ?u_1, ?u_1 subOrg $\langle P_1 \rangle$}
V_2:CONSTRUCT { ?x_2 phone ?p_2, ?x_2 course ?c_2 }
WHERE { ?x_2 phone ?p_2, ?x_2 course ?c_2, ?x_2 member ?u_2, ?u_2 subOrg $\langle P_2 \rangle$}
V_3:CONSTRUCT { ?x_3 degree ?d_3, ?x_3 course ?c_3 }
WHERE { ?x_3 degree ?p_3, ?x_3 course ?c_3, ?x_3 member ?u_3, ?u_3 subOrg $\langle P_3 \rangle$}
V_4:CONSTRUCT { ?x_4 email ?e_4, ?x_4 course ?c_4 }
WHERE { ?x_4 email ?e_4, ?x_4 course ?c_4, ?x_4 member ?u_4, ?u_4 subOrg $\langle P_3 \rangle$}
V_5:CONSTRUCT { ?x_5 email ?e_5, ?x_5 course ?c_5 }
WHERE { ?x_5 email ?e_5, ?x_5 course ?c_5, ?x_5 member ?u_5, ?u_5 subOrg $\langle P_5 \rangle$}

(a)

Q:SELECT { ?x, ?e, ?c, ?d } WHERE { ?x email ?e, ?x course ?c, ?x degreeFrom ?d }

(b)

Figure 3.5: Experimental setup 2 (a) Views templates and (b) Query template

the corresponding relational/SQL setting. In the experiment, we start by instantiating each template twice (10 views in total), and proceed by picking a template and adding view instances in a way that linearly increases the cardinality of $\mathsf{CandV}_{\mathsf{courses}}$ (the largest CandV set). Figure 3.6 shows that as the size of the largest CandV set increases, OSQR generates up to *an order of magnitude* less rewritings than SQR and the SQL view expansion, resulting in up to *an order of magnitude* savings in evaluation times.

In Section 3.3, we introduced three orthogonal optimizations and in algorithm OSQR, we incorporated all of them into a single algorithm. It is interesting to see what are the effects of each optimization in isolation, to the size of the rewriting and the evaluation time of the rewritten query. In the next three experiments, we investigate exactly this, starting here with an experiment that studies the effects of optimizing individual rewritings

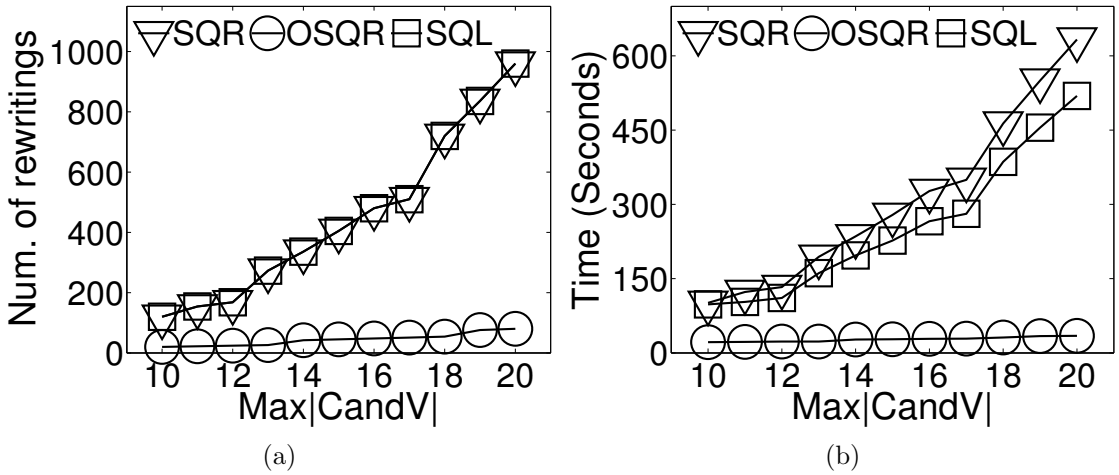


Figure 3.6: SPARQL rewriting vs. SQL expansion (a) Rewritten queries over max CandV and (b) Eval. time over max CandV

(presented in Section 3.3.1). To this end, we *switch off* in OSQR *all* other optimizations but merging views (denoted as OSQR-M) and compare it with SQR. In terms of the experimental setup, this is shown in Figure 3.7. We define 6 views over our base data, with each view exposing some aspect of a student's data (*e.g.,* email, phone). As for the queries, we execute 6 different queries, with each query increasingly bringing together data from the views. The return values and predicates of the query executed in iteration i are marked appropriately in Figure 3.7(b). Figure 3.8 shows the results of the comparison between SQR and OSQR-M, as the input query size increases. Figure 3.8(a) shows that both algorithms result in the same number of rewritings; note that merging does not influence the number of generated rewritings (this is the focus of the other optimizations). Merging optimizes each individual rewriting, and this becomes apparent in the evaluation time of the rewritings (see Figure 3.8(b)). As the size of query $|Q|$ increases, so is the potential for merging views (the same view might appear in the candidate view set of more predicates), which is confirmed in Figure 3.8(b) — savings in evaluation time of OSQR-M, compared to SQR, start from 10% to 70% for queries with 2 to 5 predicates. As $|Q|$ increases, so is the size of each rewriting (since the rewriting ultimately integrates the where clauses of candidate views). In our experiments, when a (rewritten) query has approximately 16 predicates, the engine of 4store crashes; therefore, it is impossible to execute a rewriting from SQR when $|Q| \geq 6$. Since merging results in smaller rewritings, OSQR-M can handle larger input queries.

As before, we switch off in OSQR all other optimizations but pruning empty rewritings (denoted as OSQR-P) and compare it with SQR. The experimental setup used here is shown in Figure 3.9. Using the view template in Figure 3.9(a), we generate 10 views, where

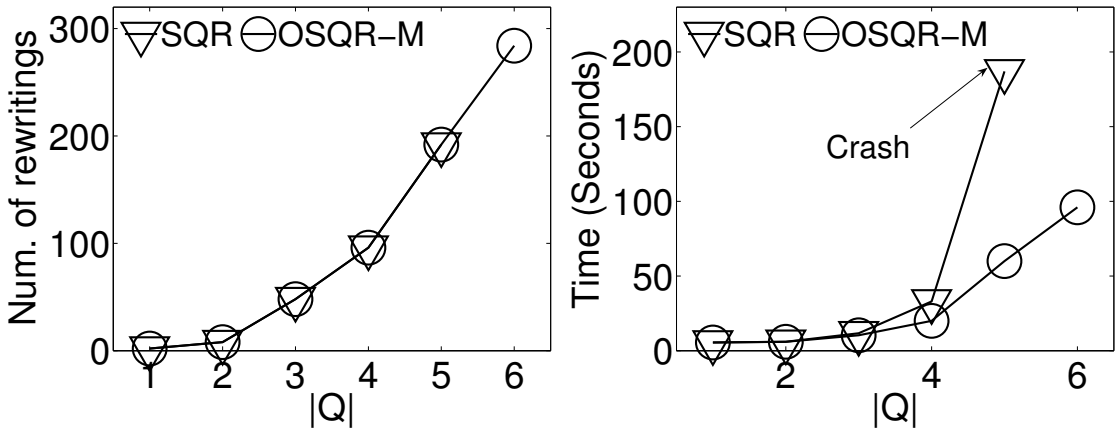
V_1:CONSTRUCT { ?x_1 name ?n_1, ?x_1 email ?e_1, ?x_1 takes ?c_1 }
WHERE { ?x_1 name ?n_1, ?x_1 email ?e_1, ?x_1 takes ?c_1 }
V_2:CONSTRUCT { ?x_2 phone ?p_2, ?x_2 course ?c_2, ?x_2 member ?u_2 }
WHERE { ?x_2 phone ?p_2, ?x_2 course ?c_2, ?x_2 member ?d_2 }
V_3:CONSTRUCT { ?x_3 phone ?p_3, ?x_3 course ?c_3, ?x_3 degree ?d_3 }
WHERE { ?x_3 phone ?p_3, ?x_3 course ?c_3, ?x_3 degree ?d_3 }
V_4:CONSTRUCT { ?x_4 name ?n_4, ?x_4 email ?e_4, ?x_4 takes ?c_4 }
WHERE { ?x_4 name ?n_4, ?x_4 email ?e_4, ?x_4 takes ?c_4 }
V_5:CONSTRUCT { ?x_5 phone ?p_5, ?x_5 course ?c_5, ?x_5 member ?u_5 }
WHERE { ?x_5 phone ?p_5, ?x_5 course ?c_5, ?x_5 member ?u_5 }
V_6:CONSTRUCT { ?x_6 phone ?p_6, ?x_6 course ?c_6, ?x_6 degree ?d_6 }
WHERE { ?x_6 phone ?p_6, ?x_6 course ?c_6, ?x_6 degree ?d_6 }

(a)

Q:SELECT { ?x, ① ?e, ② ?p, ③ ?c, ④ ?n, ⑤ ?u, ⑥ ?u′ }
WHERE { ① ?x email ?e, ② ?x phone ?p, ③ ?x takes ?c,
④ ?x name ?n, ⑤ ?x member ?u, ⑥ ?x degree ?u′ }

(b)

Figure 3.7: Experimental setup 3 (a) Views templates and (b) Query template



(a) (b)

Figure 3.8: Optimizing individual rewritings (a) Rewritten queries over query size and (b) Eval. time over query size

each view has a different value for the variable $\langle P \rangle$. Our instantiation is such that we use ten different departments from the same university as the values for variable $\langle P \rangle$. In this manner, we make sure that the views are nonoverlapping. The experiment has 8 iterations. The same query Q (shown in Figure 3.9(b)) is evaluated across all iterations over a set of $i+2$ views at iteration i. Notice that the CONSTRUCT statements of all views are identical to the graph pattern of the Q. It is not hard to see that for SQR, the CandV for each predicate of Q (name, email, course) contains all the views. Therefore, SQR will create $(i+2)^3$ rewritings

```
V:CONSTRUCT {?x1 name ?n1,
    ?x1 email ?e1,?x1 course ?c1 }
  WHERE { ?x1 name ?n1, ?x1 course ?c1,
    ?x1 email ?e1, ?x1 member ⟨P⟩ }
```

(a)

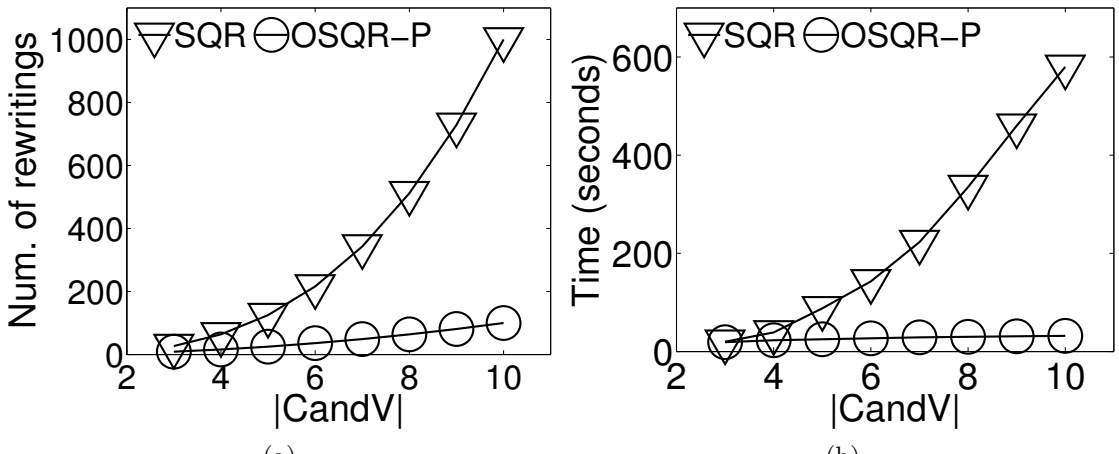
```
Q:SELECT { ?x, ?n, ?e, ?c }
  WHERE { ?x name ?n, ?x email ?e,
      ?x course ?c }
```

(b)

Figure 3.9: Experimental setup 4 (a) Views templates and (b) Query template

at iteration i. Contrarily, OSQR-P does not generate rewritings involving different views since these lead to empty results; synopses and ASK queries, which are less expensive, are executed to detect these empty results, and therefore, in each iteration i, essentially only $i + 2$ queries need to be executed by OSQR-P. Figure 3.10 shows the comparison. Through synopses and ASK queries, OSQR-P produces an *order of magnitude* less rewritings than SQR, resulting in an *order of magnitude* faster evaluation times for query Q.

Here, we investigate the influence of subquery (*i.e.,* triple pattern) ordering to OSQR. Since the objective of ordering is to improve the effectiveness of pruning, in OSQR, we *only* switch off merging views; the algorithm is denoted as OSQR-R. We consider the same experimental setup with the one used in our first experiment, shown in Figure 3.3. For this setup, Figure 3.11 compares the performance of OSQR-R using 3 different reordering strategies. The figure shows the number of ASK queries issued during query rewriting (to detect empty rewritings), and the evaluation time of the rewritten query. Note that all three reordering strategies result in the same number of nonempty rewritings, and only the numbers of ASK queries issued during rewriting are different; the latter affects the evaluation times, as shown in Figure 3.10. Using the proposed ordering on the size of CandV, OSQR-R



(a) (b)

Figure 3.10: Pruning empty rewritings (a) Rewritten queries over max CandV and (b) Eval. time over max CandV

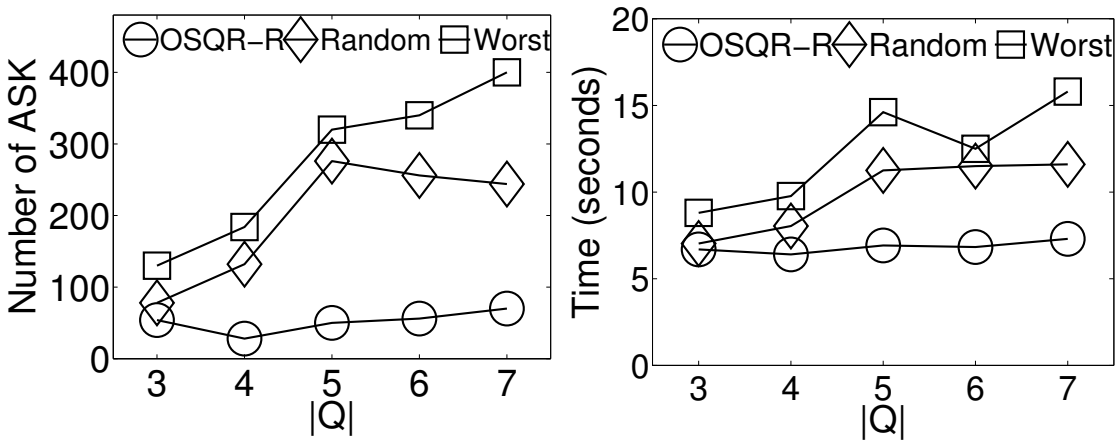


Figure 3.11: Optimizing rewriting generation (a) ASK queries over query size and (b) Eval. time over query size

detects the optimal ordering (which considers $p_1, p_2, \ldots, p_7$ in order) and generates up to an *order of magnitude* less ASK queries than either a random or the worst $(p_7, p_6, \ldots, p_1)$ ordering, resulting in nearly 60% savings in evaluation times.

3.4.2 Experimental Results from Jena TDB

Using the same query and view definitions, we have run the same set of experiments on Jena TDB, to demonstrate the flexibility and the store-independent property of our algorithms. In general, the results from Jena TDB are highly consistent with our observations from 4store. As is evident from Figure 3.12, the overall performance in Jena TDB of OSQR is several orders of magnitude better than the SQR in the first experiment using the setup in Figure 3.3. The situation is similar when using the experimental setup of Figure 3.5 and the results are shown in Figure 3.13. These trends are highly consistent with what we have observed from their comparison in 4store (Figures 3.4 and 3.6, respectively).

3.4.3 Concluding Remarks

Our experiments clearly illustrate the advantages of OSQR over SQR. These results are not limited to 4store but carry over to Jena. To summarize, our experiments show that: we have realized the *first* practical rewriting solution (OSQR), which provides, *in real time*, sound and complete access of RDF data, *independent* of underlying RDF stores, with good efficiency in practice (to rewrite and evaluate a query over tens to hundred of views) and without the need to materialize intermediate data.

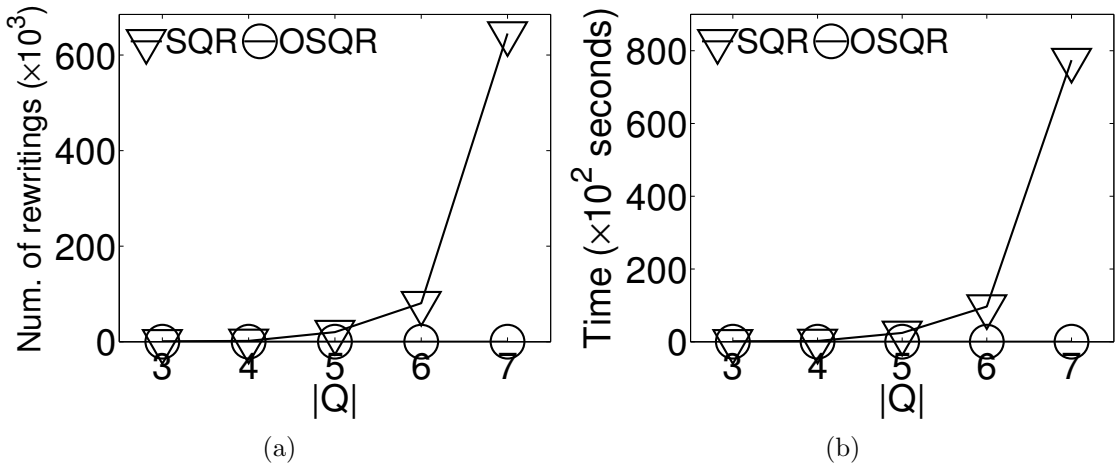


Figure 3.12: SQR vs. OSQR on Jena TDB (a) Rewritten queries over query size and (b) Eval. time over query size

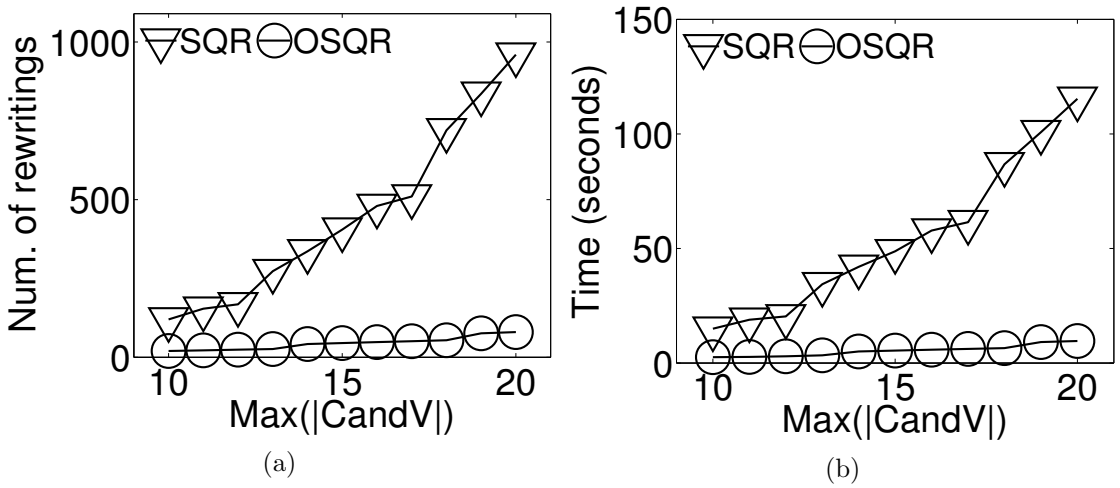


Figure 3.13: SQR vs. OSQR on Jena TDB (a) Rewritten queries over max CandV and (b) Eval. time over max CandV

3.5 Related Work

Query rewriting over views, motivated by a view-based approach to access control, has been well studied in relational databases (*e.g.,* [94]) and XML (*e.g.,* [44, 45]) databases. However, to the best of our knowledge, our work is the first on *native* query rewriting in SPARQL. SPARQL query rewriting combines the challenges that arise in the relational and XML settings: like the relational case, SPARQL query rewriting needs to synthesize multiple views; like the XML case, SPARQL query rewriting generates a query of exponential size. Previous work on rewriting SPARQL queries typically adopted a rule-based approach. Correndo *et al.* [38] perform rewritings using predefined rewriting rules, whereas our rewriting

techniques can dynamically compose the right views to rewrite a user query. Similarly in [32], the authors identify a set of tightest restrictions under which an XPath query can be rewritten over multiple views in PTIME. Such restrictions are expressed as rules during the rewriting; therefore, this approach is rule-based as well. Cautis *et al.* [33] present theoretical results for rewriting a query over multiple data sources; the authors studied the rewriting problem in the presence of embedded constraints from up to infinite data sources, and focused on the problem of deciding the *right* data sources that satisfy integrity constraints (*i.e.,* the expressibility and the support for the sources). Unlike our work, the rewriting algorithm in [33] does not guarantee completeness, and the optimization issue was not addressed.

Although our proposed SPARQL query rewriting techniques face similar challenges as the classical techniques for answering queries using views [52] and rewriting queries on semi-structured data [85], the actual rewriting steps differ significantly. In particular, relational techniques surveyed in [52] cannot efficiently address the problem in SPARQL. For example, the pruning power of MiniCon [90] vanishes due to the fact that all the variables in SQL-translated views (see Figure 3.2(b)) are *distinguished* variables [90]. Furthermore, our computation of variable mappings and selection of candidate views are distinct from the query containment techniques discussed in [90]. The exponential size of the rewriting is also unique to our setting, which forces us to address new challenges not found in [52]. To address those challenges, we propose novel optimization techniques to remove empty rewritings from execution.

Existing works on *general* query rewriting in RDF store [13] specify view definition in customized high-level languages, and perform query rewriting in an ad-hoc manner. In contrast, our work defines views in SPARQL, thus having more expressive power and wider applicability; furthermore, our SPARQL rewriting techniques are principled and independent of the underlying RDF stores.

3.6 Conclusion

We studied the classical problem of query rewriting over views in the context of SPARQL and RDF data. We proposed the first *sound* and *complete* query rewriting algorithm for SPARQL, with novel optimizations that (i) simplify individual rewritings by removing redundant triple patterns coming from the same view; (ii) eliminate rewritings with empty results based on a light-weight synopsis construction and efficient value-set intersection computation to estimate the size of joined triple patterns; and (iii) prune out big portions

of the search space of rewritings (that lead to empty results) by optimizing the sequence of subquery rewriting. Evaluation of our rewriting algorithm over two RDF stores showed its portability and its scalability in terms of query and view size. This work opens the gate to several interesting directions in future research, such as how to efficiently deal with variable predicates (instead of enumerating all predicates in the data to replace them) in query and view definition, how to partially materialize the views with the query rewriting in SPARQL to further improve the efficiency, and also, how to include other SPARQL features such as FILTER and OPTIONAL into the algorithm.

In the next chapter, we are going to study multiquery optimization, which further improves the throughput for evaluating the rewritten queries.

CHAPTER 4

SCALABLE MULTIQUERY OPTIMIZATION

4.1 Introduction

For many applications that need to perform query rewriting, *e.g.,* data integration [91] and fine-grained access control [71] on RDF data, a SPARQL query over views is often rewritten into an equivalent batch of SPARQL queries for evaluation over the base data. As the semantics of the rewritten queries in the same batch are commonly overlapped [57, 71], there is much room for sharing computation when executing these rewritten queries. This observation motivates us to revisit the classical problem of multiquery optimization (MQO) in the context of RDF and SPARQL.

Not surprisingly, MQO for SPARQL queries is NP-hard, considering that MQO for relational queries is NP-hard [99] and the established equivalence between SPARQL and relational algebra [20, 88]. It is tempting to apply the MQO techniques developed in relational systems to address the MQO problem in SPARQL. For instance, the work by P. Roy *et al.* [96] represented query plans in AND-OR DAGs and used heuristics to partially materialize intermediate results that could result in a promising query throughput. Similar themes can be seen in a variety of contexts, including relational queries [99, 100], XQueries [31], aggregation queries [110], or more recently as full-reducer tree queries [64]. These off-the-shelf solutions, however, are hard to engineer into RDF query engines in practice. The first source of complexity for using the relational techniques and the like stems from the physical design of RDF data themselves. While indexing and storing relational data commonly conform to a carefully calibrated relational schema, many variances existed for RDF data; *e.g.,* the giant triple table adopted in 3store and RDF-3X, the property table in Jena, and more recently the use of vertical partitioning to store RDF data. These, together with the disparate indexing techniques, make the cost estimation for an individual query operator (the corner stone for any MQO technique) highly error-prone and store-dependent. Moreover, as observed in previous works [12, 81], SPARQL queries feature more joins than typical SQL queries – a

fact that is also evident by comparing TPC benchmarks [10] with the benchmarks for RDF stores [28, 43, 50, 97]. While existing techniques commonly root on looking for the best plan in a greedy fashion, comparing the cost for alternative plans becomes impractical in the context of SPARQL, as the error for selectivity estimation inevitably increases when the number of joins increases [80, 103]. Finally, in W3C's envision [8], RDF is a very general data model; therefore, knowledge and facts can be seamlessly harvested and integrated from various SPARQL endpoints on the Web [2] (powered by different RDF stores). While a specialized MQO solution may serve inside the optimizer of certain RDF stores, it is more appealing to have a generic MQO framework that could smoothly fit into any SPARQL endpoint, which would be coherent with the design principle of RDF data model.

With the above challenges in mind, in this chapter, we study MQO of SPARQL queries over RDF data, with the objective to minimize total query evaluation time. Specifically, we employ query rewriting techniques to achieve desirable and consistent performance for MQO across different RDF stores, with the guarantee of *soundness* and *completeness*. While the previous works consider alignments for the common substructures in *acyclic* query plans [64, 96], we set forth to identify common subqueries (cyclic query graphs included) and rewrite them with SPARQL in a meaningful way. Unlike [96], which requires explicitly materializing and indexing the common intermediate results, our approach works on top of any RDF engine and ensures that the underlying RDF stores can automatically cache and reuse such results. In addition, a full range of optimization techniques in different RDF stores and SPARQL query optimizers can seamlessly support our MQO technique. Our contributions can be summarized as follows.

- We present a generic technique for MQO in SPARQL. Unlike the previous works that focus on synthesizing query plans, our technique summarizes similarity in the structure of SPARQL queries and takes into account the unique properties (*e.g.,* cyclic query patterns) of SPARQL.

- Our MQO approach relies on query rewriting, which is built on the algorithms for finding common substructures. In addition, we tailored efficient and effective optimizations for finding common subqueries in a batch of SPARQL queries.

- We proposed a practical cost model. Our choice of the cost model is determined both by the idiosyncrasies of the SPARQL language and by our empirical digest of how SPARQL queries are executed in existing RDF data management systems.

- Extensive experiments with large RDF data (close to 10 million triples) performed on three different RDF stores consistently demonstrate the efficiency and effectiveness of our approach over the baseline methods.

4.2 Problem Statement

We have introduced SPARQL in Chapter 2. In this chapter, we focus our discussion on the selection query of SPARQL. In particular, we distinguish two types of selection queries by the way in which the search patterns are specified:

Type 1: Q := SELECT RD WHERE GP

Type 2: $\mathsf{Q_{OPT}}$:= SELECT RD WHERE GP (OPTIONAL $\mathsf{GP_{OPT}})^+$

Consider the data and SPARQL query in Figure 4.1(a) and (b). The query looks for triples whose subjects (each corresponding to a person) have the predicates name and zip, with the latter having the value 10001 as object. For these triples, it returns the object of the name predicate. Due to the first OPTIONAL clause, the query also returns the object of predicate mbox, if the predicate exists. Due to the second OPTIONAL clause, the query also independently returns the object of predicate www, if the predicate exists. Evaluating the query over the input data D (can be viewed as a graph) results in output $\mathsf{Q_{OPT}}(\mathsf{D})$, as shown in Figure 4.1(c).

We associate with each query Q ($\mathsf{Q_{OPT}}$) a *query graph pattern* corresponding to its pattern GP (resp., GP (OPTIONAL $\mathsf{GP_{OPT}})^+$). Formally, a query graph pattern is a 4-tuple (V, E, ν, μ) where V and E stand for vertices and edges, ν and μ are two functions which assign labels (*i.e.,* constants and variables) to vertices and edges of GP, respectively. Vertices represent the subjects and objects of a triple; gray vertices represent constants, and white vertices represent variables. Edges represent predicates; dashed edges represent predicates in the optional patterns $\mathsf{GP_{OPT}}$, and solid edges represent predicates in the required patterns GP. Figure 4.2 shows a pictorial example for the query in Figure 4.1(b). Its query graph patterns GP and $\mathsf{GP_{OPT}}$s are defined separately. GP is defined as (V, E, ν, μ), where $V = \{v_1, v_2, v_3\}$, $E = \{e_1, e_2\}$ and the two naming functions $\nu = \{\nu_1 : v_1 \rightarrow ?x, \nu_2 : v_2 \rightarrow ?n, \nu_3 : v_3 \rightarrow 10001\}$, $\mu = \{\mu_1 : e_1 \rightarrow name, \mu_2 : e_2 \rightarrow zip\}$. For the two OPTIONALs, they are defined as $\mathsf{GP_{OPT}}_1 = (V', E', \nu', \mu')$, where $V' = \{v_1, v_4\}$, $E' = \{e_3\}$, $\nu' = \{\nu'_1 : v_1 \rightarrow ?x, \nu'_2 : v_4 \rightarrow ?m\}$, $\mu' = \{\mu'_1 : e_3 \rightarrow mbox\}$; Likewise, $\mathsf{GP_{OPT}}_2 = (V'', E'', \nu'', \mu'')$, where $V'' = \{v_1, v_5\}$, $E'' = \{e_4\}$, $\nu'' = \{\nu''_1 : v_1 \rightarrow ?x, \nu''_2 : v_5 \rightarrow ?p\}$, $\mu'' = \{\mu''_1 : e_4 \rightarrow www\}$.

subj	pred	obj
p1	name	"Alice"
p1	zip	10001
p1	mbox	alice@home
p1	mbox	alice@work
p1	www	http://home/alice
p2	name	"Bob"
p2	zip	"10001"
p3	name	"Ella"
p3	zip	"10001"
p3	www	http://work/ella
p4	name	"Tim"
p4	zip	"11234"

(a)

```
SELECT   ?name, ?mail, ?hpage
WHERE    { (?x, name, ?name), (?x, zip, 10001),
           OPTIONAL {(?x, mbox, ?mail) }
           OPTIONAL {(?x, www, ?hpage) }}
```

(b)

name	mail	hpage
"Alice"	alice@home	
"Alice"	alice@work	
"Alice"		http://home/alice
"Bob"		
"Ella"		http://work/ella

(c)

Figure 4.1: An example (a) Input data D, (b) Example query $\mathsf{Q_{OPT}}$ and (c) Output $\mathsf{Q_{OPT}(D)}$

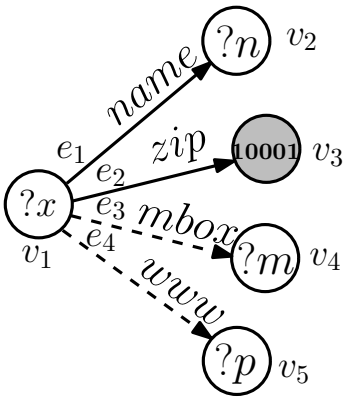
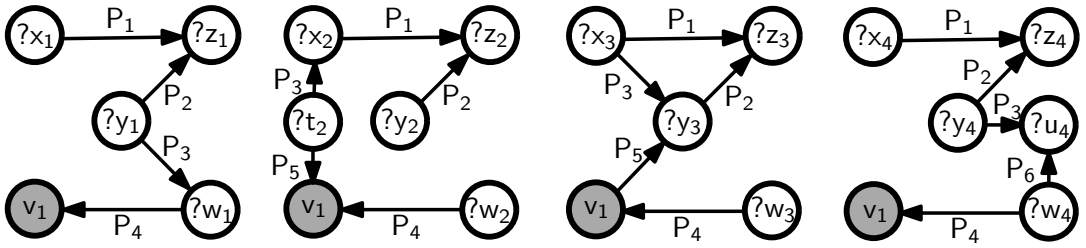


Figure 4.2: A query graph

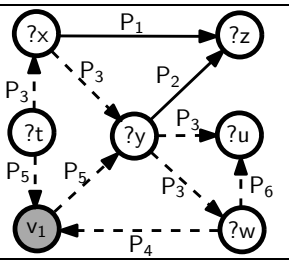
Formally, the problem of MQO in SPARQL, from a query rewriting perspective, is defined as follows: Given a data graph G, and a set Q of **Type 1** queries, compute a new set Q_{OPT} of **Type 1** and **Type 2** queries, evaluate Q_{OPT} over G, and distribute the results to the queries in Q. There are two requirements for the rewriting approach to MQO: (i) The query results of Q_{OPT} can be used to produce the same results as executing the original queries in Q, which ensures the *soundness* and *completeness* of the rewriting; and (ii) the evaluation time of Q_{OPT}, including query rewriting, execution, and result distribution, should be less than the baseline of executing the queries in Q sequentially. To ease presentation, we assume that the input queries in Q are of **Type 1**, while the output (optimized) queries are either of **Type 1** or **Type 2**. Our optimization techniques can easily handle more general scenarios where both query types are given as input (section 4.4).

We use a simple example to illustrate the MQO envisioned and some challenges for the rewriting approach. Figure 4.3(a)-(d) show the graph representation of four queries of **Type 1**. Figure 4.3(e) shows a **Type 2** query Q_{OPT} that *rewrites* all four input queries into one. To generate query Q_{OPT}, we identify the (largest) common subquery in all four queries: the subquery involving triples (?x, P_1, ?z), (?y, P_2, ?z) (the second largest common subquery involves only one predicate, P_3 or P_4). This common subquery constitutes the graph pattern GP of Q_{OPT}. The remaining subquery of each individual query generates an OPTIONAL clause in Q_{OPT}. Note that by generating a query like Q_{OPT}, the triple patterns in GP of Q_{OPT} are evaluated only *once*, instead of being evaluated for multiple times when the input queries are executed independently. Intuitively, this is where the savings MQO could be brought. As mentioned earlier, MQO must consider generic directed graphs, possibly with cyclic patterns, which makes it hard to adapt existing techniques for this optimization. Also, the proposed optimization has a unique characteristic in that it leverages SPARQL-specific



(a) (b) (c) (d)

```
SELECT *
WHERE { (?x, P1, ?z), (?y, P2, ?z),
   OPTIONAL {(?y, P3, ?w), (?w, P4, v1) }
   OPTIONAL {(?t, P3, ?x), (?t, P5, v1), (?w, P4, v1) }
   OPTIONAL {(?x, P3, ?y), (v1, P5, ?y), (?w, P4, v1) }
   OPTIONAL {(?y, P3, ?u), (?w, P6, ?u), (?w, P4, v1) }
}
```



(e)

```
SELECT *
WHERE { (?w, P4, v1),
   OPTIONAL {(?x1, P1, ?z1), (?y1, P2, ?z1),
             (?y1, P3, ?w) }
   OPTIONAL {(?x2, P1, ?z2), (?y2, P2, ?z2),
             (?t2, P3, ?x2), (?t2, P5, v1) }
   OPTIONAL {(?x3, P1, ?z3), (?y3, P2, ?z3),
             (?x3, P3, ?y3), (v1, P5, ?y3) }
   OPTIONAL {(?x4, P1, ?z4), (?y4, P2, ?z4),
             (?y4, P3, ?u4), (?w, P6, ?u4) }
}
```

pattern p	α(p)
(?x, P1, ?z)	15%
(?y, P2, ?z)	9%
(?y, P3, ?w)	18%
(?w, P4, v_1)	4%
(?t, P5, v_1)	2%
(v_1, P5, ?t)	7%
(?w, P6, ?u)	13%

(f)

Figure 4.3: Multiquery optimization examples (a) Query Q_a, (b) Query Q_b, (c) Query Q_c, (d) Query Q_d, (e) Example query $\mathsf{Q}_{\mathsf{OPT}}$ and (f) Structure and cost-based optimization

features such as the OPTIONAL clause for query rewriting.

Note that the above rewriting only considers query structures, without considering query selectivity. Suppose we know the selectivity $\alpha(\mathsf{p})$ of each pattern p in the queries, as shown in Figure 4.3(f). Let us assume a simple cost model in which the cost of each query Q or $\mathsf{Q}_{\mathsf{OPT}}$ is equal to the minimum selectivity of the patterns in GP; we ignore for now the cost of OPTIONAL patterns, which is motivated by how *real* SPARQL engines evaluate queries (The actual cost model used in this work is discussed in Section 4.3-D.). So, the cost for all four queries Q_1 to Q_4 is respectively 4, 2, 4, and 4 (with queries executed on a dataset of size 100). Therefore, executing all queries individually (without optimization)

costs 4+2+4+4 = 14. In comparison, the cost of the structure-based only optimized query in Figure 4.3(e) is 9, resulting in a saving of approximately 30%. Now, consider another rewriting in Figure 4.3(f) that results in from optimization along the second largest common subquery that just contains P_4. The cost for this query is only 4, which leads to even more savings, although the rewriting utilizes a smaller common subquery. As this simple example illustrates, it is critical for MQO to construct a cost model that integrates query structure overlap with selectivity estimation.

4.3 The Algorithm

Our MQO algorithm, shown in Figure 4.4, accepts as input a set $\mathsf{Q} = \{\mathsf{Q}_1, \ldots, \mathsf{Q}_n\}$ of n queries over a graph G. Without loss of generality, assume the sets of variables used in different queries are distinct. The algorithm identifies whether there is a *cost-effective* way to share the evaluation of structurally-overlapping graph patterns among the queries in Q. At a high level, the algorithm works as follows: (1) It partitions the input queries into groups, where queries in the same group are more likely to share common subqueries that can be optimized through query rewriting; (2) it rewrites a number of **Type 1** queries in each group to their correspondent cost-efficient **Type 2** queries; and (3) it executes the rewritten queries and *distributes* the query results to the original input queries (along with a refinement). Several challenges arise during the above process: (i) There exists an exponential number of ways to partition the input queries. We thus need a heuristic to prune out the space of less optimal partitioning of queries. (ii) We need an efficient algorithm to identify potential common subqueries for a given query group. And (iii) since different common subqueries result in different query rewritings, we need a robust cost model to compare candidate rewriting strategies. We describe how we tackle these challenges next.

4.3.1 Bootstrapping

Finding structural overlaps for a set of queries amounts to finding the isomorphic subgraphs among the corresponding query graphs. This process is computationally expensive (the problem is NP-hard [27] in general), so ideally we would like to find these overlaps only for groups of queries that will eventually be optimized (rewritten). That is, we want to minimize (or ideally eliminate) the computation spent on identifying common subgraphs for query groups that lead to less optimal MQO solutions. One heuristic we adopt is to quickly prune out subsets of queries that clearly share little in query graphs, without going to the next expensive step of computing their common subqueries; therefore, the group of queries that have few predicates in common will be pruned from further consideration for

Input: Set $\mathsf{Q} = \{\mathsf{Q}_1, \ldots, \mathsf{Q}_n\}$
Output: Set $\mathsf{Q}_{\mathsf{OPT}}$ of optimized queries

// Step 1: `Bootstrapping the query optimizer`
Run k-means on Q to generate a set $\mathcal{M} = \{\mathsf{M}_1, \ldots, \mathsf{M}_k\}$ of k query groups based on query similarity in terms of their predicate sets;

// Step 2: `Refining query clusters`
for *each query group* $\mathsf{M} \in \mathcal{M}$ **do**
  Initialize a set $\mathcal{C} = \{\mathsf{C}_1, \ldots, \mathsf{C}_{|\mathsf{M}|}\}$ of $|\mathsf{M}|$ clusters;
  for *each query* $\mathsf{Q}_i \in \mathsf{M}, 1 \leq i \leq |\mathsf{M}|$ **do** $\mathsf{C}_i = \mathsf{Q}_i$;
  while $\exists$ *untested pair* $(\mathsf{C}_i, \mathsf{C}_{i'})$ *with* $\mathsf{argmax}(\mathcal{J}accard(\mathsf{C}_i, \mathsf{C}_{i'}))$ **do**
    Let $\mathsf{Q}^{ii'} = \{\mathsf{Q}_1^{ii'}, \ldots, \mathsf{Q}_m^{ii'}\}$ be the queries of $\mathsf{C}_i \cup \mathsf{C}_{i'}$;
    Let $\mathcal{S}$ be the top-s most selective triple patterns in $\mathsf{Q}^{ii'}$;
    // Step 2.1: `Building compact linegraphs`
    Let $\mu_\cap \leftarrow \mu_1 \cap \mu_2 \ldots \cap \mu_m$ and $\tau = \{\emptyset\}$;
    for *each query* $\mathsf{Q}_j^{ii'} \in \mathcal{Q}^{ii'}$ **do**
      Build linegraph $\mathcal{L}(\mathsf{Q}_j^{ii'})$ with only the edges in $\mu_\cap$;
      Keep indegree matrix m_j^-, outdegree matrix m_j^+ for $\mathcal{L}(\mathsf{Q}_j^{ii'})$;
    for *each vertex* e *defined in* $\mu_\cap$ *and* $\mu_\cap(e) \neq \emptyset$ **do**
      Let $I = m_1^-[e] \cap \ldots \cap m_m^-[e]$ **and** $O = m_1^+[e] \cap \ldots \cap m_m^+[e]$;
      if $I = O = \emptyset$ **then** $\mu_\cap(e) \stackrel{\mathsf{def}}{=} \emptyset$ and $\tau = \tau \cup \{$triple pattern on $e\}$;
    for $\mathcal{L}(\mathsf{GP}_j), 1 \leq j \leq m$ **do**
      Prune the $\mathcal{L}(\mathsf{GP}_j)$ vertices not in $\mu_\cap$ and their incident edges;
    // Step 2.2: `Building product graphs`
    Build $\mathcal{L}(\mathsf{GP}_p) = \mathcal{L}(\mathsf{GP}_1) \otimes \mathcal{L}(\mathsf{GP}_2) \otimes \ldots \otimes \mathcal{L}(\mathsf{GP}_m)$;
    // Step 2.3: `Finding cliques in product graphs`
    $\{K_1, \ldots, K_r\} = AllMaximalClique(\mathcal{L}(\mathsf{GP}_p))$;
    if $r = 0$ **then goto** 22;
    for *each* K_i, $i = 1, 2, \ldots, r$ **do**
      find all $K_i' \subseteq K_i$ having the maximal strong covering tree in K_i;
    sort $\mathsf{SubQ} = \{K_1', \ldots, K_t'\} \cup \tau$ in descending order by size;
    Initialize $K = \emptyset$;
    for *each* $q_i \in \mathsf{SubQ}$, $i = 1, 2, \ldots, t + |\tau|$ **do**
      if $\mathcal{S} \cap q_i \neq \emptyset$ **then** Set $K = q_i$ and **break**
    if $K \neq \emptyset$ **then**
      Let $\mathsf{C}_{tmp} = \mathsf{C}_i \cup \mathsf{C}_{i'}$ and *cost(*C_{tmp}*)=cost(subquery for* K*)*;
      if *cost(*C_{tmp}*)* $\leq$ *cost(*C_i*) + cost(*$\mathsf{C}_{i'}$*)* **then**
        Put K with $\mathcal{C}_{tmp}$;
        remove C_i, $\mathsf{C}_{i'}$ from $\mathcal{C}$ and add C_{tmp};

// Step 3: `Generating optimized queries`
  for *each cluster* C_i *in* $\mathcal{C}$ **do**
    if *a clique* K *is associated with* C_i **then**
      Rewrite queries in C_i using triple patterns in K;
    Output the query into set $\mathsf{Q}_{\mathsf{OPT}}$;
return $\mathsf{Q}_{\mathsf{OPT}}$.

Figure 4.4: Multiquery optimization algorithm

optimization. We thus define the similarity metric for two queries as the Jaccard similarity of their predicate sets. The rational is that if the Jaccard similarity of two queries is small, their structural overlap in query graphs must also be small; therefore, it is safe to not consider grouping such queries for MQO. We implement this heuristic as a *bootstrap* step in line 1 using k-means clustering (with Jaccard as the similarity metric) for an initial partitioning of the input queries into a set $\mathcal{M}$ of k query groups. Notice that the similarity metric identifies queries with substantial overlaps in their predicate sets, ignoring for now the common substructure and the selectivity of these predicates.

4.3.2 Refining Query Clusters

Starting with the k-means generated groups $\mathcal{M}$, we refine the partitioning of queries further based on their structure similarity and the estimated cost. To this end, we consider each query group generated from the k-means clustering $\mathsf{M} \in \mathcal{M}$ in isolation (since queries across groups are guaranteed to be sufficiently different) and perform the following steps: In lines 5–30, we (incrementally) *merge* structurally similar queries within M through *hierarchical clustering* [59], and generate query clusters such that each query cluster is optimized together (*i.e.,* results in one **Type 2** query). Initially, we create one *singleton* cluster C_i for each query Q_i of M (line 4). Given two clusters C_i and $\mathsf{C}_{i'}$, we have to determine whether it is more cost-efficient to merge the two query clusters into a single cluster (*i.e.,* a single **Type 2** query) than to keep the two clusters separate (*i.e.,* executing the corresponding two queries independently). From the previous iteration, we already know the cost of the optimized queries for each of the C_i and $\mathsf{C}_{i'}$ clusters. To determine the cost of the merged cluster, we have to compute the query that merges all the queries in C_i and $\mathsf{C}_{i'}$ through rewriting, which requires us to compute the common substructure of all these queries, and to estimate the cost of the rewritten query generated from the merged clusters. For the cost computation, we do some preliminary work here (line 7) by identifying the most selective triple patterns from the two clusters (selectivity is estimated by [103]). Note that our refinement of M might lead to more than one queries: one for each *cluster* of M, in the form of either **Type 1** or **Type 2**.

While finding the maximum common subgraph for two graphs is known to be NP-hard [27], the challenge here is asymptotically harder as it requires finding the largest common substructures for *multiple* graphs. Existing solutions on finding common subgraphs also assume *untyped* edges and nodes in *undirected* graphs. However, in our case, the graphs represent queries, and different triple patterns might correspond to different semantics (*i.e.,*

typed and directed). Thus, the predicates and the constants associated with nodes must be taken into consideration. This mix of typed, constant, and variable nodes/edges is not typical in classical graph algorithms, and therefore, existing solutions can not be directly applied for query optimization. We therefore propose an efficient algorithm to address these challenges.

In a nutshell, our algorithm follows the principle of finding the maximal common edge subgraphs (MCES) [93, 112]. Concisely, three major substeps are involved (steps 2.1 to 2.3 in Figure 4.4): (a) transforming the input query graphs into the equivalent linegraph representations; (b) generating a product graph from the linegraphs; and (c) executing a tailored clique detection algorithm to find the maximal cliques in the product graph (a maximal clique corresponds to an MCES). We describe these substeps in details next.

The linegraph $\mathcal{L}(\mathsf{G})$ of a graph G is a directed graph built as follows. Each node in $\mathcal{L}(\mathsf{G})$ corresponds to an edge in G, and there is an edge between two nodes in $\mathcal{L}(\mathsf{G})$ if the corresponding edges in G share a common node. Although it is straightforward to transform a graph into its linegraph representation, the context of MQO raises new requirements for the linegraph construction. We represent the linegraph of a query graph pattern in a 4-tuple, defined as $\mathcal{L}(\mathsf{G}) = (\mathcal{V}, \mathcal{E}, \pi, \omega)$. During linegraph construction, besides the inversion of nodes and edges for the query graph, our transformation also assigns to each edge in the linegraph one of 4 labels ($\ell_0 \sim \ell_3$). Specifically, for two triple patterns, there are 4 possible joins between their subjects and objects (ℓ_0 = subject-subject, ℓ_1 = subject-object, ℓ_2 = object-subject, ℓ_3 = object-object). The assignment of labels on linegraph edges captures these four join types (useful for pruning and will become clear shortly). Figure 4.5 (a)-(d) shows the linegraphs for the queries in Figure 4.3(a)-(d).

The classical solution for finding common substructures of input graphs requires building Cartesian products on their linegraphs. This raises challenges in scalability when finding the maximum common substructure for *multiple* queries in *one* shot. To avoid the foreseeable explosion, we propose fine-grained optimizations (lines 8–16) to keep linegraphs as small as possible so that only the most *promising* substructures would be transformed into linegraphs, with the rest being temporarily *masked* from further processing.

To achieve the above, queries in $\mathsf{Q}^{ii'}$ pass through a two-stage optimization. In the first stage (lines 8–11), we identify (line 8) the common predicates in $\mathsf{Q}^{ii'}$ by building the intersection $\mu_\cap$ for all the labels defined in the μ's (recall that function μ assigns predicate names to graph edges). Predicates that are not common to all queries can be safely pruned, since by definition they are not part of any common substructure, *e.g.,*

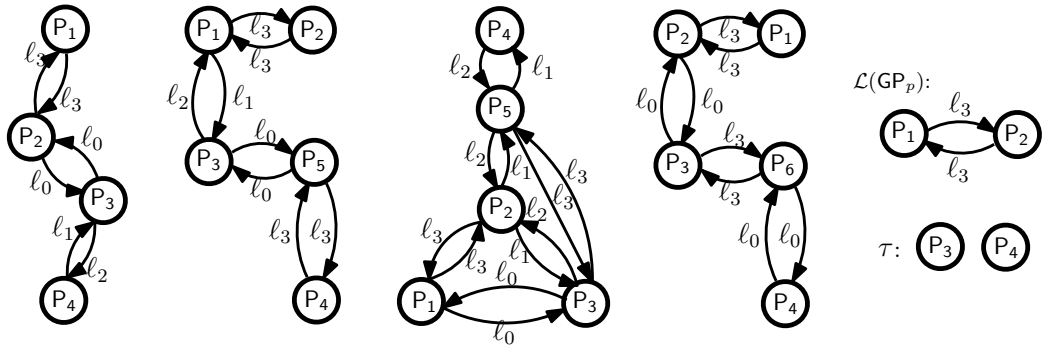


Figure 4.5: Examples for finding common substructures, (a)–(d) linegraphs for queries $\mathsf{Q_a}$–$\mathsf{Q_d}$, (e) their common substructures

$\mathsf{P_5}$ and $\mathsf{P_6}$ in Figure 4.3. In finding the intersection of predicates, the algorithm also checks for *compatibility* between the corresponding subjects and objects, so that same-label predicates with different subjects/objects are not added into $\mu_\cap$. In addition, we maintain two adjacency matrices for a linegraph $\mathcal{L}(\mathsf{GP})$: the indegree matrix m^- storing all incoming, and the outdegree matrix m^+ storing all outgoing edges from $\mathcal{L}(\mathsf{GP})$ vertices. For a vertex v, we use $m^-[v]$ and $m^+[v]$, respectively, to denote the portion of the adjacency matrices storing the incoming and outgoing edges of v. For example, the adjacency matrices for vertex $\mathsf{P_3}$ in linegraph $\mathcal{L}(\mathsf{Q_1})$ of Figure 4.5 are $m_1^+[\mathsf{P_3}] = [\emptyset, \ell_0, \emptyset, \ell_2, \emptyset, \emptyset]$, $m_1^-[\mathsf{P_3}] = [\emptyset, \ell_0, \emptyset, \ell_1, \emptyset, \emptyset]$, while for linegraph $\mathcal{L}(\mathsf{Q_2})$, they are $m_2^+[\mathsf{P_3}] = [\ell_2, \emptyset, \emptyset, \emptyset, \ell_0, \emptyset]$, $m_2^-[\mathsf{P_3}] = [\ell_1, \emptyset, \emptyset, \emptyset, \ell_0, \emptyset]$.

In the second stage (lines 12–16), to further reduce the size of linegraphs, for each linegraph vertex e, we compute the Boolean intersection for the $m^-[e]$'s and $m^+[e]$'s from all linegraphs, respectively (line 13). We also prune e from $\mu_\cap$ if both intersections equal $\emptyset$ and set aside the triple pattern associated with e in a set τ (line 14). Intuitively, this optimization acts as a *look-ahead* step in our algorithm, as it quickly detects the cases where the common subqueries involve only one triple pattern (those in τ). Moreover, it also improves the efficiency of the clique detection (step 2.2 and 2.3) due to the smaller sizes of input linegraphs. Going back to our example, just by looking at the m_1^-, m_1^+, m_2^-, m_2^+, it is easy to see that the intersection $\cap m_i^+[\mathsf{P_3}] = \cap m_i^-[\mathsf{P_3}] = \emptyset$ for all the linegraphs of Figure 4.5(a)-(d). Therefore, our optimization temporarily masks $\mathsf{P_3}$ (so as $\mathsf{P_4}$) from the expensive clique detection in the following two steps.

Next, we illustrate how to build a product graph. The product graph $\mathcal{L}(\mathsf{GP}_p) := (\mathcal{V}_p, \mathcal{E}_p, \pi_p, \omega_p)$ of two linegraphs, $\mathcal{L}(\mathsf{GP}_1) := (\mathcal{V}_1, \mathcal{E}_1, \pi_1, \omega_1)$ and $\mathcal{L}(\mathsf{GP}_2) := (\mathcal{V}_2, \mathcal{E}_2, \pi_2, \omega_2)$, is denoted as $\mathcal{L}(\mathsf{GP}_p) := \mathcal{L}(\mathsf{GP}_1) \otimes \mathcal{L}(\mathsf{GP}_2)$. The vertices $\mathcal{V}_p$ in $\mathcal{L}(\mathsf{GP}_p)$ are defined on the Cartesian product of $\mathcal{V}_1$ and $\mathcal{V}_2$. In order to use product graphs in MQO, we optimize the

standard definition with the additional requirement that vertices paired together must have the same label (*i.e.,* predicate). That is, $\mathcal{V}_p := \{(v_1, v_2) \mid v_1 \in \mathcal{V}_1 \wedge v_2 \in \mathcal{V}_2 \wedge \pi_1(v_1) = \pi_2(v_2)\}$, with the labeling function defined as $\pi_p := \{\pi_p(v) \mid \pi_p(v) = \pi_1(v_1)$, with $v = (v_1, v_2) \in \mathcal{V}_p\}$. For the product edges, we use the standard definition, which creates edges in the product graph between two vertices (v_{1i}, v_{2i}) and (v_{1j}, v_{2j}) in $\mathcal{V}_p$ if either (i) the same edges (v_{1i}, v_{1j}) in $\mathcal{E}_1$, and (v_{2i}, v_{2j}) in $\mathcal{E}_2$ exist; or (ii) no edges connect v_{1i} with v_{1j} in $\mathcal{E}_1$, and v_{2i} with v_{2j} in $\mathcal{E}_2$. The edges due to (i) are termed as *strong* connections, while those for (ii) as *weak* connections [112].

Since the product graph for two linegraphs conforms to the definition of linegraph, we can recursively build the product for multiple linegraphs (line 17). Theoretically, there is an exponential blowup in size when we construct the product for multiple linegraphs. In practice, thanks to our optimizations in Steps 2.1 and 2.2, our algorithm is able to accommodate tens to hundred of queries, and generates the product graph efficiently (which will be verified through Section 4.5). Figure 4.5(e) shows the product linegraph $\mathcal{L}(\mathsf{GP}_p)$ for the running example.

A (maximal) clique with a strong covering tree (a tree only involving strong connections) in the product graph equals to an MCES – a (maximal) common subquery in essence. In addition, we are interested in finding cost-effective common subqueries. To verify if the found common subquery is selective, it is checked with the set $\mathcal{S}$ (from line 7) of selective query patterns.

In the algorithm, we proceed by finding all maximal cliques in the product graph (line 18), a process for which many efficient algorithms exist [68, 83, 108]. For each discovered clique, we identify its subcliques with the maximal strong covering trees (line 21). For the $\mathcal{L}(\mathsf{GP}_p)$ in Figure 4.5(e), it results in one clique (itself): *i.e.,* $K'_1 = \{\mathsf{P}_1, \mathsf{P}_2\}$. As the cost of subqueries is another dimension for query optimization, we look for the substructures that are both large in size (*i.e.,* the number of query graph patterns in overlap) and correspond to selective common subqueries. Therefore, we first sort SubQ (contributed by K's and τ, line 22) by their sizes in descending order, and then loop through the sorted list from the beginning and stop at the *first* substructure that intersects $\mathcal{S}$ (lines 22–25), *i.e.,* P_4 in our example. We then merge (if it is cost-effective, line 28) the queries whose common subquery is reflected in K and also merge their corresponding clusters into a new cluster (while remembering the found common subquery) (lines 26–30). The algorithm repeats lines 5–30 until every possible pair of clusters have been tested and no new cluster can be generated.

4.3.3 Generating Optimized Queries and Distributing Results

After the clusters are finalized, the algorithm rewrites each cluster of queries into one query and thus generates a set of rewritings $\mathsf{Q}_{\mathsf{OPT}}$ (lines 31–34). The result from evaluating $\mathsf{Q}_{\mathsf{OPT}}$ over the data is a superset of evaluating the input queries Q (more expositions in Section 4.3.5). Therefore, we must filter and distribute the results from the execution of $\mathsf{Q}_{\mathsf{OPT}}$. This necessitates one more step of parsing the result of $\mathsf{Q}_{\mathsf{OPT}}$ (refer to Figure 4.1(c)), which checks each row of the result against the RD of each query in Q. Note that the result description $\mathsf{RD}_{\mathsf{OPT}}$ is always the union of RD_is from the queries being optimized, and we record the mappings between the variables in the rewritings and the variables in the original input queries. As in Figure 4.1(c), the result of a **Type 2** query might have *empty* (*null*) columns corresponding to the variables from the OPTIONAL clause. Therefore, a row in the result of $\mathsf{RD}_{\mathsf{OPT}}$ might not conform to the description of every RD_i. The goal of parsing is to identify the valid overlap between each row of the result and the individual RD_i, and return to each query the result it is supposed to get. To achieve this, the parsing algorithm performs a Boolean intersection between each row of result and each RD_i: if the columns of this row corresponding to those columns of RD_i are not null, the algorithm distributes the corresponding part of this row to Q_i as one of its query answers. This step iterates over each row and each Q_i that composed the **Type 2** query. The parsing on the results of $\mathsf{Q}_{\mathsf{OPT}}$ only requires a linear scan on the results to the rewritten query. Therefore, it can be done on-the-fly as the results of $\mathsf{Q}_{\mathsf{OPT}}$ is streamed out from the evaluation.

4.3.4 Cost Model for SPARQL MQO

The design of our cost module is motivated by the way in which a SPARQL query is evaluated on popular RDF stores. This includes a well-justified principle that the most selective triple pattern is evaluated first [103] and that the $\mathsf{GP}_{\mathsf{OPT}}$ clause is evaluated on the result of GP (for the fact that $\mathsf{GP}_{\mathsf{OPT}}$ is a left-join). This suggests that a good optimization should keep the result cardinality from the common subquery as small as possible for two reasons: 1) The result cardinality of a **Type 2** SPARQL query is upper bounded by result cardinality of its GP clause since $\mathsf{GP}_{\mathsf{OPT}}$s are simply left-joins; 2) Intermediate result from evaluating the GP clause is not well indexed, which implies that a nonselective GP will result in significantly more efforts in processing the corresponding rewriting $\mathsf{GP}_{\mathsf{OPT}}$s.

In [103], the authors discussed the selectivity estimation for the conjunctive Basic Graph Patterns (BGP). In a nutshell, given a triple pattern $\mathsf{t} = (\mathsf{s}, \mathsf{p}, \mathsf{o})$, where each entry could be bound or unbound, its selectivity is estimated by $sel(t) = sel(s) \times sel(p) \times sel(o)$. *sel* is the selectivity estimation function, whose value falls in the interval of $[0, 1]$. Specifically,

for unbound variable, its selectivity equals 1. For bound variables/constants, depending on whether it is a subject, predicate, or object, different methods (*e.g.,* [103]) are used to implement *sel*. Notice that the formula implicitly assumes statistical independence for the subject, predicate, and object; thus, it is an approximation. Precomputed statistics of the dataset are also required. For a join between two triple patterns, independence assumption is adopted [103]. However, in practice, such estimation is not accurate enough for optimizing complex queries. The culprit comes from the fact that as the number of joins increases, the accuracy of the estimated selectivity drops quickly, resulting in a very loose estimation [81].

With the above limitations in mind, we propose a cost function for conjunctive SPARQL query. It has a simple design and roots on the well-justified principle in query optimization that the selective triple patterns have higher priorities in evaluation. Our cost model is an incarnation of this intuition, as in Formula 4.1:

$$\mathsf{Cost}(\mathsf{Q}) = \begin{cases} \mathsf{Min}(sel(t)) & \mathsf{Q} \text{ is a } \textbf{Type 1} \text{ query}, t \in \mathsf{GP} \\ \mathsf{Min}(sel(t)) + \Delta & \mathsf{Q} \text{ is a } \textbf{Type 2} \text{ query}, t \in \mathsf{GP} \end{cases} \tag{4.1}$$

For a **Type 1** conjunctive query, Formula 4.1 simply returns the selectivity for the most selective triple pattern in the query graph GP as the cost of evaluating Q. For a **Type 2** query, the cost is the summation of the cost on evaluating the common graph pattern GP and the cost on the evaluating the OPTIONALs, *i.e.,* the cost denoted by Δ. We extrapolate (backed by a comprehensive empirical study on three different RDF query engines) that Δ is a hidden function of (i) the cost of GP; (ii) the number of OPTIONALs; and (iii) the cost of the query pattern of each $\mathsf{GP}_{\mathsf{OPT}}$. However, we observed empirically that when the cost of GP is small (being selective), Δ would be a trivial value and $\mathsf{Cost}(\mathsf{Q})$ is mostly credited to the evaluation of GP. Hence, we can approximate $\mathsf{Cost}(\mathsf{Q})$ with the cost of GP in such cases. We show (experimentally) that using our cost model to choose a good common substructure can consistently improve the performance of query evaluation over the pure structure-based optimization (*i.e.,* without considering the evaluation cost of common subqueries) on different RDF stores.

Finally, notice that the proposed cost function requires using the precomputed statistics of the RDF dataset to estimate the selectivity of triple patterns. Therefore, a preprocessing step should be performed to collect some statistics from the dataset. This mainly includes: (i) building the histogram for distinct predicates in the dataset; and (ii) that for each disparate predicate, we build histograms for the subjects and objects attached to this predicate in the dataset. In practice, for some RDF stores, like Jena, part of such statistics (*e.g.,* the histogram of predicates) is provided by the SPARQL query optimizer and is

accessible for free; for the others (*e.g.,* Virtuoso and Sesame), the statistics of the dataset need to be collected in a preprocessing step.

4.3.5 Completeness and Soundness of Our MQO Algorithm

Suppose a **Type 2** rewritten query $\mathsf{Q_{OPT}}$ optimizes a set of n **Type 1** queries, *i.e.,* $\mathcal{Q} = \{\mathsf{Q}_1, \mathsf{Q}_2, \ldots, \mathsf{Q}_n\}$. Without loss of generality, denote the common relation (*i.e.,* the common subquery) used in $\mathsf{Q_{OPT}}$ as GP and its outer join relations (*i.e.,* the OPTIONALs) as GP_i $(i = 1, 2, \ldots, n)$. As we only consider conjunctive queries as input, by construction, $\mathcal{Q} = \cup_{i=1}^{n} \mathsf{GP} \bowtie \mathsf{GP}_i$ and $\mathsf{Q_{OPT}} = \cup_{i=1}^{n} \mathsf{GP} \ltimes \mathsf{GP}_i$. By the definition of left outer join $\ltimes$, $\mathsf{GP} \bowtie \mathsf{GP}_i \subseteq \mathsf{GP} \ltimes \mathsf{GP}_i$ for any i. It follows $\mathcal{Q} \subseteq \mathsf{Q_{OPT}}$ in terms of query results.

Soundness requires $\mathcal{Q} \supseteq \mathsf{Q_{OPT}}$. This is achieved by evaluating the results from $\mathsf{Q_{OPT}}$ and distributing the matched results to correspondent queries in $\mathcal{Q}$ (Section 4.3.3). As such, false positives are discarded and the remainings are valid bindings for one or more graph patterns in $\mathcal{Q}$. Therefore, $\mathcal{Q} \supseteq \mathsf{Q_{OPT}}$ in terms of results after the refining step.

Completeness and soundness together guarantee that the final answers resulted by our MQO techniques are equivalent to the results from evaluating queries in $\mathcal{Q}$ independently.

4.4 Extensions

For the ease of presentation, the input queries discussed so far are **Type 1** queries using constants as their predicates. It is interesting to note that with some minimal modifications to the algorithm and little preprocessing of the input, the algorithm in Figure 4.4 can optimize more general SPARQL queries. Here, we introduce two simple yet useful extensions: (i) optimizing input queries with variables as the predicates; and (ii) optimizing input queries of **Type 2** (*i.e.,* with OPTIONALs).

4.4.1 Queries with Variable Predicates

We treat variable predicates slightly differently from the constant predicates when identifying the structural overlap of input queries. Basically, a variable predicate from one query can be matched with any variable predicate in another query. In addition, each variable predicate of a query will correspond to one variable vertex in the linegraph representation, but the main flow of the MQO algorithm remains the same.

4.4.2 Handling TYPE 2 Queries

Our MQO algorithm takes a batch of **Type 1** SPARQL queries as input and rewrites them to another batch of **Type 1** and **Type 2** queries. It can be extended to optimize a batch of

input queries with both **Type 1** and **Type 2** queries.

To this end, it requires a preprocessing step on the input queries. Specifically, by the definition of left-join, a **Type 2** input query will be rewritten into its equivalent **Type 1** form, since our MQO algorithm only works on **Type 1** input queries. The equivalent **Type 1** form of a **Type 2** query GP (OPTIONAL $\mathsf{GP_{OPT}})^+$) consists two sets of queries: (i) a **Type 1** query solely using the GP as its query graph pattern; and (ii) the queries by replacing the left join(s) with inner join(s) between GP and each of the $\mathsf{GP_{OPT}}$ from the OPTIONAL, *i.e.,* $\cup$GP $\bowtie$ $\mathsf{GP_{OPT}}$. For example, to strip off the OPTIONALs in the **Type 2** query in Figure 4.6(a), applying the above preprocessing will result in a group of three **Type 1** rewritings as in Figure 4.6(b).

By applying the above transformation to all **Type 2** queries in the input and then passing the batch of queries to algorithm in Figure 4.4 for optimization, we can handle **Type 2** queries seamlessly. Finally, the result to the original **Type 2** query can be generated through the union of the results, from the transformed **Type 1** queries after MQO.

4.5 Experimental Evaluation

We implemented all algorithms in C++ and performed an extensive experimental evaluation using a 64-bit Linux machine with a 2GHz Intel Xeon(R) CPU and 4GB of memory.

Our evaluation is based on LUBM benchmark. The popular benchmark models universities with students, departments, *etc.*, using only 18 predicates [50]. This limits the complexity of queries we can evaluate (similar limitations in [28, 97]), and results in queries with considerable overlaps (which favors MQO but is not very realistic). Thus, we extended the LUBM data generator, and added a random subset from 50 new predicates to each person in the dataset, where predicate selectivity follows the distribution in Figure 4.7. Therefore, given the number of triples N in a dataset D, the number of times that a predicate appears in D (dubbed its frequency) is precisely its selectivity multiplied by N.

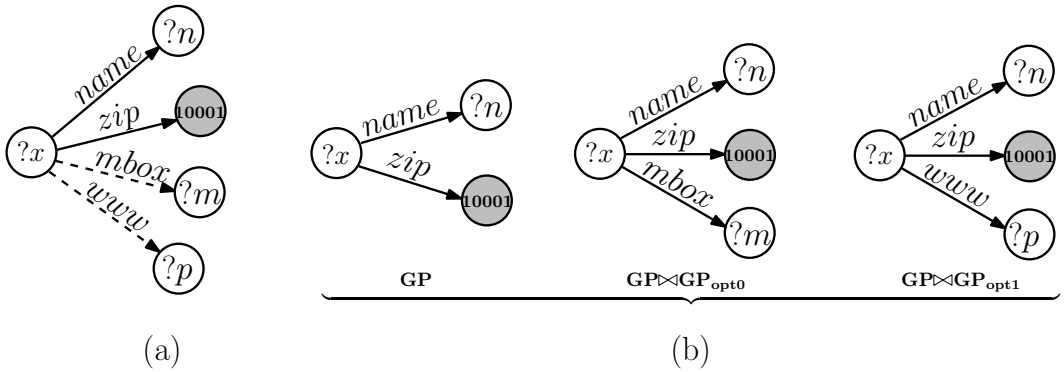


Figure 4.6: Convert (a) A **Type 2** query to (b) its equivalent **Type 1** form

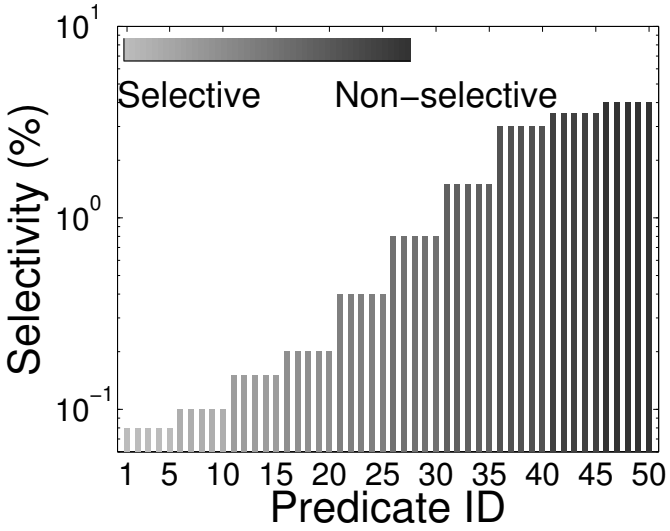


Figure 4.7: Predicate selectivity

We experimented with three popular RDF stores: Jena TDB 0.85, OpenLink Virtuoso 6.01, and Sesame Native 2.0. Here, we analyze mainly the experiments with Jena TDB. Results for the other two stores are highly consistent with the results from Jena TDB and will be reported at the end of this section. For all stores, we created full indexes using the technique in [114]. For Virtuoso, we also built bitmap indexes as reported in [23].

For all experiments, we measure the number of optimized queries and their end-to-end evaluation time, including query rewriting, execution, and result distribution. We compare our MQO algorithm with the evaluation without any optimization (*i.e.,* No-MQO), and the approach with structure-only optimization (*i.e.,* MQO-S). To realize the latter strategy as a baseline solution, we need to turn off all the cost-based comparisons in Figure 4.4. Specifically, in line 24 of Figure 4.4, instead of walking through the set of SubQ (which correspond to different common substructures), structure-based optimization (*i.e.,* MQO-S) simply returns the the largest clique (*i.e.,* the largest common subquery) for optimization.

Comparing MQO with MQO-S illustrates the benefits of blending structured-based with the cost-based optimization versus a purely structural approach. In the algorithms, we use the suffix -C to denote the cost by rewriting queries (*e.g.,* MQO-C) and the suffix -P to denote the cost by parsing and distributing the query results (*e.g.,* MQO-P). For finding cliques, we customized the Cliquer library [83], which is an efficient implementations for clique detection. For selectivity estimation, we implemented the technique in [103]. All experiments are performed using cold caches, and the bootstrapping parameter k in the k-means algorithm is set to $k = \lceil |\mathcal{Q}|/40 \rceil$. Table 4.1 provides the summary along with

Table 4.1: Parameter table

Parameter	Symbol	Default	Range
Dataset size	D	4M	3M to 9M
Number of queries	$\|\mathcal{Q}\|$	100	60 to 160
Size of query (num of triple patterns)	$\|\mathsf{Q}\|$	6	5 to 9
Number of seed queries	κ	6	5 to 10
Size of seed queries	$\|q_{\text{cmn}}\|$	$\sim \|\mathsf{Q}\|/2$	1 to 5
Max selectivity of patterns in Q	$\alpha_{max}(\mathsf{Q})$	random	0.1% to 4%
Min selectivity of patterns in Q	$\alpha_{min}(\mathsf{Q})$	1%	0.1% to 4%

ranges and the default values used for various parameters in our experiments.

LUBM has only 14 SPARQL queries, which have limited variance in both structure and evaluation cost. Therefore, we created a module to generate query sets $\mathcal{Q}$ with varying sizes $|\mathcal{Q}|$, where we generated queries that combine *star*, *chain*, and *circle* pattern structures. In addition, we attached to each person (as a subject) in the LUBM data a (random) subset of 50 new predicates $\mathsf{P_1} \sim \mathsf{P_{50}}$. In particular, we customized the data generator of LUBM in such a way that whenever a triple $((\mathsf{s}, \mathsf{P}_i, \mathsf{c}_i))$ is added to the data, c_i is an integer value serving as the object of this triple and it is set to the number of predicate P_i that has been added in the dataset so far. Therefore, triples with different predicates could join on their subjects or objects, so as the triple patterns in the query, which we will detail next.

Our query generator utilizes the aforementioned patterns in the customized data to compose queries. Specifically, we ensure that the queries have reasonably high randomness in structure (such that they are not replicas of limited query templates) and reasonable variances in selectivity (such that any predicate could be part of a query regardless of the structure). To this end, we first show how to compose three basic query patterns: star, chain, and circle with a set of four basic triple patterns; see Figure 4.8 (a) – (c). The star and the chain can be built with any number of triple patterns while the circle can only be built with an even number of triple patterns.

To blend the three basic patterns into one query Q with $|\mathsf{Q}|$ triple patterns, the generator

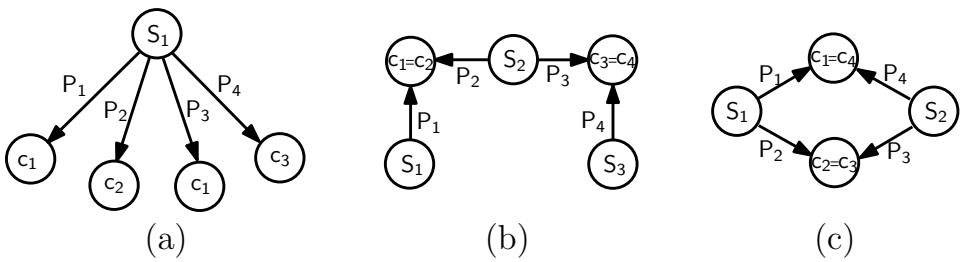


Figure 4.8: Three basic query patterns: (a) Star, (b) Chain, and (c) Circle

first randomly distributes the set of triple patterns into k groups of subqueries (k is a random integer, $k \in (0, |\mathsf{Q}|)$), with each subquery randomly composing one of the three basic patterns, *i.e.,* star, chain, and if possible, circle. To ensure Q to be conjunctive, the generator then makes equal the (randomly) chosen pairs of subjects and/or objects from the k subqueries by unifying their variable names or binding them to the same constant. This concludes composing the structure of Q. Finally, to ensure that Q conforms to the selectivity requirement posed by a specific experiment (refer to Table 4.1), the generator fills in the structure of Q with the predicates that would make Q a legitimate query.

In the experiments, a group of queries in $\mathcal{Q}$ were rendered to share a common seed subquery q_{cmn}. The generator first constructs q_{cmn} and the remaining portion of the queries independently. Then, by equaling the subjects and/or objects of these two subqueries, the generator propagates q_{cmn} over the group such that q_{cmn} joins with each of the subqueries in the group. In addition, individual query sizes $|\mathsf{Q}|$ can be varied where the probability of a predicate being part of a query conforms to its frequency in the dataset. We ensure that 90% of the queries in $\mathcal{Q}$ are amenable to optimization, while 10% are not. We use a parameter κ to determine seed queries that will be used to generate the queries in this 90%. For a given κ, κ seed-groups are generated, each corresponding to $\lceil(90/\kappa)\rceil$% of queries in $\mathcal{Q}$. The seed in each seed-group is what our algorithm will (hopefully) discover.

In short, we generated datasets and queries with various size, complexity, and statistics to evaluate the proposed MQO algorithm in a comprehensive way.

4.5.1 Experimental Results

The objective of our experiments is to evaluate: (i) how much each step of MQO (from bootstrapping step to cost estimation) contributes to the optimization, *i.e.,* drop in performance due to omission of each step; (ii) whether the combination of structure and cost-based optimization *consistently* outperforms purely structure-based optimizations; (iii) how well Algorithm MQO optimizes its alternatives, including the comparison with the baseline approach without any optimization, in *every* experimental setting; and (iv) whether Algorithm MQO consistently works across RDF stores.

We start with an experiment to illustrate the benefit of bootstrapping MQO using k-means. Figure 4.9 shows the cost of hierarchical clustering in Step 2 of MQO with (MQO-C) and without (MQO-noKM-C) bootstrapping. The figure shows an order of magnitude difference between the MQO-C and MQO-noKM-C, since without bootstrapping, Step 2 of MQO requires $\mathcal{O}((|\mathsf{Q}| \times |\mathcal{Q}|)^2)$ pairwise checks between all the queries in the input set $\mathcal{Q}$. The next experiment, in Figure 4.10, illustrates algorithm MQO-KM, which after Step 1 of MQO, finds

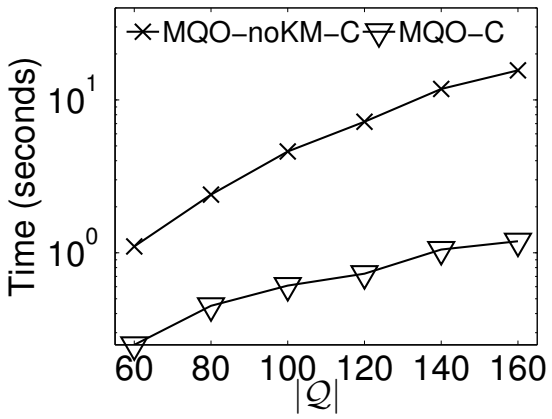


Figure 4.9: Clustering time

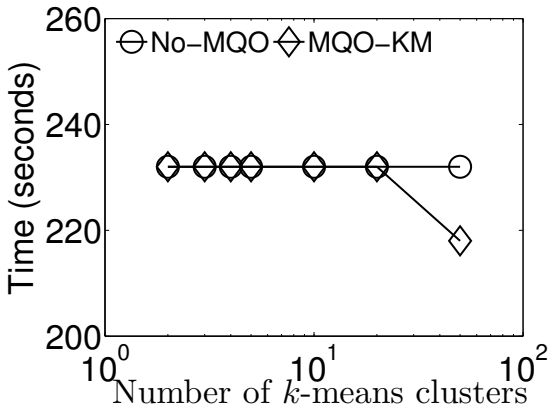


Figure 4.10: Evaluation time

the common substructures for the *coarse-grained* groups that are obtained from k-means and then performs Step 3 (*i.e.,* MQO-KM does not perform hierarchical clustering in Step 2). The figure shows that the resulting optimization has limited (less than 10%) to no benefits in evaluation time, when compared with the case of having no optimizations (No-MQO). This is because k-means ignores query structures and relies solely on the predicate names to determine groups. Therefore, the fine-grained groups that are produced by hierarchical clustering (in Step 2) are *necessary* for the considerable savings (as illustrated in the following experiments) in terms of evaluation times.

In the second set of experiments, we study scalability w.r.t. the cardinality $|\mathcal{Q}|$ of the query set $\mathcal{Q}$, for which we vary from 60 to 160 queries. As Figure 4.11 shows, both MQO and MQO-S are successful in identifying common substructures, the former resulting in up to 60% savings and the latter having up to 80% savings in terms of the number of queries, compared to No-MQO. However, in terms of evaluation times (see Figure 4.12), MQO-S results in *less* savings than MQO, with the former achieving up to 45%, and the latter up to 60% savings in evaluation times, when compared to No-MQO. So MQO is more efficient, despite generating a larger number of optimized queries than MQO-S. The following example, along with the example in Figure 4.3, illustrates this situation. Consider a set of queries $\mathcal{Q}$, such that (i) predicate p_{cmn} is common to all the queries in $\mathcal{Q}$; (ii) predicate p_1 is common to the subset $\mathcal{Q}_\infty \subset \mathcal{Q}$; and (iii) predicate p_2 is common to the subset of queries in $\mathcal{Q}_\in \subset \mathcal{Q}$, with $\mathcal{Q}_\infty \cap \mathcal{Q}_\in = \emptyset$. MQO-S looks only at the structure and thus, it may opt to generate a single optimized query for $\mathcal{Q}$ with $q_{\text{cmn}} = p_{\text{cmn}}$. If predicate p_{cmn} is not selective, while predicates p_1 and p_2 are highly selective, then MQO will generate two different optimized queries, one for set $\mathcal{Q}_\infty$ and involving q_1, and one for set $\mathcal{Q}_\in$ and involving p_2. As this example illustrates, MQO-S can generate fewer but cost-wise less optimized queries when

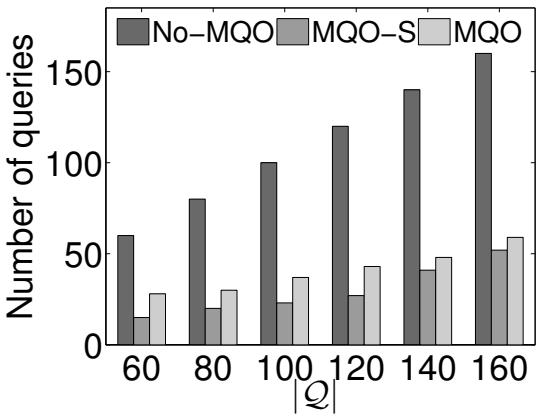


Figure 4.11: Vary $|\mathcal{Q}|$: $|\mathcal{Q}_{\mathsf{OPT}}|$

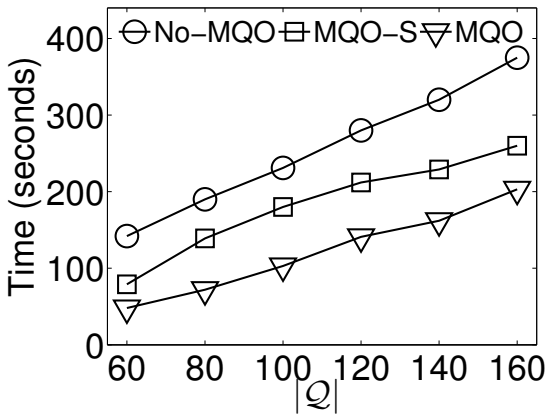


Figure 4.12: Vary $|\mathcal{Q}|$: time

compared with MQO, which is exactly the pattern in Figure 4.12.

Next, we further analyze the evaluating cost spent on clustering/rewriting the queries, and distributing the final results. In Figure 4.13, we report the clustering time, which includes both the bootstrapping k-means clustering and the hierarchical clustering that relies on finding common substructures. Notice that MQO requires more time than MQO-S. This is because (i) MQO involves an additional check on the selectivity; and (ii) queries with nonselective common subqueries are recycled into the pool of clusters by MQO, leading to more rounds of comparisons and thus a slower convergence. Contrarily, since the common subqueries rewritten by MQO-S are on average less selective, parsing and distributing these results inevitably requires more effort, as in Figure 4.14. Nevertheless, clustering and parsing times are a small fraction of the total evaluating cost (less than 2% in the worst case). In the remaining experiments, we only report the end-to-end evaluating cost.

Here, we study the impact on optimization of the size $|q_{\text{cmn}}|$ of the common subquery,

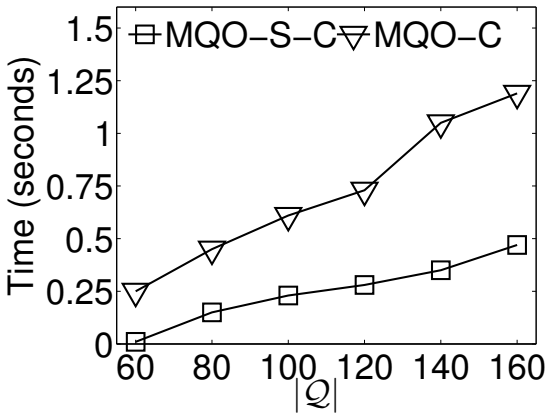


Figure 4.13: Clustering cost

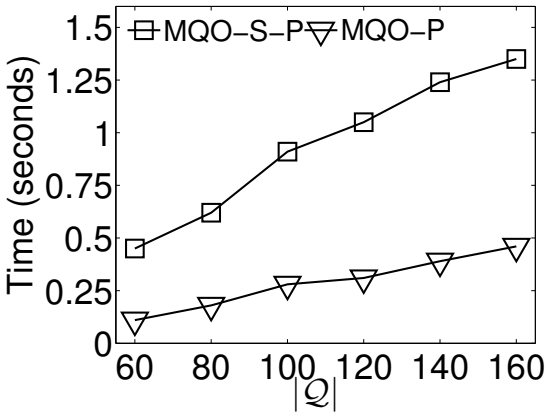


Figure 4.14: Parsing cost

i.e., the size of seed queries. At iteration i we make sure that for the queries in the same group of $\mathcal{Q}$, we have $|q_{\text{cmn}}| = i$. Figure 4.15 shows the number of optimized queries generated by MQO-S and MQO. Notice that the number of optimized queries is reduced (optimization improves) as $|q_{\text{cmn}}|$ increases. This is because, as the maximum size of each query is kept constant, the more $|q_{\text{cmn}}|$ increases, the more the generated queries become similar (less randomness in query generation). Therefore, more queries are clustered and optimized together. Like before, MQO-S is more aggressive and results in less queries compared to MQO. However, like before, Figure 4.16 shows that MQO is *always* better and results in optimized queries whose evaluation time is half less than MQO-S and up to 75% less than No-MQO. Notice that for small values of $|q_{\text{cmn}}|$, MQO-S performs worse than No-MQO. Intuitively, the more selective GP is in a **Type 2** optimized query, the less *work* a SPARQL query engine needs to do to evaluate the GP_{OPT} terms in the OPTIONAL of the query. MQO-S relies only on the structural similarity, while ignoring predicate selectivity, which negatively influences the overall evaluation time for the optimized query to the point that any benefits from the optimization are alleviated by the extra cost of evaluating the OPTIONAL terms.

MQO combines structured and cost optimization and does not suffer from these limitations. This is evident in Figure 4.17, which plots the percentage of the evaluation time of the optimized query that is spent evaluating q_{cmn}. By carefully selecting the common subquery q_{cmn}, MQO results in optimized queries whose evaluation time goes mostly (more than 90%) into evaluating q_{cmn} (while less than 10% goes to evaluating OPTIONAL terms). In contrast, MQO-S results in queries whose large extent of evaluation time goes into evaluating OPTIONAL terms (when $|q_{\text{cmn}}| = 1$ this is almost 30%). Things improve for MQO-S as the size of q_{cmn} increases, but still MQO retains the advantage of selecting substructures not

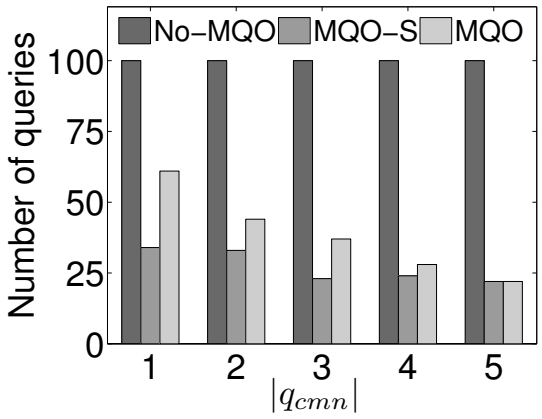


Figure 4.15: Vary $|q_{\text{cmn}}|$: $|\mathcal{Q}_{\text{OPT}}|$

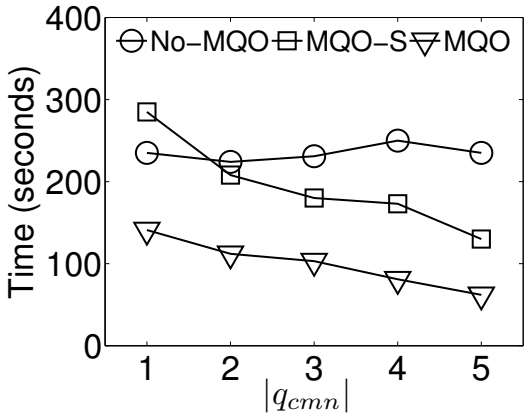


Figure 4.16: Vary $|q_{\text{cmn}}|$: time

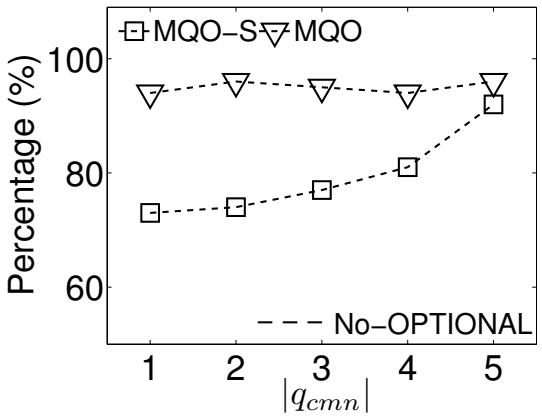


Figure 4.17: Evaluating q_{cmn}

just based on their size, but also on their selectivity, and therefore, overall evaluation times are still much better.

In Figures 4.18 and 4.19, we analyze the impact of the number κ of seed queries on the optimization, by varying κ from 5 to 10. Figure 4.18 shows that as κ increases, less queries can be optimized by both MQO-S and MQO, which resulted in more rewritten queries. Not surprisingly, a larger κ increases query diversity and reduces the potential for optimization. This affects evaluation times, but MQO is still the best of the three.

In Figures 4.20 and 4.21, we study the impact of query size, which we increase from 5 to 9 predicates in GP of a query Q. For this experiment we *keep the* $|q_{cmn}|/|\mathsf{Q}|$ *a rough constant and equal to* 0.5. So, the increase in query size does not result in a significant change in query overlap (or potential for optimization). Since the size of a query increases, there is higher chance for the query generator to assign it a selective predicate, which in turn affects the evaluation times. As a result, Figure 4.21 shows that the evaluation time of No-MQO decreases with the query size. Clearly, MQO still provides savings in evaluation time, ranging from 40% to 70%.

We study the impact of the minimum predicate selectivity in q_{cmn} (seed query), by varying $\alpha_{min}(q_{\text{cmn}})$ from 0.1% to 4%. As Figure 4.22 shows, selectivity has minimal impact for MQO-S, which ignores evaluation costs, but noticeable impact in MQO. As selectivity is reduced, the number of optimized queries increases (less optimization) since MQO increasingly rejects optimizations that lead to more expensive (nonselective) common subqueries. While reduced selectivity increases the evaluation time of queries for all algorithms (Figure 4.23), MQO still achieves between 10% and 50% savings in evaluation

times.

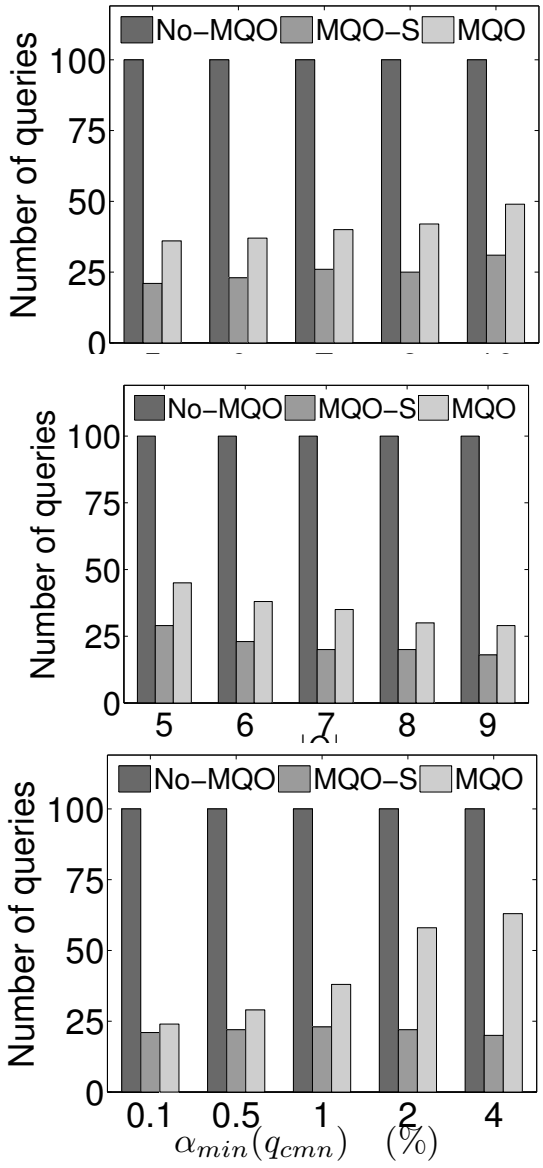


Figure 4.22: Vary α_{min}: $|\mathcal{Q}_{\mathsf{OPT}}|$

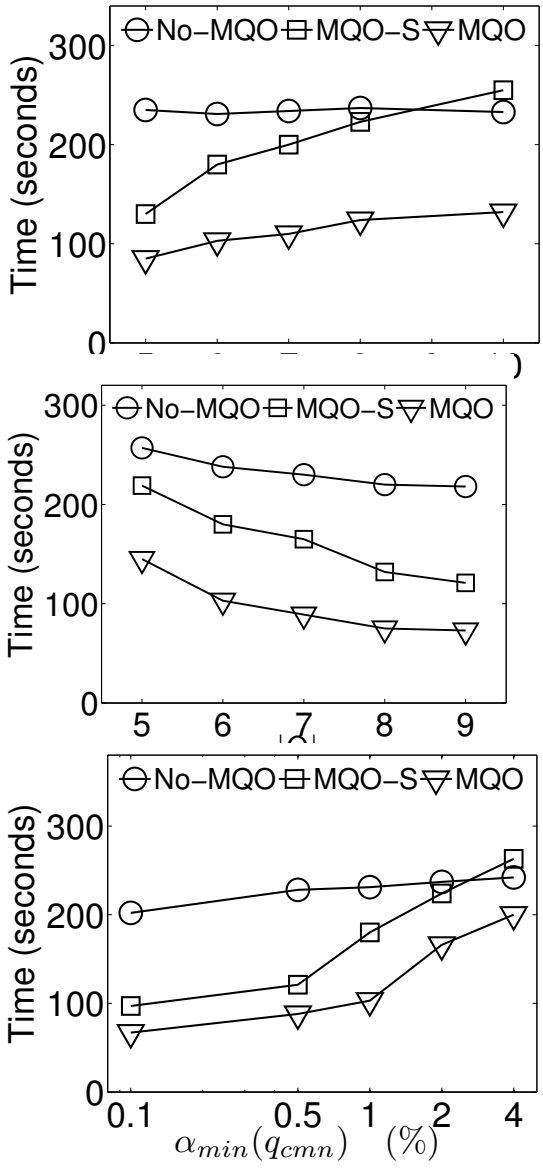


Figure 4.23: Vary α_{min}: time

While changing minimum selectivity has an impact on deciding the substructure that forms q_{cmn}, maximum selectivity mostly affects the cost of evaluating the (nonseed) OPTIONAL terms. Here, we vary the maximum selectivity for predicates in a query, $\alpha_{max}(\mathsf{Q})$, from 0.1% to 4%. Like before, Figure 4.24 shows that the number of optimized queries is almost unaffected for MQO-S. Unlike the previous experiment, this number is also unaffected for algorithm MQO since the change in selectivity concerns OPTIONAL predicates and thus has less of an effect in the generation of optimized queries. Figure 4.25 shows that both MQO-S and MQO outperform No-MQO, with MQO achieving a *minimum* of 50% savings. Again, notice that when MQO-S chooses nonselective predicates for optimization, evaluation times quickly degrade to No-MQO as when $\alpha_{max}(\mathsf{Q}) > 1\%$.

We investigate the impact of dataset size $|\mathsf{D}|$ on the optimization results, by varying $|\mathsf{D}|$ from 3M to 9M triples. While this does not affect the number of rewritings of $\mathcal{Q}$, it clearly affects evaluation times, as shown in Figures 4.26. Notice that MQO consistently has a *minimum* of 50% (achieving up to 65%) savings.

In Section 4.3, we extrapolate that the evaluation cost of a **Type 2** query is inversely correlated with the estimated cost of GP, *i.e.,* the minimum selectivity of its triple patterns. This is indeed a reasonable approximation in practice. As shown in Figure 4.23, reduced minimum selectivity in the common subquery GP would incur higher evaluation cost for **Type 2** queries. Similarly, both the number of OPTIONALs and the cost of the query pattern of each $\mathsf{GP}_{\mathsf{OPT}}$ are indispensable factors in determining the value of Δ, as shown respectively in Figure 4.19 and Figure 4.16. However, we observed that when the cost of GP is small (being selective), Δ would be a trivial value and $\mathsf{Cost}(\mathsf{Q})$ is mostly credited to the evaluation of GP. This is clearly shown in Figure 4.17 that when GP is selective, the dominant cost is contributed by evaluating GP (more than 90%) with the rest factors being almost irrelevant.

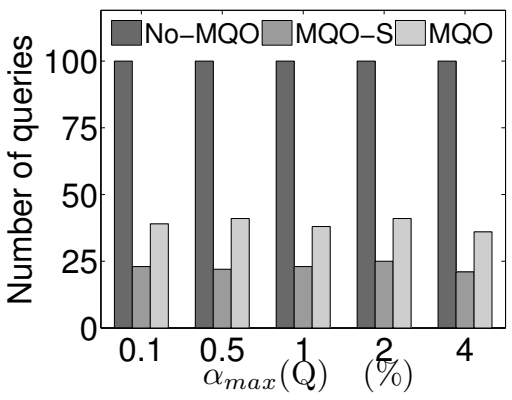


Figure 4.24: Vary α_{max}: $|\mathcal{Q}_{\mathsf{OPT}}|$

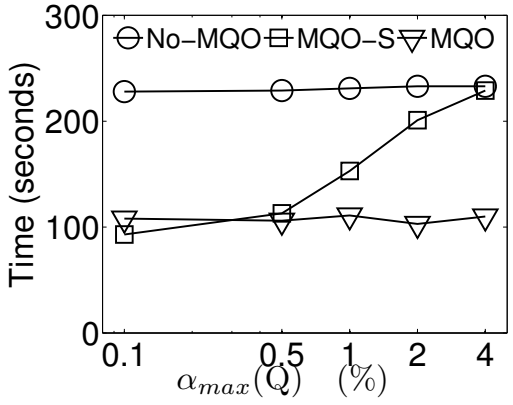


Figure 4.25: Vary α_{max}: time

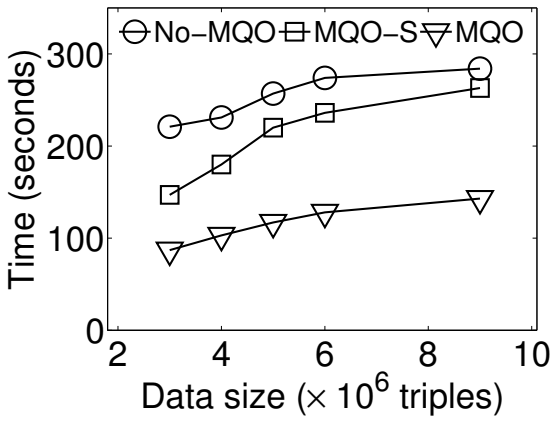


Figure 4.26: Varying |D|

This suggests that when dealing with a selective GP, a possibly good approximation of Cost(Q) can set $\Delta \simeq 0$. This observation also motivates us to choose a selective GP in rewriting. In practice, this simple cost model and its approximation give excellent cost estimation in MQO.

Up to now, all results reported were performed with Jena TDB. Using the same queries and parameters, we also ran the experiments on Virtuoso and Sesame native, to evaluate the desired property of store independence. In general, the results from Virtuoso and Sesame are consistent with what we observed in Jena TDB (see Figures 4.27–4.32), when we used the same setup as that in the experiments for Jena TDB, and varied values of one parameter while using default values for all other parameters. The proposed optimization algorithm, MQO, significantly reduces the evaluation time of multiple SPARQL queries on both stores. In particular, we consistently observed that the cost-based optimization can remarkably improve the performance in almost all experiments, leading to a 40%–75% speedup compared to No-MQO on both Virtuoso and Sesame. For example, using the same setting and optimized queries as Figure 4.12 where we vary the number of queries in a batch $\mathcal{Q}$, Figures 4.27(a) and 4.27(b) report the results from Virtuoso and Sesame. It is clear that MQO consistently outperforms MQO-S and No-MQO, leading to savings of 50%–60% across engines. Similarly, in the experiment that studies the impact of minimum selectivity in q_{cmn}, *i.e.,* Figure 4.28(a) and Figure 4.28(b), reducing the minimum selectivity of q_{cmn} results in increasing evaluation times for all algorithms. While MQO-S is sensitive to such variance since it does not proactively take cost into account, MQO still achieves 40%–75% savings in evaluation times.

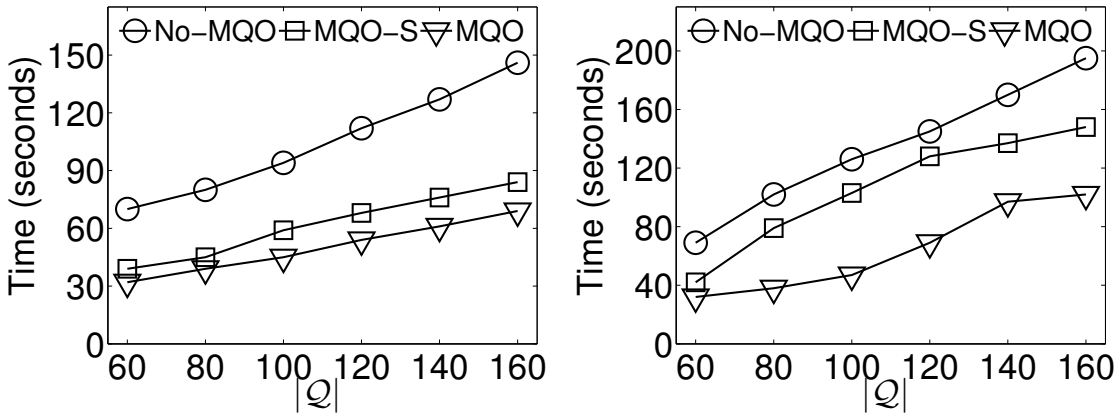


Figure 4.27: Vary $|\mathcal{Q}|$: evaluation time (a) Virtuoso and (b) Sesame

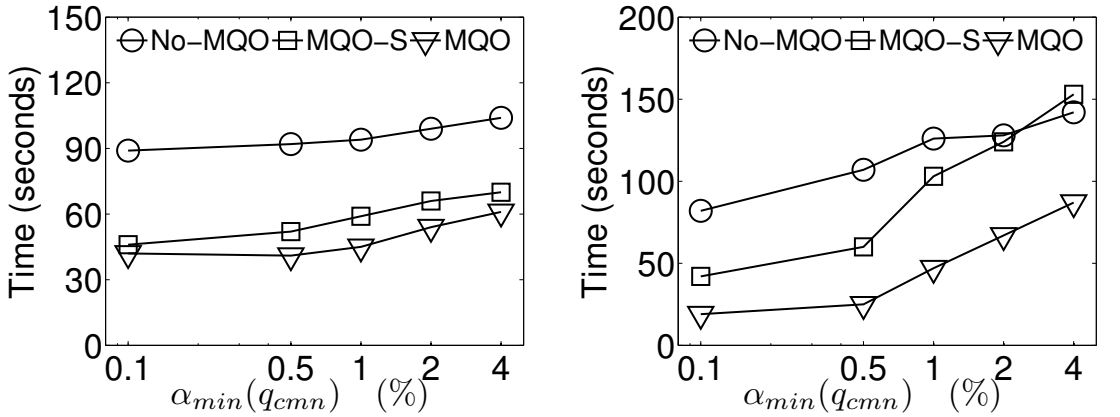


Figure 4.28: Vary $\alpha_{min}(q_{\text{cmn}})$: evaluation time (a) Virtuoso and (b) Sesame

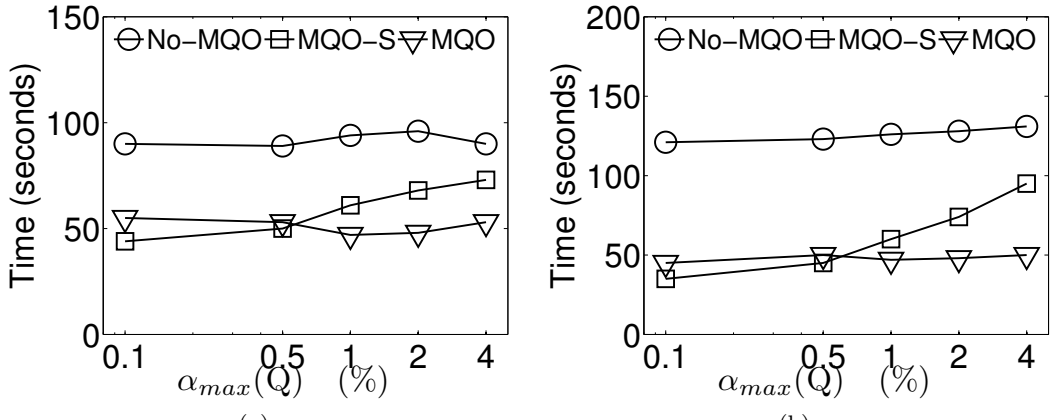


Figure 4.29: Vary α_{max}(Q): evaluation time (a) Virtuoso and (b) Sesame

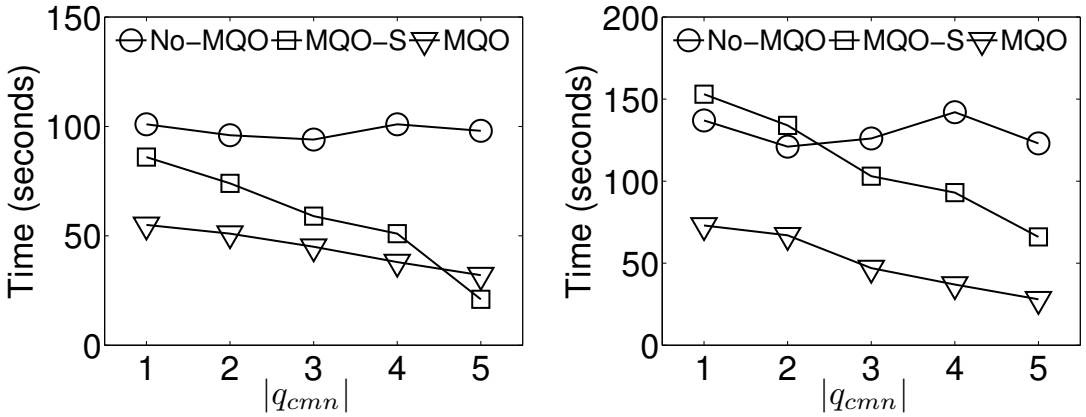


Figure 4.30: Vary $|q_{\text{cmn}}|$: evaluation time (a) Virtuoso and (b) Sesame

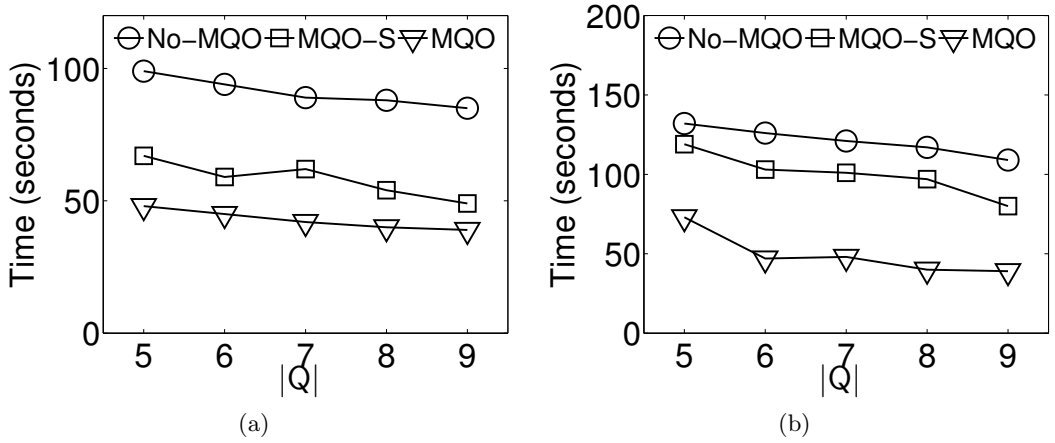


Figure 4.31: Vary |Q|: evaluation time (a) Virtuoso and (b) Sesame

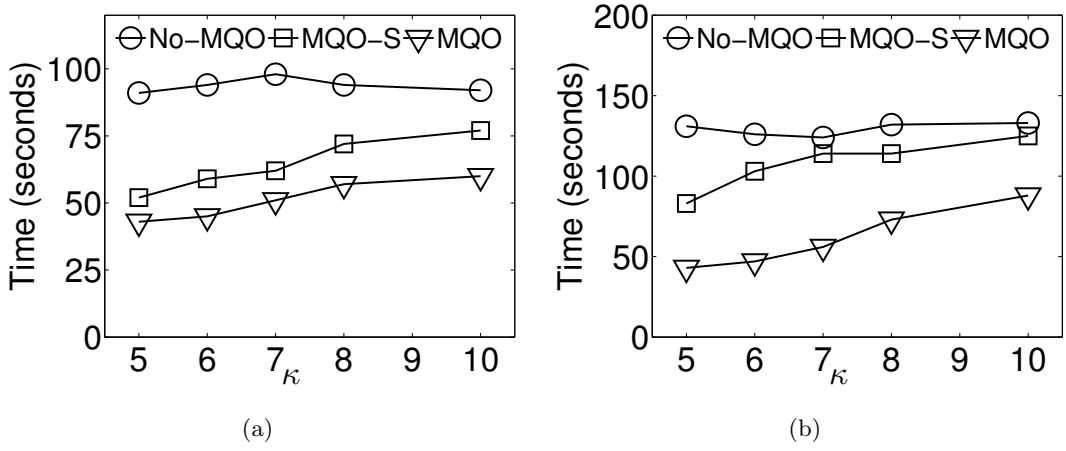


Figure 4.32: Vary κ: evaluation time (a) Virtuoso and (b) Sesame

4.6 Related Work

The problem of multiquery optimization has been well studied in relational databases [86, 96, 100, 102, 118]. The main idea is to identify the common subexpressions in a batch of queries. Global optimized query plans are constructed by reordering the join sequences and sharing the intermediate results within the same group of queries, therefore minimizing the cost for evaluating the common subexpressions. The same principle was also applied in [96], which proposed a set of heuristics based on dynamic programming to deal with nested subexpressions. There has also been studies on identifying common expressions [46, 115] with complexity analysis of MQO; the general MQO problem for relational databases is NP-hard. Even with heuristics, the search space for individual candidate plans and their combinatorial hybrid (*i.e.,* the global plan) is often astronomical [96]. In light of the hardness, [96] proposed some heuristics that were shown to work well in practice; however, those heuristics were proposed to work inside query optimizers (*i.e.,* engine dependent), and are only applicable when the query plans are expressible as AND-OR DAGs. Dalvi *et al.* [41] considered pipelining intermediate results to avoid unnecessary materialization. In addition, Diwan *et al.* [42] studied the issue of scheduling and caching in MQO. A cache-aware heuristics was proposed in [82] to make maximal use of the buffer pool.

All of the above work focuses on MQO in the relational case. MQO has also been studied on semistructured data. Hong *et al.* [55] considered concurrent XQuery join optimization in publish/subscribe systems. Join queries were mapped to a precomputed tree structure, called query template, for evaluation. Due to the limitation of the precomputed templates, only basic join structures were supported. Another work by Bruno *et al.* [31] in XML studied navigation and index-based path MQO. Unlike the MQO problem in relational and SPARQL cases, path queries can be encoded into a prefix tree where common prefixes share the same branch from the root. This nature provides an important advantage in optimizing concurrent path queries. Nevertheless, the problem of multiquery join optimization was not addressed. The work of Kementsietsidis *et al.* [64] considered a *level-wise merging* of query trees based on the tree depth of edges in a distributed setting, with the main objective to minimize the communication cost in evaluating tree-based queries in a distributed setting.

In summary, existing MQO techniques proposed in relational and XML cases cannot be trivially extended to work for SPARQL queries over RDF data (which can be viewed as SPJ queries over generic graphs), since relational techniques need to reside in relational query optimizers, which cannot be assumed in the management of RDF data, and notions like prefix-tree and tree depth do not apply to generic graphs. Also, there has been work on

query optimization for *single* SPARQL query [80, 98, 103], as well as *single* graph query optimization for general graph databases [117]. However, to the best of our knowledge, our work is the first to address MQO for SPARQL queries over RDF data.

4.7 Conclusion

We studied the problem of multiquery optimization in the context of RDF and SPARQL. Our optimization framework, which integrates a novel algorithm to efficiently identify common subqueries with a fine-tuned cost model, partitions input queries into groups and rewrites each group of queries into equivalent queries that are more efficient to evaluate. We showed that our rewriting approach to multiquery optimization is both sound and complete. Furthermore, our techniques are store-independent and therefore can be deployed on top of any RDF store without modifying the query optimizer. Useful extensions on handling more general SPARQL queries are also discussed. Extensive experiments on different RDF stores show that the proposed optimizations are effective, efficient, and scalable. An interesting future work is to extend our study to generic graph queries over general graph databases.

So far, we have studied two closely related problems regarding SPARQL query rewriting and optimization. Querying RDF data with SPARQL assumes users have good understandings for the schema of the data. However, when interacting with massive RDF data sets that are constantly evolving (for instance, DBpedia [3]), the schemas are not always available. In such a scenario, a more attractive tool to investigate the semantics and the structure of the data and perform data integration is keyword search. Next, we are going to study the problem of keyword search on large linked data and address the new challenges arise in the context of RDF data.

CHAPTER 5

KEYWORD SEARCH ON RDF DATA

5.1 Introduction

Keyword search is an important tool for exploring and searching large data corpuses whose structure is either unknown, or constantly changing. So, keyword search has already been studied in the context of relational databases [17, 35, 56, 78], XML documents [36, 105], and more recently over graphs [53, 63] and RDF data [48, 109]. However, existing solutions for RDF data have limitations. Most notably, these solutions suffer from: (i) returning incorrect answers, *i.e.,* the keyword search returns answers that do not correspond to real subgraphs or misses valid matches from the underlying RDF data; or (ii) inability to scale to handle typical RDF datasets with tens of millions of triples. Consider the results from two representative solutions [53, 109], as shown in Figures 5.1 and 5.2. Figure 5.1 shows the query results on three different datasets using the solution specifically designed for RDF in [109], the *Schema* method. While this solution may perform well on datasets that have regular topological structure (*e.g.,* DBLP), it returns incorrect answers for others (*e.g.,* LUBM [50] *etc.*) when compared to a naive but *Exact* method. On the other hand, classical techniques [53] proposed for general graphs can be used for RDF data, but they assume a distance matrix built on the data, which makes it prohibitively expensive to apply to a large RDF dataset, as shown in Figure 5.2.

Motivated by these observations, we present a comprehensive study to address the keyword search problem over big RDF data. Our goal is to design a scalable and exact solution that handles realistic RDF datasets with tens of millions of triples. To address the scalability issues, our solution builds a new, succinct, and effective summarization structure from the underlying RDF graph based on its *type*s. Given a keyword search query, we use the summarization structure to prune the search space, which leads to much better efficiency compared to approaches that process queries directly on the RDF graph. Figure 5.3 shows a tiny fraction of the DBpedia data that we will use as a running example throughout this chapter. To summarize, our contributions are as follows.

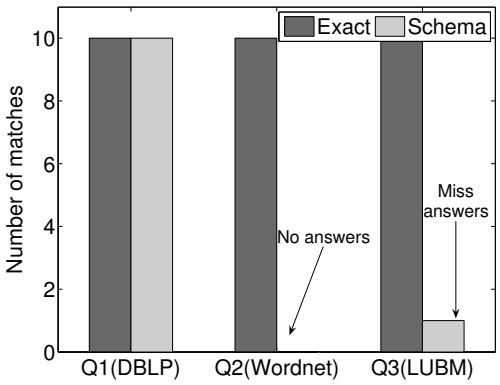


Figure 5.1: *Schema* method in [109]

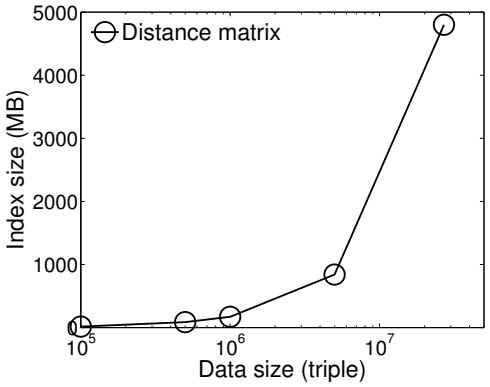


Figure 5.2: Distance matrix method in [53]

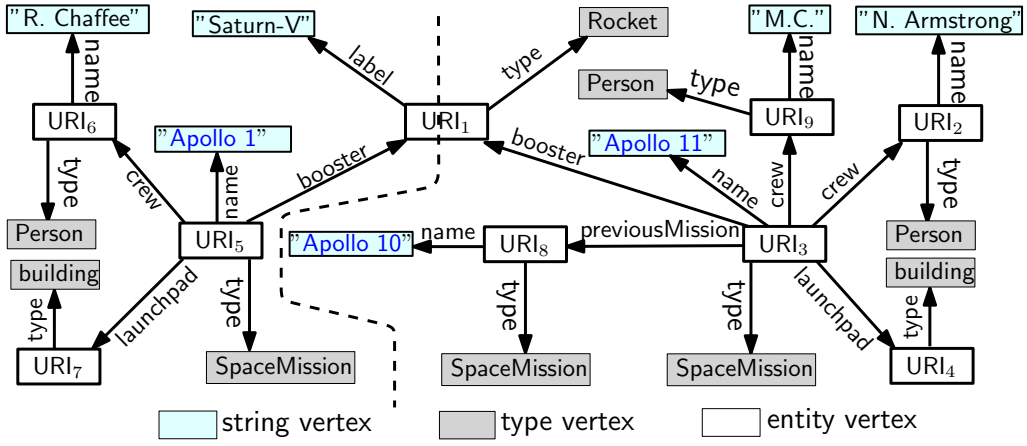


Figure 5.3: Keywords in a small sample from the DBpedia dataset

• We identify and address limitations in the existing, state-of-the-art methods for keyword search in RDF data [109]. We show that these limitations could lead to incomplete and incorrect answers in real RDF datasets. We propose a new, correct baseline solution based on the backward search idea.

• We develop efficient algorithms to summarize the structure of RDF data, based on the *types* in RDF graphs, and use it to speed up the search. Compared to previous works that also build summarization [48, 109], our new summarization leverages on completely different intuitions, and it is scalable for large disk-resident RDF data and lends significant pruning power without sacrificing the soundness of the result. Further, our summarization is light-weight and updatable.

• Our experiments on benchmark and large real RDF datasets show that our techniques are much more scalable and efficient in correctly answering realistic keyword search queries

than the existing methods.

In what follows, we formulate the keyword search problem on RDF data in Section 5.2, present our solutions in Sections 5.3 to 5.6, survey related work in Section 5.7, show experimental results in Section 5.8, and conclude in Section 5.9. Table 5.1 lists the frequently used symbols.

5.2 Preliminaries and Problem Statement

5.2.1 Ontology in RDF Data

We have given a brief overview for RDF data in Chapter 2. W3C has also provided a set of unified vocabularies (as part of the RDF standard) to encode the rich semantics, *e.g.,* RDFS/OWL. Among these, the rdfs:type predicate (or type for short) is particularly useful to our problem (see Section 5.4), since it provides a classification of vertices and edges of an RDF graph into different groups. For instance, in Figure 5.3, the entity URI$_3$ has type SpaceMission. Formally, we view an RDF dataset as an RDF graph $G = (V, E)$ where

• V is the union of disjoint sets, V_E, V_T, and V_W, where V_E is the set of entity vertices (*i.e.,*URIs), V_T is the set of type vertices, and V_W is a set of keyword vertices.

• E is the union of disjoint sets, E_R, E_A, and E_T, where E_R is the set of entity-entity edges (*i.e.,* connecting two vertices in V_E), E_A is the set of entity-keyword edges (*i.e.,*

Table 5.1: Frequently used notations

Symbol	Description
$G\{V, E\}$	the condensed view of an RDF graph.
$\mathcal{A}(q)$	top-k answers for a query.
r	answer root.
w_i	the i-th keyword.
$d(x, y)$	graph distance between node x and node y.
$C(q)$	a set of candidate answers.
α	used to denote the α-hop neighborhoods.
g	an answer subgraph of G.
S	the summaries of $\mathcal{P}$.
W_i	the set of nodes containing keyword w_i.
$\mathcal{P}$	partitions.
M	for bookkeeping the candidate answers.
$h(v, \alpha)$, h	the α-hop neighborhoods of v, a partition.
$h_t(v, \alpha)$, h_t	the covering tree of $h(v, \alpha)$, a covering tree.
$\mathcal{S}$	a path represented by a sequence of partitions.
d_l, d_u	the lower and upper bounds (for a path).
σ	a one-to-many mapping in converting h to h_t.
Σ	a set of σ's from a partition h.

connecting an entity to a keyword), and E_T is the set entity-type edges (*i.e.,* connecting an entity to a type).

For example, in Figure 5.3, all gray vertices are type vertices while entity vertices are in white. Each entity vertex also has associated keyword vertices (in cyan). The division on vertices results in a corresponding division on the RDF predicates, which leads to the classification of the edge set E discussed earlier. Clearly, the main structure of an RDF graph is captured by the entity-entity edges represented by the set E_R. As such, an alternative view is to treat an entity vertex and its associated type and keyword vertices as *one vertex*. For example, the entity vertices URI_5, URI_1, and URI_3 from Figure 5.3, with their types and keywords, can be viewed as the structure in Figure 5.4.

In general, for an RDF graph $G = \{V, E\}$, we will refer to this as the *condensed view* of G, denoted as $G_c = \{V'_E, E_R\}$. While $|V'_E| \equiv |V_E|$, every vertex $v' \in V'_E$ contains not only the entity value of a corresponding vertex $v \in V_E$, but also the associated keyword(s) and type(s) of v. For ease of presentation, hereafter, we associate a *single* keyword and a *single* type to each entity. Our techniques can be efficiently extended to handle the general cases. Also for simplicity, hereafter, we use $G = \{V, E\}$ to *represent the condensed view* of an RDF graph.

5.2.2 Problem Statement

Given an RDF graph $G = \{V, E\}$, for any vertex $v \in V$, let $w(v)$ be the keyword stored in v. Formally, a keyword search query q in an RDF data $G = \{V, E\}$ is defined by m unique keywords $\{w_1, w_2, \ldots, w_m\}$. A set of vertices $\{r, v_1, \ldots, v_m\}$ from V is a *qualified candidate* when:

- $r \in V$ is called the *root answer node*, which is reachable by $v_i \in V$ for $i \in [1, m]$
- $w(v_i) = w_i$.

If we define the answer for q as $\mathcal{A}(q)$ and the set of all qualified candidates in G with respect to q as $C(q)$, then

$$\mathcal{A}(q) = \min_{g \in C(q)} s(g), \text{ and } s(g) = \sum_{\substack{r, v_i \in g, i = \\ 1..m}} d(r, v_i) \tag{5.1}$$

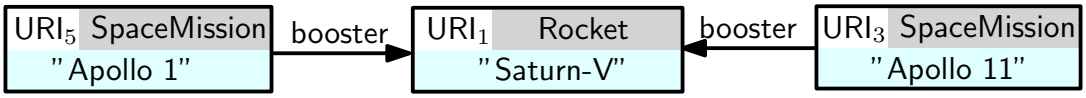


Figure 5.4: Condensed view: combining vertices

where $d(r, v_i)$ is the *graph distance* between vertices r and v_i (when treating G as *an undirected graph*). Intuitively, this definition looks for a subgraph in an RDF graph that has minimum length to connect all query keywords from a root node r. In prior works concerning keyword search in RDF data, the graph distance of $d(v_1, v_2)$ is simply the shortest path between v_1 and v_2 in G, where each edge is assigned a weight of 1 (in the case of general graph [53], the weight of each edge could be different). Note that if v_1 and v_2 belong to disconnected parts of G, then $d(v_1, v_2) = +\infty$. Also note that this metric (*i.e.,* Equation 5.1) is proposed by [53] and has been used by prior work on keyword search in RDF data [48, 109].

This definition has a top-k version, where the query asks for the top k qualified candidates from $C(q)$. Let the score of a qualified candidate $g \in C(q)$ be defined as $s(g)$ in Equation (5.1); then we can rank all qualified candidates in $C(q)$ in an ascending order of their scores, and refer to the ith ranked qualified candidate as $A(q, i)$. The answer to a top-k keyword search query q is an *ordered set* $\mathcal{A}(q, k) = \{A(q, 1), \ldots, A(q, k)\}$. $\mathcal{A}(q)$ is a special case when $k = 1$, and $\mathcal{A}(q) = \mathcal{A}(q, 1)$. Lastly, we adopt the same assumption in the prior works [53, 109] that the answer roots in $\mathcal{A}$ are *distinct*.

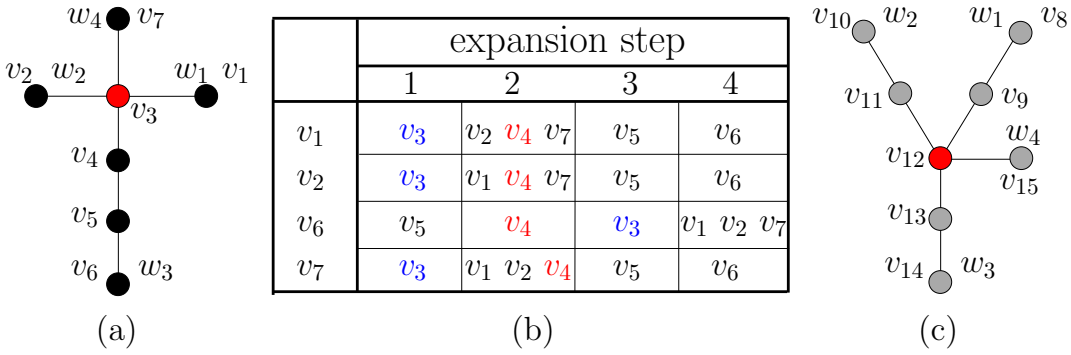
5.3 The Baseline Method

A baseline solution is based on the *"backward search"* heuristic on generic graphs [26, 61]. Intuitively, the "backward search" (for the root node r) starts simultaneously from each vertex in the graph G that corresponds to a query keyword, and expands to its neighboring nodes recursively until a candidate answer is generated. A *termination condition* is used to determine whether the search is complete.

The state-of-the-art keyword search method on RDF graphs [109] has applied the backward search idea. Their termination condition is to stop the search whenever the expansions originating from m vertices $\{v_1, \ldots, v_m\}$ (each corresponding to a distinct query keyword) meet at a node r *for the first time*, where $\{r, v_1, \ldots, v_m\}$ is returned as the answer. Unfortunately, this termination condition is *incorrect*.

5.3.1 A Counter Example

Consider the graph in Figure 5.5(a) and a top-1 query $q = \{w_1, w_2, w_3, w_4\}$. The steps for the four backward expansions performed on Figure 5.5(a) are shown in Figure 5.5(b). Using the above termination condition, the backward expansions from the four vertices $\{v_1, v_2, v_6, v_7\}$ covering the query keywords $\{w_1, w_2, w_3, w_4\}$ meet for the first time in the second iteration, so the candidate answer $g : -\{r{=}v_4, v_1, v_2, v_6, v_7\}$ is returned and $s(g) = 8$. However, if we continue to the next iteration, the four expansions will meet again at v_3,



(b)

	expansion step			
	1	2	3	4
v_1	v_3	v_2 v_4 v_7	v_5	v_6
v_2	v_3	v_1 v_4 v_7	v_5	v_6
v_6	v_5	v_4	v_3	v_1 v_2 v_7
v_7	v_3	v_1 v_2 v_4	v_5	v_6

Figure 5.5: Backward search

with $g = \{r{=}v_3, v_1, v_2, v_6, v_7\}$ and $s(g') = 6$, which is the optimal answer. One may argue that the subgraph covering the query keywords is still correctly identified even though the procedure fails to enforce the cost metric defined in Equation (5.1). However, if we augment the aforementioned RDF dataset by including the data graph in Figure 5.5(c), *i.e.,* an RDF dataset with two disjoint graphs – Figure 5.5(a) and (c), then by applying the same procedure, the (wrong) answer would be $g'' = \{r{=}v_{12}, v_8, v_{10}, v_{14}, v_{15}\}$ (*i.e.,* Figure 5.5(c)) with a score $s(g'') = 7 < s(g)$. This is clearly wrong as g' (with a cost of 6) should have been identified as the top-1 answer for q instead of g'. Furthermore, we will show that even if we fix this error in the terminating condition, this method [109] may still return incorrect results due to the limitations in the summary it builds, as shown in Figure 5.1.

5.3.2 The Correct Termination

We give the correct termination condition for the backward search on RDF graphs and the complete algorithm is shown in Algorithm 4.

We first introduce the data structures. Given $q = \{w_1, \ldots, w_m\}$ and a (condensed) RDF graph $G = \{V, E\}$, let W_i be the set of vertices in V containing the keyword w_i (line 1). We initialize m empty priority queues (*e.g.,* min-heaps) $\{a_1, ..a_m\}$, one for each query keyword (line 1). We also maintain a set M of elements (line 2), one for each distinct node we have explored so far in the backward expansion to track the state of the node, *i.e.,* what keywords are reachable to the node and their best known distances. In what follows, we use $M[v]$ to indicate the bookkeeping for the node v. Specifically, in each element of M, we store a list of m (`vertex`, `distance`) pairs. A (`vertex`, `distance`) pair in the jth entry of $M[v]$ indicates a (shortest) path from `vertex` that reaches v in `distance` hops and it is the shortest possible path starting from any instance of w_j (recall that there could be multiple copies of w_j in G). Next, we also use $M[v][j]$ to indicate the jth pair in $M[v]$. For instance, in Figure 5.5(a),

Algorithm 4: BACKWARD

Input: $q = \{w_1, w_2, \ldots, w_m\}$, $G = \{V, E\}$
Output: top-k answer $\mathcal{A}(q)$

```
1  Initialize {W_1, ..W_m} and m min-heaps {a_1, ..a_m};
2  M ← ∅;                                          // for tracking potential C(q)
3  for v ∈ W_i and i = 1..m do
4      for ∀u ∈ V and d(v,u) ≤ 1 do
5          a_i ⇐ (v, p ← {v,u}, d(p) ← 1) ;                          // enqueue
6          if u ∉ M then M[u] ← {nil, ..(v,1).., nil};
7          else M[u][i] ← (v,1);          ↑the ith entry
8  while not terminated and A not found do
9      (v, p, d(p)) ← pop(arg min_{i=1}^{m} {top(a_i)});
10     for ∀u ∈ V and d(v,u) = 1 and u ∉ p do
11         a_i ⇐ (u, p ∪ {u}, d(p) + 1);
12         update M the same way as in lines 6 and 7;
13 return A (if found) or nil (if not);
```

consider an element $M[v_3] = \{(v_1, 1), (v_2, 1), \text{nil}, (v_7, 1)\}$ in M. The entry indicates that v_3 has been reached by three expansions from vertices v_1, v_2, and v_7, containing keywords w_1, w_2, and w_4, respectively – each can reach v_3 in one hop. However, v_3 has not been reached by any expansion from any vertex containing w_3 yet.

With the structures in place, the algorithm proceeds in iterations. In the first iteration (lines 3-7), for each vertex v from W_i and every neighbor u of v (including v itself), we add an entry $(v, p \leftarrow \{v, u\}, d(p))$ to the priority queue a_i (entries are sorted in the ascending order of $d(p)$ where p stands for a path and $d(p)$ represents its length). Next, we look for the newly expanded node u in M. If $u \in M$, we simply replace $M[u][i]$ with $(v, d(\mathsf{p}))$ (line 7). Otherwise, we initialize an empty element for $M[u]$ and set $M[u][i] = (v, d(\mathsf{p}))$ (line 6). We repeat this process for all W_i's for $i = 1..m$.

In the jth ($j > 1$) iteration of our algorithm (lines 8-12), we pop the smallest top entry of $\{a_1..a_m\}$ (line 9), say an entry $(v, p = \{v, \ldots, u\}, d(p))$, from the queue a_i. For each neighboring node u' of u in G such that u' is not in p yet (*i.e.,* not generating a cycle), we push an entry $(v, p \cup \{u\}, d(p) + 1)$ back to the queue a_i (line 11). We also update M with u' similarly as above (line 12). This concludes the jth iteration.

In any step, if an entry $M[u]$ for a node u has no nil pairs in its list of m (`vertex`, `distance`) pairs, this entry identifies a candidate answer and u is a candidate root. Notice that due to the property of the priority queue (which implicitly enforces a BFS search from each keyword instance) and the the fact that all edges have a unit weight, the paths in $M[u]$ are the shortest paths to u from m distinct query keywords. Let g be the graph pieced by the

list of shortest paths in $M[u]$, and we have:

Lemma 3 *$g = \{r{=}u, v_{\ell_1}, \ldots, v_{\ell_m}\}$ is a candidate answer with $s(g) = \sum_{i=1}^{m} d(u, v_{\ell_i})$.*

Proof. By our construction for entries in M, $w(v_{\ell_i}) = w_i$ and $d(u, v_{\ell_i})$ is the length for shortest path from v_{ℓ_i} to u, which completes the proof. ■

A node v has not been fully explored if it has not been reached by at least one of the query keywords. Let V_t be the set of all not fully explored vertices, and the top entries from the m expansion queues (*i.e.,* min-heaps) $a_1, \ldots, a_m$ be $(v_1, p_1, d(p_1)), \ldots, (v_m, p_m, d(p_m))$. There are two cases to consider: (i) in order for an unseen vertex, *i.e.,* $v \notin M$, to be the answer root, the best possible cost is bounded by:

Lemma 4 *Let g_1 be the best possible candidate answer, with $v \notin M$ being the answer root of g_1. Then $s(g_1) > \sum_{i=1}^{m} d(p_i)$.*

Proof. Since v is not in M, indicating that v has not yet been included in any expansion path from any entries in these m queues, we need to expand at least one neighboring node to the end node in a path from at least one of these top m entries to possibly reach v. Furthermore, all m expansion queues sort their entries in ascending order of the distance of the corresponding paths; hence, any candidate answer using v as the root node must have at least a distance of $d(p_i) + 1$ to reach a vertex v' with $w(v') = w_i$. That shows $s(g_1) > \sum_{i=1}^{m} d(p_i)$, which completes the proof. ■

(ii) in the second case, consider $v \in M$ that has at least one nil entry. Clearly, $v \in V_t$. Let its list of $M[v]$ be $(v_{b_1}, d_1), \ldots, (v_{b_m}, d_m)$, and we have the following result:

Lemma 5 *Suppose the best possible candidate answer using such a v ($v \in M$ and $v \in V_t$) as the answer root is g_2, then*

$$s(g_2) > \sum_{i=1}^{m} f(v_{b_i}) d_i + (1 - f(v_{b_i})) d(p_i), \tag{5.2}$$

where $f(v_{b_i}) = 1$ if $M[v][b_i] \neq nil$, and $f(v_{b_i}) = 0$ otherwise.

Proof. When v_{b_i} is not nil, that means this vertex v has been reached by an expansion from a vertex v_{b_i} where $w(v_{b_i}) = w_i$ (*i.e.,*, $v_{b_i} \in W_i$), and $d_i = d(v_{b_i}, v)$. More importantly, d_i is the shortest possible distance from any vertex in W_i to reach v, since we expand paths in a_i in the ascending order of their distances.

When v_{b_i} is nil, that means no expansions initiated from any vertices from W_i have yet reached v. Following the same argument from the proof for Lemma 4, the shortest distance to reach any vertex in W_i from v is at least $d(p_i) + 1$.

Finally, by combining these two arguments, we can establish Equation (5.2). ■

Notice that in Lemma 5, if $M[v][b_i] \neq \text{nil}$, then $d(p_i) \geq d_i$ due to the fact that a_i is a min-heap. It follows that $s(g_2) \leq s(g_1)$.

5.3.3 The Termination Condition

These v's represent all nodes that have not been fully explored. For case (i), we simply let $s(g_1) = \sum_{i=1}^{m} d(p_i)$; for case (ii), we find a vertex with the smallest possible $s(g_2)$ value w.r.t. the RHS of (5.2), and simply denote its best possible score as $s(g_2)$.

Let g be the candidate answer we have identified so far in our algorithm with the kth smallest score; our search can safely terminate when $s(g) \leq \min(s(g_1), s(g_2)) = s(g_2)$. We denote this algorithm as the BACKWARD method. By Lemmas 3, 4, 5, we have Theorem 3 as follows.

Theorem 3 *The* BACKWARD *method finds the top-k answers* $\mathcal{A}(q, k)$ *for any top-k keyword query* q *on an* RDF *graph.*

Proof. This is a straightforward result by Lemma 3 and the termination conditions stated in Lemmas 4, 5. ■

5.4 Type-Based Summarization

The BACKWARD method is clearly not scalable on large disk-resident RDF graphs. For instance, the keyword "Armstrong" appears 269 times in our experimental DBpedia dataset, but only one is close to the keyword "Apollo 11", as in Figure 5.3. If we are interested in the smallest subgraphs that connect these two keywords, the BACKWARD method will initiate many random accesses to the data on disk, and has to construct numerous search paths in order to complete the search. However, the majority of them will not lead to any answers. Intuitively, we would like to reduce the input size to BACKWARD and apply BACKWARD only on the most promising subgraphs. We approach this problem by proposing a type-based summarization on the RDF data. The idea is that, by operating our keyword search initially on the summary (which is typically much smaller than the data), we can navigate and prune large portions of the graph that are irrelevant to the query, and only apply BACKWARD method on the smaller subgraphs that guarantee to find the optimal answers.

5.4.1 The Intuition for Summarization

The idea is to first induce partitions over the RDF graph G. Keywords in query will be first pieced up by partitions. The challenge lies on how to safely prune connections (of par-

titions) that will not result in any top-k answer. To this end, we need to calibrate the length of a path in the backward expansion that crosses a partition. However, maintaining the exact distance for every possible path is expensive, especially when the data are constantly changing. Therefore, we aim to distill an updatable summary from the distinct structures in the partitions such that any path length in backward expansion can be effectively estimated. The key observation is that neighborhoods in close proximity surrounding vertices of the same type often share *similar structures* in how they connect to vertices of other types. To illustrate, consider the condensed view of Figure 5.3. The graph in Figure 5.6(a) is common for the 1-hop neighborhoods of URI_3 and URI_5 with the type SpaceMission.

This observation motivates us to study how to build a type-based summarization for RDF graphs. A similar effort can be seen in [109], where a *single* schema is built for all the types of entities in the data. However, this is too restrictive as RDF data are known to be schema-less [43], *e.g.,* entities of the same type do not have a unified property conformance.

5.4.2 Outline and Preliminaries

Our approach starts by splitting the RDF graph into multiple, smaller partitions. Then, it defines a minimal set of common type-based structures that summarizes the partitions. Intuitively, the summarization bookkeeps the distinct structures from all the partitions. In general, the keyword search can benefit from the summarization in two perspectives. With the summarization,

• we can obtain the upper and lower bounds for the distance traversed in any backward expansion without constructing the actual path (Section 5.5); and

• we can efficiently retrieve *every* partition from the data by collaboratively using SPARQL query and any RDF store without explicitly storing the partition (Section 5.6).

Before defining the summarization, we introduce two notions from graph theory: *graph homomorphism* and *core*.

We first consider homomorphism across partitions. As in Figure 5.6(a), type vertices that are at close proximity are a good source to generate induced partitions of the data graph. However, if we were to look for such induced partitions that are exactly the same across the whole graph, we get a large number of them. Consider another type-based structure in Figure 5.6(b), which is extracted from 1-hop neighbors around the vertex URI_3 in Figure 5.3. Notice that the two graphs are different; however, Figure 5.6(a) is a substructure of Figure 5.6(b). We consider discovering such embeddings between the induced partitions, so that one template can be reused to bookkeep multiple structures.

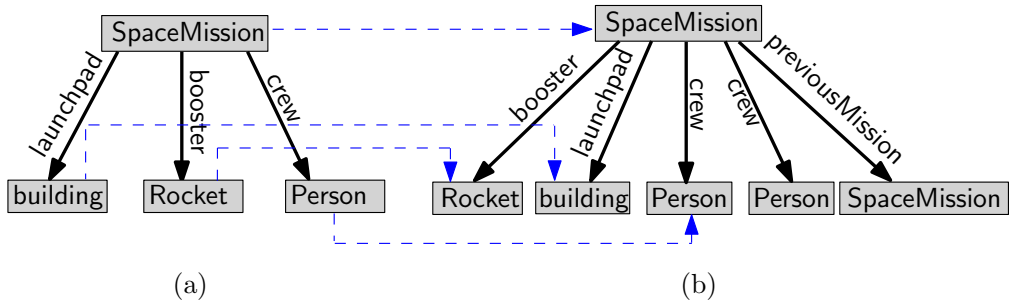


Figure 5.6: Graph homomorphism across summaries

Definition 2 *A graph homomorphism f from a graph $G = \{V, E\}$ to a graph $G' = \{V', E'\}$, written as $f : G \to G'$, is a mapping function $f : V \to V'$ such that (i) $f(x) = x$ indicates that x and $f(x)$ have the same type; and (ii) $(u, v) \in E$ implies $(f(u), f(v)) \in E'$ and they have the same label. When such an f exists, we say G is homomorphic to G'.*

Intuitively, embedding G to G' will not only reduce the number of structures we need to keep but also preserve any path from G in G', as shown by the homomorphism in Figure 5.6 (more expositions in Section 5.5). Finally, notice that homomorphism is transitive, *i.e.*, $G \to G'$ and $G' \to G''$ imply that $G \to G''$.

A *core* is a graph that is only homomorphic to itself, but not to any one of its proper subgraphs.

Definition 3 *A* core *c of a graph G is a graph with the following properties: there exists a homomorphism from c to G; there exists a homomorphism from G to c; and c is minimal (in the number of vertices) with these properties.*

Intuitively, a core of a partition succinctly captures how different types of entities connect to each other. For example, the partition in Figure 5.7(b) is converted to its core in Figure 5.7(a) by eliminating one of its branches.

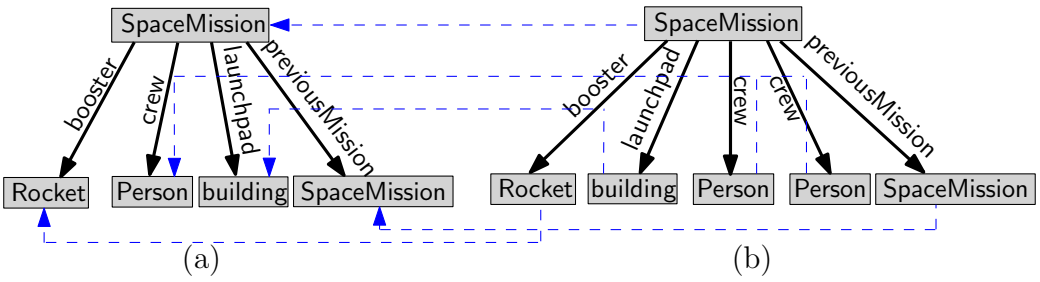


Figure 5.7: Build a core (a) from (b)

5.4.3 Partition

The summarization starts with splitting the data graph into smaller but semantically similar and *edge disjoint* subgraphs. Given our observation that nodes with the same type often share similar type-neighborhoods, we induce a distinct set of partitions for G based on the types in G, using small subgraphs surrounding vertices of the same type. Our partitioning algorithm treats an input RDF dataset as a *directed graph* G concerning only the type information, *i.e.,* we use the condensed view of an RDF graph. For any vertex that does not have a type specified by the underlying dataset, we assign an universal type NA to them. Notice that graph partitioning is a well-studied problem in the literature; here, we do not propose any new technique in that respect but rather focus on how to build semantically similar partitions for our purpose. The partitioning algorithm is shown in Algorithm 5.

Algorithm 5: Partition

Input: $G = \{V, E\}$, α
Output: A set of partitions in $\mathcal{P}$

```
1 Let 𝒯 = {T_1, ..., T_n} be the distinct types in V;
2 𝒫 ← ∅;
3 for T_i ∈ 𝒯 do
4     for v ∈ V_i do
5         extract h(v, α), the α neighborhood of v;
6         𝒫 ← 𝒫 ∪ h(v, α);
7 return 𝒫;
```

In Algorithm 5, suppose G has n distinct number of types $\{\mathsf{T}_1, \ldots, \mathsf{T}_n\}$, and we use the set V_i to represent the vertices from V that have a type T_i (line 4). We define the *α-neighborhood surrounding a vertex*, where α is a parameter used to produce a set of edge disjoint partitions $\mathcal{P}$ over G. Formally, for any vertex $v \in V$ and a constant α, the α-neighborhood of v is the subgraph from G obtained by expanding v with α hops in a breadth-first manner, denoted as $h(v, \alpha)$ (line 5), but subject to the constraint that the expansion only uses edges that have not been included by any partition in $\mathcal{P}$ yet. We define the i-hop neighboring nodes of v as the set of vertices in G that can be connected to v through a *directed path* with exactly i directed edges. Note that since we are using directed edges, it is possible the i-hop neighboring nodes of v is an empty set. Clearly the nodes in $h(v, \alpha)$ are a subset of the α-hop neighboring nodes of v (since some may have already been included in another partition).

To produce a partition $\mathcal{P}$, we initialize $\mathcal{P}$ to be an empty set (line 2) and then iterate through different types (line 3). For a type T_i and for each vertex $v \in V_i$, we find its

α-neighborhood $h(v, \alpha)$ and simply add $h(v, \alpha)$ as a new partition into $\mathcal{P}$. The following lemma summarizes the properties of our construction:

Lemma 6 *Partitions in $\mathcal{P}$ are edge disjoint and the union of all partitions in $\mathcal{P}$ cover the entire graph G.*

Proof. The edge disjoint property trivially holds by our construction of $h(v, \alpha)$. By visiting the vertices in each type, we have effectively included the α-neighborhoods of all vertices in G into $\mathcal{P}$, which leads to the conclusion that the union of the resulting partitions covers G. ■

Note that the order in which we iterate through different types may affect the final partitions $\mathcal{P}$ we build. However, no matter which order we choose, vertices in the same type always induce a set of partitions based on their α-neighborhoods. For example, the partitions $\mathcal{P}$ of Figure 5.3 (as condensed in Figure 5.4) are always the ones shown in Figure 5.8, using $\alpha = 1$.

5.4.4 Summarization

We first outline our approach to summarize the distinct structures in a partition $\mathcal{P}$. Then, we discuss how to make it more practical by proposing our optimizations. Finally, we discuss the related indices in Section 5.4.5. The general framework of our approach is shown in Algorithm 6.

Given a partition $\mathcal{P}$, Algorithm 6 retrieves all the distinct structures and stores them in a set S. Algorithm 6 begins with processing partitions in $\mathcal{P}$ in a loop (line 2). For a

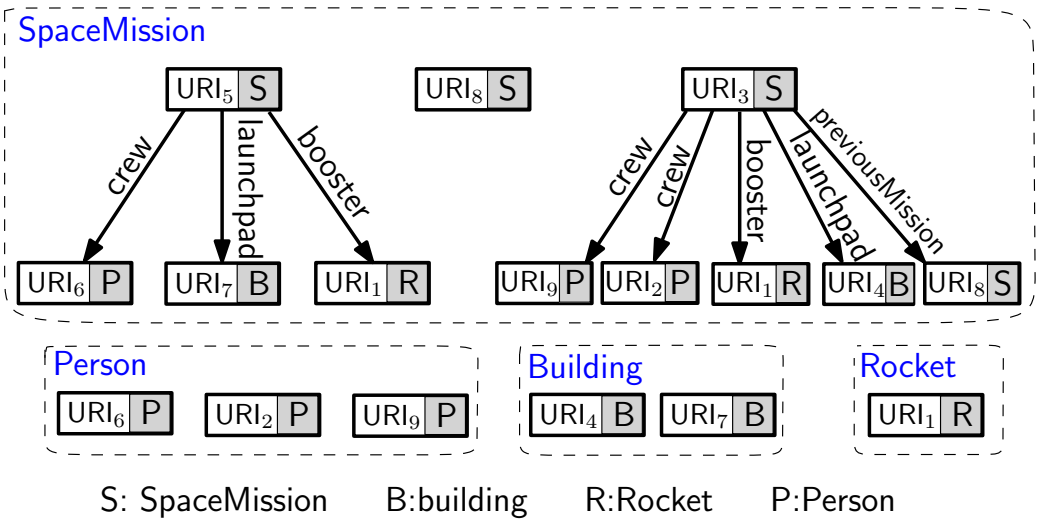


Figure 5.8: Partitions $\mathcal{P}$ of the RDF data in Figure 5.3, $\alpha = 1$

Algorithm 6: Summarize structures in $\mathcal{P}$

Input: $\mathcal{P} = \{h(v_1, \alpha), h(v_2, \alpha), \ldots\}$
Output: A set of summaries in S

```
1  S ← ∅;
2  for h_i ∈ P, i = 1, ..., |P| do
3      c ← core(h_i); //see discussion on optimization
4      for s_j ∈ S, j = 1, ..., |S| do
5          if f : c → s_i then
6              goto line 2; // also bookkeep f : c → s_i
7          else if f : s_i → c then
8              S ← S − {s_i}; //also bookkeep f : s_i → c
9      S ← S ∪ {c};
10 return S;
```

partition h_i, we use its core c to succinctly represent the connections between different types in h_i (line 3). Once a core c is constructed for a partition, we scan the existing summary structures in S to check (a) if c is homomorphic to any existing structure s_i in S; or (b) if any existing structure s_i in S is homomorphic to c. In the former case, we terminate the scan and S remains intact (without adding c), as in lines 5-6; in the latter case, we remove s_i from S and continue the scan, as in lines 7-8. When S is empty or c is not homomorphic to any of the structures in S after a complete scan on S, we add c into S. We repeat the procedure until we exhaust all the partitions in $\mathcal{P}$.

There are two practical problems in Algorithm 6. First, the algorithm requires testing subgraph isomorphism for two graphs in lines 3, 5, and 7, which is very expensive due to the NP-hard nature of problem. Second, we want to reduce $|S|$ as much as possible so that it can be cached in memory for query processing. The latter point is particularly important for RDF datasets that are known to be irregular, *e.g.,* DBpedia. Next, we address the two problems by preprocessing the partitions in $\mathcal{P}$.

The optimization is as follows. Before line 3 of Algorithm 6, consider each partition $h(v, \alpha)$ in $\mathcal{P}$, which visits the α-neighborhood of v in a breadth-first manner. We redo this traversal on $h(v, \alpha)$ and construct a *covering tree* for the edges in $h(v, \alpha)$, denoted as $h_t(v, \alpha)$. In more detail, for each visited vertex in $h(v, \alpha)$, we extract its type and create a *new* node in $h_t(v, \alpha)$ (even if a node for this type already exists). By doing so, we build a tree $h_t(v, \alpha)$ that represents all the distinct type-paths in $h(v, \alpha)$. In the rest of the algorithm (lines 3-10), we simply replace $h(v, \alpha)$ with $h_t(v, \alpha)$.

We illustrate the previous discussion with an example. As in Figure 5.9, a tree $h_t(v_1, 2)$ is built for the partition $h(v_1, 2)$. Notice that the vertex v_4 is visited three times in the

traversal (across three different paths), leading to three distinct nodes with type T_4 created in $h_t(v_1, 2)$. In the same figure, a tree $h_t(v_5, 2)$ is built from the partition $h(v_5, 2)$ and isomorphic to $h_t(v_1, 2)$.

There are two motivations behind this move. First, using the covering tree instead of the exact partition potentially reduces the size of the summary S. As seen in Figure 5.9, two partitions with distinct structures at the data level (*e.g.,* $h(v_1, 2)$ and $h(v_5, 2)$) could share an identical structure at the type level. Taking advantage of such overlaps is the easiest way to reduce the number of distinct structures in S. The second reason is efficiency. Whereas testing subgraph isomorphism is computationally hard for generic graphs, there are polynomial time solutions if we can restrict the testing on trees [101] – leading to better efficiency. For instance, to find the core of a covering tree h_t, it simply amounts to a bottom-up and recursive procedure to merge the homomorphic branches under the same parent node in the tree.

However, the improvements on efficiency and the size of S come with a cost, *i.e.,* it leads to a more dedicated design as to how to use the summaries in S for path-length estimation. We will discuss this issue in Section 5.5. Here, we first introduce the indices we construct.

5.4.5 Auxiliary Indexing Structures

To facilitate the keyword search, along with the summary S, we maintain three auxiliary (inverted) indexes.

A *portal* node ℓ is a data node that is included in more than one partitions (remember that partitions are edge disjoint, not node disjoint). Intuitively, a portal node joins different

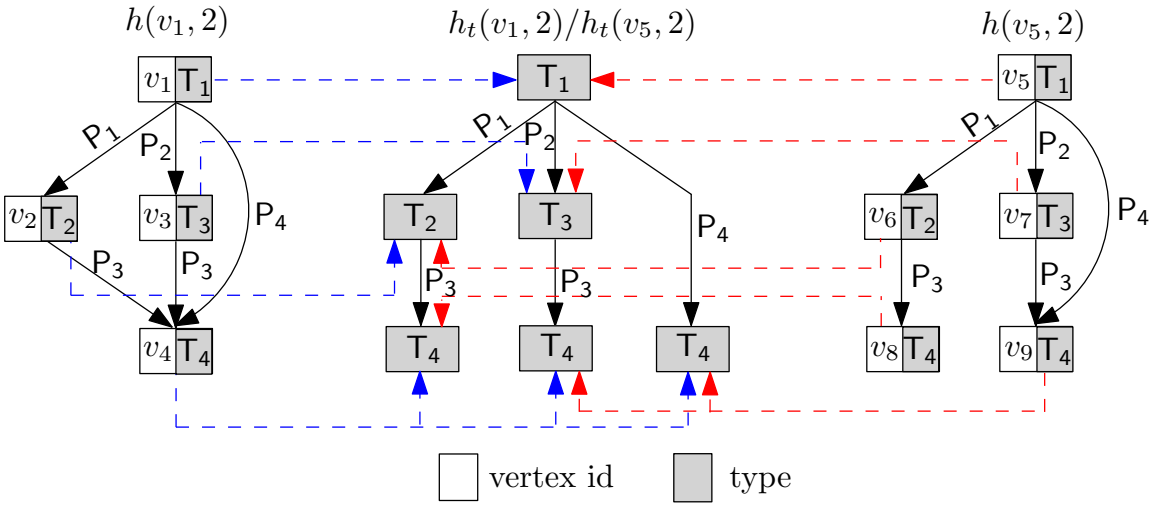


Figure 5.9: A tree structure for two partitions

partitions. A partition may have multiple portals but usually much less than the total number of nodes in the partition. Portal nodes allow us to piece together different partitions. In our *first index*, for each partition $h(v, \alpha)$, we assign it a unique id, and associate it with the list of portals in the partition. In practice, since the partition root node v is unique in each partition, we can simply use it to denote the partition $h(v, \alpha)$ when the context is clear. Notice that the *partition root* is different from the *answer root* defined in section 5.2.2.

Recall that we use $h_t(v, \alpha)$ to represent $h(v, \alpha)$, where a vertex in $h(v, \alpha)$ could correspond to more than one vertex in $h_t(v, \alpha)$. Let $\sigma(v_i)$ be such a one-to-many mapping from a vertex v_i in $h(v, \alpha)$ to at least two vertices in $h_t(v, \alpha)$; clearly, all vertices mapped by $\sigma(v_i)$ in h_t are of the same type, as they are the same node on different paths. Therefore, to make the bookkeeping efficient, we register in σ the *type* of v_i instead of the node ids. W.l.g., let $\Sigma = \{\sigma(v_1), \sigma(v_2), \ldots\}$ be all the one-to-many mappings in a partition. For instance, consider $h(v_1, 2)$ and $h_t(v_1, 2)$ in Figure 5.9, $\Sigma \leftarrow \{\sigma(v_4) = \{T_4\}\}$. The *second index* is to map the partition root v of $h(v, \alpha)$ to its Σ. Intuitively, this index helps rebuild from $h_t(v, \alpha)$ a graph structure that is similar to $h(v, \alpha)$ (more rigorous discussion in section 5.5).

Our *third index* maps data nodes in partitions to summary nodes in S. In particular, we assign a unique id sid to each summary in S and denote each node in S with a unique id nid. For any data node u in a partition $h(v, \alpha)$, this index maps the node u in $h(v, \alpha)$ to an entry that stores the partition root v, the id sid of the summary, and the id nid of the summary node that u corresponds to. Notice that since $h_t(v, \alpha)$ is built in a BFS traversal, we can easily compute the shortest path from v to any node in $h_t(v, \alpha)$ using this index.

In order to obtain the homomorphic mappings from each $h_t(v, \alpha)$ to a summary in S, one needs to maintain a log for all the homomorphisms found during the construction of S, as in lines 6 and 8 of Algorithm 6. Once S is finalized, we trace the mappings in this log to find all the mappings from data to summaries in S. As each partition (represented by its core) is either in the final S or is homomorphic to *one* other partition, the size of the log is linear to G. An example for such a log is shown in Figure 5.10 (h_t^i is the covering tree for the ith partition). It shows sets of trees (and their homomorphic mappings); each set is associated with one of the summaries in S that all trees in that set are homomorphic to. To find the final mappings, we scan each set of trees in the log and map the homomorphisms of each entry in a set to the corresponding entry from S, *i.e.,* the blue arrows in Figure 5.10. We remove the log once all the mappings to S are found.

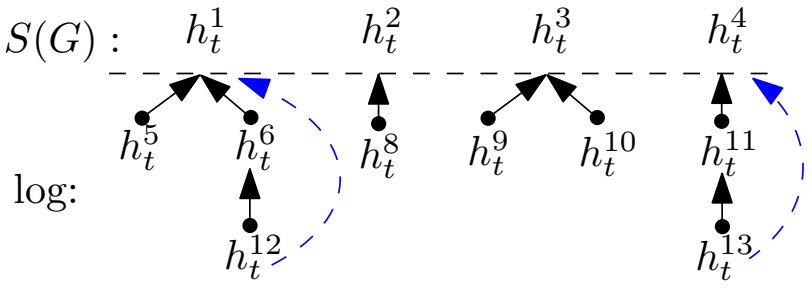


Figure 5.10: All the homomorphism in building S

5.5 Search with Summarization

Next, we present a scalable and exact search algorithm by leveraging graph partitions and the summarization introduced in Section 5.4. It performs a *two-level* backward search: one backward search at the summary-level, and one at the data-level. Only for identified connected partitions that are found to contain all the distinct keywords at the summary-level and whose score could enter the top-k answers do we initiate a backward search at the data-level on the selected partitions. Remember that path-length computation is at the heart of backward search and pruning. While working at the summary-level, exact path lengths are not available. Therefore, we first show how to estimate the path length of the actual data represented by our summary. Then, we proceed to describe the algorithm in detail.

5.5.1 Bound the Shortest Path Length

At the summary-level, any shortest path in the underlying RDF graph must go through a number of partitions, and for each partition, the path includes two of its portals, *i.e.,* an entrance and an exit node. By construction, the shortest distance from the partition root v of a partition to any vertex u in the same partition can be computed with the third index. By triangle inequality, the shortest distance $d(v_1, v_2)$ for any two vertices v_1 and v_2 in a partition with a partition root v can be upper bounded by $d(v_1, v_2) \leq d(v, v_1) + d(v, v_2)$, and lower bounded by $d(v_1, v_2) \geq |d(v, v_1) - d(v, v_2)|$. Yet, a possibly tighter lower bound can be found by using the correspondent summary of the partition that is rooted at v and Lemma 7.

Lemma 7 *Given two graphs g and h, if $f : g \to h$, then $\forall v_1, v_2 \in g$ and their homomorphic mappings $f(v_1)$, $f(v_2) \in h$, $d(v_1, v_2) \geq d(f(v_1), f(v_2))$.*

Proof. By definition, $\forall (u, v) \in g$, $(f(u), f(v)) \in h$. Since every edge in g is mapped to an edge in h by homomorphism, the shortest path p that leads to $d(v_1, v_2)$ in g can be mapped

to a path in h that starts at $f(v_1)$ and ends at $f(v_2)$ by applying f on each of the edges on p. Thus, $d(f(v_1), f(v_2))$ is at most $d(v_1, v_2)$. ■

The homomorphic mappings between a partition h, its covering tree h_t, and its summary s in S are shown in Figure 5.11(a). Notice that due to the optimization we employ in Section 5.4, there is no homomorphism from h to s, so that we can not apply Lemma 7 directly. In order to obtain a lower bound for the distance of any two vertices in h, we need to rebuild a homomorphic structure for h, using the second index and the correspondent h_t. To do so, we first define a join operator Join that combines all the nodes correspondent to $\sigma(x)$ of h_t in a single node that has same type as x. Recall that a $\sigma(x)$ registers the vertex x that appears in more than one path in h_t, *i.e.,* its replicas. Applying the Join on each σ of Σ for a partition, written as $\mathsf{Join}(h_t, \Sigma)$, undoes such splits.

We illustrate the previous discussion with an example. Applying the Join operator on $h_t(v_1, 2)$ and the respective Σ rebuilds $h(v_1, 2)$ in Figure 5.9. On the other hand, applying Join on $h_t(v_5, 2)$ with its Σ cannot reconstruct $h(v_5, 2)$ but results in a structure that is identical to $h(v_1, 2)$ in this example, which $h_t(v_5, 2)$ is homomorphic to. More formally, we have the following result.

Lemma 8 *For a partition h and its covering tree h_t, there is a homomorphism from h to* $\mathsf{Join}(h_t, \Sigma)$.

Proof. We construct such a homomorphic function $f : h \to \mathsf{Join}(h_t, \Sigma)$. Notice that by the objective of summarization, we only consider the types of nodes. For a node $v \in h$, if it is not registered by any mapping $\sigma \in \Sigma$, *i.e.,* it does not appear in two different paths, then let $f(v) = v$, since $\mathsf{Join}(h_t, \Sigma)$ has no effect on v; else if a node v is registered by some $\sigma_i \in \Sigma$, then by the property of Join, all vertices in h_t that have the type $\sigma_i(v)$ will be combined into one node. Let this node be u, then $f(v) = u$. Now consider the edges in h, by construction, h_t records all the paths from the partition root in h to every other node in h and Join does

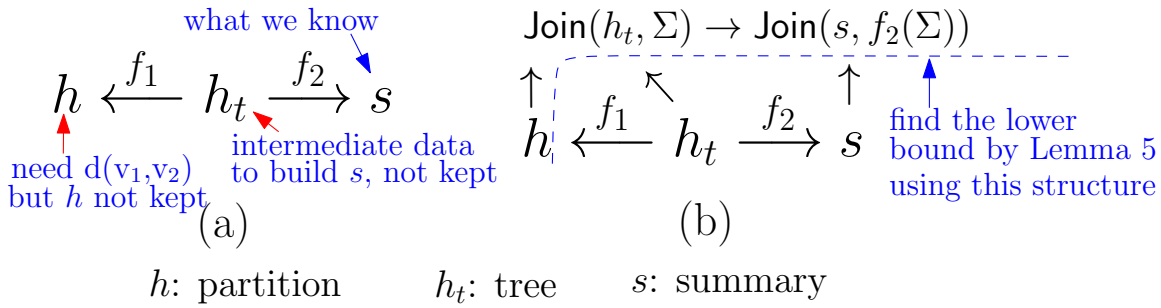


Figure 5.11: Homomorphic mappings

not add or delete edges or change their labels. Thus, if there is an edge $(v_1, v_2) \in h$, then there is an edge in h_t with the same label and the same types of the starting and ending nodes in h_t. Since Join will not alter the type of a node as well, it follows $(f(v_1), f(v_2))$ must be an edge in $\mathsf{Join}(h_t, \Sigma)$ with the same label and the same starting and ending types. ■

In what follows, we show how to establish a homomorphism from $\mathsf{Join}(h_t, \Sigma)$ to a structure derived from the correspondent summary s of h_t, such that Lemma 7 can be applied.

By construction, every h_t of a partition h in $\mathcal{P}$ is homomorphic to a summary s in S. As shown in Figure 5.11(a), assume the homomorphism is: $f_2 : h_t \to s$. Given the Σ of h, define $f_2(\Sigma) = \{f_2(\sigma(v)) \mid \sigma(v) \in \Sigma\}$ where $f_2(\sigma(v)) = \{f_2(u) \mid u \in \sigma(v) \wedge u \in h_t\}$, *i.e.,* the sets of correspondent mappings in the summary s by homomorphism. We have the following result:

Lemma 9 *For a partition h, its covering tree h_t and its summary s that has $f_2 : h_t \to s$, there is a homomorphism from $\mathsf{Join}(h_t, \Sigma)$ to $\mathsf{Join}(s, f_2(\Sigma))$.*

Proof. Here, we will construct such a homomorphism by the function $f : \mathsf{Join}(h_t, \Sigma) \to \mathsf{Join}(s, f_2(\Sigma))$.

For a node $v \in h_t$, consider the xcorrespondent $f_2(v)$ in s. If v is not registered by any $\sigma \in \Sigma$, then $\mathsf{Join}(h_t, \Sigma)$ has no effect on v and $\mathsf{Join}(s, f_2(\Sigma))$ has no effect on $f_2(v)$, hence $v \in \mathsf{Join}(h_t, \Sigma)$ and $f_2(v) \in \mathsf{Join}(s, f_2(\Sigma))$. Define $f = f_2$ for such v's. If v is in some $\sigma_i \in \Sigma$, all nodes in h_t that have the same type as v will be combined into one node. Let this node be u. On the other hand, by $\mathsf{Join}(s, f_2(\Sigma))$, all nodes that have the same type as $f_2(v)$ will be combined into one node. Let this node be u'. Define $f(u) = u'$. Now consider the edges. Notice that Join has no effect on edges and for every $(v_1, v_2) \in h_t$, $(f_2(v_1), f_2(v_2)) \in s$. This follows that for every $(f(v_1), f(v_2)) \in \mathsf{Join}(h_t, \Sigma)$, there is an edge $(f(f_2(v_1)), f(f_2(v_2))) \in \mathsf{Join}(s, f_2(\Sigma))$. ■

By Lemmas 8, 9 and the transitivity of homomorphism, a partition h is homomorphic to $\mathsf{Join}(s, f_2(\Sigma))$, as shown in Figure 5.11(b). Notice f_2 is a part of our third index, which maps a vertex in data to a vertex in summary. Finally, given any two vertices in a partition h, its shortest distance can be (lower) bounded by combining Lemmas 7, 8, 9 and using any shortest path algorithm, *e.g.,* Dijkstra's algorithm, in finding the shortest distance between the correspondent nodes on $\mathsf{Join}(s, f_2(\Sigma))$. In practice, we use the larger lower bound from either the summary or the triangle inequality.

5.5.2 The Algorithm

The algorithm is shown in Algorithm 7. We denote this algorithm as the SUMM method.

Algorithm 7: SUMM

Input: $q = \{w_1, w_2, \ldots, w_m\}$, $G = \{V, E\}$
Output: top-k answer $\mathcal{A}$

initialize $\{W_1, ..W_m\}$ and m min-heaps $\{a_1, ..a_m\}$;
$M \leftarrow \emptyset$; `// for tracking partitions`
for $u \in W_i$ **and** $i = 1..m$ **do**
  if $u \in h(v, \alpha)$ **then**
    $t \leftarrow (u, \{\emptyset\}, 0, 0)$;
    $a_i \Leftarrow (v, t)$; `// enqueue`
    if $v \notin M$ **then** $M[v] \leftarrow \{\text{nil}, ..., \underline{t}, ..., \text{nil}\}$;
    else $M[v][i] \leftarrow t$; `↑the ith entry`
while not *terminated* **and** $\mathcal{A}$ **not** *found* **do**
  $(v, (u, \mathcal{S}, d_l, d_u)) \leftarrow \texttt{pop}(arg\min_{i=1}^{m}\{\texttt{top}(a_i)\})$;
  let the last entry in $\mathcal{S}$ be (ℓ, v_ℓ) and $\mathcal{L} = \{\ell'_1, \ell'_2, \ldots\}$ be the portals in the partition rooted at v;
  for $\forall \ell' \in \mathcal{L}$ **do**
    compute d_l and d_u for $d(\ell, \ell')$ or $d(u, \ell')$;
    let $t \leftarrow (u, \mathcal{S} \cup (\ell', v_r), d_l + d'_l, d_u + d'_u)$;
    update $M[v]$ with t; `// see discussions`
    if $M[v]$ *is updated* **and** *nil* $\notin M[v]$ **then**
      $a_i \Leftarrow (v_r, t)$; `// enqueue`
      for *each new subgraph* g *incurred by* t **do**
        retrieve g from data;
        use BACKWARD on g **and** update $\mathcal{A}$;
return $\mathcal{A}$ (if found) **or nil** (if not);

Similar to the BACKWARD method in section 5.3, we define $\{W_1, W_2, \ldots, W_m\}$, where W_i is the set of vertices in G that contains the query keyword w_i. We also initialize m priority queues $\{a_1, \ldots, a_m\}$ and maintain a set M of entries, one for each considered partition. Each entry in M stores a unique partition root followed by m lists. The ith list records all the reachable vertices found so far that contain keyword w_i and through which other partitions they connect to the current partition in the backward expansions. An entry is in the form of quadruples – (u, $\mathcal{S}$, d_l, d_u). In the quadruple, the node u is the first vertex in the backward expansion that contains the keyword w_i; the expansion reaches the current partition by routing through a sequence of the portals from some partitions, stored in $\mathcal{S}$ as a sequence of (*portal*, *partition root*) pairs. A sequence $\mathcal{S}$ defines a path (of partitions) that begins at u. We illustrate the previous discussion with an example. A subsequence

$\{(\ell, v_a), (\ell', v_b)\}$ of $\mathcal{S}$ indicates that the path enters the partition rooted at v_b from the portal ℓ (exiting from a partition rooted at v_a) and use ℓ' as its exit portal. We are interested in (lower and upper) bounding the shortest distance that connects two adjacent portals in $\mathcal{S}$, *e.g.,* $d(\ell, \ell')$ in the partition rooted at v_b.

In the quadruple, the lower and upper bounds for the path defined by the portals (and the starting vertex u) in $\mathcal{S}$ are denoted as d_l and d_u.

Here is another example. In Figure 5.12, assume $m = 2$ (*i.e.,* the query has two keywords) and an entry in M for a partition rooted at v is shown as below.

The entry records that there is a path (of partitions) from w_1 that reaches the current partition rooted at v. This path starts at v_a, enters the concerning partition at portal ℓ_2, and has a length of at least 5 hops and at most 7 hops. To reach the partition rooted at v, the path has already passed through a partition rooted at v_0. The same for w_2, the concerning partition is reachable from two paths starting at v_b and v_c, respectively; both contain the keyword w_2.

With the data structures in place, the algorithm proceeds in iterations, which can be summarized as follows.

• In the first iteration. For each vertex u from W_i, we retrieve the partition root v that u corresponds to, from the third index. Next, if there is an entry for v in M, we append a quadruple t=(u, $\{\emptyset\}$, 0, 0) to the ith list of the entry; otherwise, we initialize a new entry for v in M (with m empty lists) and update the ith list with t, as in lines 7-8. We also add an entry (v, t) to the priority queue a_i (entries in the priority queue are sorted in ascending order by their lower bound distances in t's). We repeat this process for all W_i's for $i = 1, \ldots, m$, which completes the first iteration (lines 3-8).

• In the jth iteration. We pop the smallest entry from all a_i's, say $(v, (u, \mathcal{S}, d_l, d_u))$ (line 10). We denote the partition rooted at v as the current partition. Let the last pair in $\mathcal{S}$ be (ℓ, v_ℓ), which indicates that the path leaves the partition rooted at v_ℓ and enters the current partition using portal ℓ. Next, for the current partition, we find its portals $\mathcal{L} = \{\ell'_1, \ell'_2, \ldots\}$ from the first index. For each ℓ' in $\mathcal{L}$, we compute the lower and upper bounds for $d(\ell, \ell')$ (or $d(u, \ell')$ if ℓ=*nil*) in the current partition using the approach discussed in Section 5.5.1, denoted as d'_l and d'_u (line 13). A portal ℓ' can connect the current partition to a set P'

w_1	w_2
t_1=(v_a, $\{(\ell_2, v_0)\}$, 5, 7)	t_2=(v_b, $\{(\ell_1, v_4), (\ell_0, v_5)\}$, 3, 5)
	t_3=(v_c, $\{(\ell_3, v_2)\}$, 5, 6)

Figure 5.12: An entry in M for the partition rooted at v

of neighboring partitions. For each partition in P', denoted by its partition root v_r, we construct a quadruple t=(u, $\mathcal{S} \cup (\ell', v_r)$, $d_l + d'_l$, $d_u + d'_u$) as in line 14. We also search the entry for v_r in M and update its ith list with t in the same way as in the first iteration. However, if either of the following cases is satisfied, we stop updating the entry for v_r in M: (i) adding ℓ' to $\mathcal{S}$ causes cycle; and (ii) $d_l + d'_l$ is greater than the kth largest upper bound in the ith list. Otherwise, we also push (v_r, t) to the queue a_i.

At any iteration, if a new quadruple t has been appended to the ith list of an entry indexed by v in M, and all of its other $m-1$ lists are nonempty, then the partition rooted at v contains potential answer roots for the keyword query. To piece together the partitions that contain all the query keywords, we find all the possible combinations of the quadruples from the $(m-1)$ lists, and combine them with t. Each combination of m quadruples denotes a conjunctive subgraph that contains all the query keywords and the answer root.

To see a concrete example, consider the example in Figure 5.12. Let t_1 be the new quadruple just inserted to the first list of an entry in M. Since both of its lists are now nonempty, two combinations can be found, *i.e.,* (t_1, t_2) and (t_1, t_3), which leads to two conjunctive subgraphs. Using the partition information in the quadruples, we can easily locate the correspondent partitions. We will detail how to efficiently retrieve the instance data for a partition by using its summary in Section 5.6. Once the instance data from the selected partitions are ready, we can simply proceed to the second-level backward search by applying the BACKWARD method to find the top-k answers on the subgraph pieced together by these partitions (line 20). In any phase of the algorithm, we track the top-k answers discovered so far with a priority queue.

• Termination condition. The following Lemmas provide a correct termination condition for the SUMM method.

Lemma 10 *Let* (v, (u, $\mathcal{S}$, d_l, d_u)) *be an entry in the priority queue, then for any* v' *in the partition rooted at* v *and for any path starting from* u *and using the portals in* $\mathcal{S}$*, its length* $d(u, v') \geq d_l$.

Proof. Let $\mathcal{S}$=$\{(\ell_1, v_1), (\ell_2, v_2), \ldots, (\ell_k, v_k)\}$. It has $d(u, v') \geq d(u, \ell_k) + d(\ell_k, v') \geq d(u, \ell_k)$, where ℓ_k is the portal in $\mathcal{S}$ that the path uses to enter the partition rooted at v and $d(u, \ell_k)$ is the length for the subpath that reaches the portal ℓ_k from u. Let $d(\ell_i, \ell_{i+1})$ be the fragment of the path that is in the partition rooted at v_{i+1} $(i = 0, \ldots, k-1)$ and $\ell_0 = u$, we have $d(u, \ell_k) = d(u, \ell_1) + d(\ell_1, \ell_2) + \ldots + d(\ell_{k-1}, \ell_k) \geq d_l(u, \ell_1) + d_l(\ell_1, \ell_2) + \ldots + d_l(\ell_{k-1}, \ell_k) = d_l$.

■

Lemma 11 *Let* $(v, (u, \mathcal{S}, d_l, d_u))$ *be the top entry in the priority queue* a_i*, then for any explored path* p *from* w_i *in the queue* a_i*, the length of* p*, written as* $d(p)$*, has* $d(p) \geq d_l$*.*

Proof. Let $(v', (u', \mathcal{S}', d'_l, d'_u))$ be *any* entry in a_i that represents a path starting at u' and reaches partition rooted at v'. Denote this path as p' and its length as $d(p')$. From Lemma 10, $d(p') \geq d'_l$. By the property of priority queue (min-heap), $d'_l \geq d_l$ for any entry in a_i. ■

We denote the set of all unexplored partitions in $\mathcal{P}$ as $\mathcal{P}_t$. For a partition h rooted at v that has not been included in M, clearly, $h \in \mathcal{P}_t$. The best possible score for an answer root that can be found in the partition h is to sum the d_l's from all the top entries of the m expansion queues, *i.e.,* $a_1, \ldots, a_m$. Let these m top entries be $(v_1, (u_1, \mathcal{S}^1, d_l^1, d_u^1)), \ldots, (v_m, (u_m, \mathcal{S}^m, d_l^m, d_u^m))$, respectively. We have the following results.

Lemma 12 *Let* g_1 *be the possible unexplored candidate answer rooted at a vertex in partition* h*, with* $h \in \mathcal{P}_t$*,*

$$s(g_1) > \sum_{i=1}^{m} d_l^i. \tag{5.3}$$

Proof. Since $h \notin M$, in order to reach h, we need at least one expansion from some partition to its neighboring partition from at least one of the top m entries in the priority queues. By Lemma 11, the length for the shortest possible path from a keyword w_i is lower bounded by d_l^i. Therefore, to reach partition h, it requires a distance of at least $d_l^i + 1$. This shows $s(g_1) > \sum_{i=1}^{m} d_l^i$. ■

Next, consider the set of partitions that have been included in M, *i.e.,* the set $\mathcal{P} - \mathcal{P}_t$. For a partition $h \in \mathcal{P} - \mathcal{P}_t$, let the first quadruple from each of the m lists for its entry in M be: $t_1 = (\hat{u}_1, \hat{\mathcal{S}}_1, \hat{d}_l^1, \hat{d}_u^1), \ldots, t_m = (\hat{u}_m, \hat{\mathcal{S}_m}, \hat{d}_l^m, \hat{d}_u^m)$ (note that due to the order of insertion, each list has been implicitly sorted by the lower bound distance $\hat{d}_l$ in ascending order), where $t_j = nil$ if the jth list is empty. Then, we have:

Lemma 13 *Let the best possible unexplored candidate answer as* g_2*, which is rooted at a vertex in the partition* h*, where* $h \in \mathcal{P} - \mathcal{P}_t$*, then*

$$s(g_2) > \sum_{i=1}^{m} f(t_i)\hat{d}_l^i + (1 - f(t_i))d_l^i, \tag{5.4}$$

where $f(t_i)$=1 *if* $t_i \neq nil$ *otherwise* $f(t_i)$=0*.*

Proof. The candidate subgraph formed by $(t_1, \ldots, t_m)$ from the entry h in M has the smallest (aggregated) lower bound from combining the lower bounds in all m lists, due to the fact that each list is sorted in ascending order by the lower bound distance d_l.

When some $t_i = nil$, no expansion from any vertex associated with the keyword w_i has reached the partition h yet. Following the same argument from the proof for Lemma 12, the shortest distance to reach a vertex in W_i is at least $d_l^i + 1$. For the others where $t_i \neq nil$, the best possible unexplored path from any keyword w_i to reach some node in h will have a distance that is no less than $\hat{d}_l^i$ due to the property of the priority queue and the fact that it is a lower bound. ■

Finally, we can derive the termination condition for the search. We denote the score of the best possible answer in an unexplored partition as $s(g_1)$, as defined by the RHS of (5.3); and the score of the best possible answer in all explored partitions as $s(g_2)$, as defined by the RHS of (5.4) . Let g be the candidate answer with the kth smallest score during any phase of the algorithm. Then, the backward expansion on the summary level can safely terminate when $s(g) < \min(s(g_1), s(g_2))$. By Lemmas 12 and 13, we have:

Theorem 4 *The* SUMM *method finds the top-k answers* $\mathcal{A}(q, k)$ *for any top-k keyword search query* q *on an* RDF *graph.*

Proof. By combining Lemmas 12 and 13, it suffices to derive the termination condition. Together with Theorem 3, it guarantees the correctness of the SUMM method. ■

5.6 Accessing Data and Update

The SUMM algorithm uses the summaries to reduce the amount of data accessed in the BACKWARD method. For the algorithm to be effective, we need to efficiently identify and retrieve the instance data from selected partitions. One option is to store the triples by partitions and index on their partition ids, *i.e.,* adding another index to the algorithm. However, whenever an update on the partition happens, we need to update the index. Furthermore, the approach enforces a storage organization that is particular to our methods (*i.e.,* not general). In what follows, we propose an alternative efficient approach that has no update overhead and requires no special storage organization. Our approach stores the RDF data in an RDF store and works by dynamically identifying the data of a partition using appropriately constructed SPARQL queries that retrieve only the data for that partition.

Since graph homomorphism is a special case of homomorphism on relational structure (*i.e.,* binary relations) [60] and SPARQL is equivalent to relational algebra [88], we can

use the Homomorphism Theorem [14] to characterize the results of two homomorphic and conjunctive SPARQL query patterns.

Theorem 5 *Homomorphism Theorem [14]. Let q and q' be relational queries over the same data D. Then $q'(D) \subseteq q(D)$ iff there exists a homomorphism mapping $f : q \to q'$.*

Recall that $f_1 : h_t \to h$ (see Figure 5.11(a)) and for each h_t, we extract a core c from h_t. By definition, c is homomorphic to h_t, thus c is homomorphic to h (transitivity). Using c as a SPARQL query pattern can extract h due to Theorem 5.

Here, we need to address two practical issues. First, there is usually a many-to-one mapping from a set of h_t's to the same core c – leading to a low selectivity by using c as the query pattern. To address this issue, we can bind constants from the targeted partition to the respective variables in query pattern. These constants could include the root and the portals of the targeted partition that are retrievable from the inverted indexes. The second issue is that in our construction of S, we do not explicitly keep every c. Instead, a core c is embedded (by homomorphism) to a summary $s \in S$, where c is a subtree of s. To construct a SPARQL query from s, we first need to find a mapping for the partition root in s, then the triple patterns corresponding to the subtree in s are expressed in (nested) OPTIONALs from the root to the leaves. For example, the SPARQL query for the partition rooted at URI_5 in Figure 5.8 can be constructed by using the summary in Figure 5.7(a). Notice that URI_5 is bound to the root to increase selectivity. The query to retrieve to the respective portion of data is shown in Figure 5.13.

One limitation of previous work on summarizing RDF data is their inability to handle updates in an *incremental* way. Here, we show that our summaries can be incrementally updated. We first discuss how to handle insertions. Insertions can be handled efficiently. A new subgraph (a set of triples) is simply treated as a data partition that has not been traversed. Indexing structures and the summarization can be updated accordingly.

Next, we discuss how to settle deletions. Let t be the triple deleted. Then all the partitions that visit the subject/object of t will be updated. As a deletion only affects

```
SELECT * WHERE{URI5 name "A1". URI5 type S.
OPTIONAL{URI5 launchPad ?x. ?x type B.}
OPTIONAL{URI5 booster ?y. ?y type R}
OPTIONAL{URI5 crew ?z. ?z type C} .
OPTIONAL{URI5 previousmission ?m. ?m type S} . }
```

Figure 5.13: A query to retrieve the targeted partition

the nodes in the α-neighborhood of t's subject and object, this can be done efficiently. To update S, there are two cases to consider: (i) if the core of an updated partition is not in S, *i.e.,* it is homomorphic to a core in S, we simply rebuild its core and update the corespondent inverted indexes; (ii) if the core of an updated partition is in S, this will lead to a removal for the core in S. In addition, we retrieve all the partitions homomorphic to the deleted core and summarize them (together with the updated partition) as if they are new data. To access these partitions efficiently, we can leverage the technique discussed at the beginning of this section and use the (to be) deleted core as the query pattern.

5.7 Related Work

For keyword search on generic graphs, many techniques [53, 63] assume that graphs fit in memory, an assumption that breaks for big RDF graphs. For instance, the approaches in [53, 63] maintain a distance matrix for all vertex pairs, and clearly do not scale for graphs with millions of vertices. Furthermore, these works do not consider how to handle updates. A typical approach used here for keyword-search is backward search. Backward search aims to find a Steiner tree in the data graph, which is NP-hard. Therefore, in the past, a heuristic was used for answering keyword queries on graphs that cannot guarantee the correctness of the search result. Reference [53] outlined a tractable scoring function that enables the backward search idea as a baseline solution. However, unlike our work, they did not provide details, nor a rigorous analysis for the soundness of their backward search approach. In this work, we extended this idea to big RDF graphs with rigorous soundness analysis.

Techniques for summarizing large graph data to support keyword search were also studied [40]. They assumed edges across the boundaries of the partitions are weighted. A partition is treated as a *supernode* and edges with minimal weights are *superedge*s. Recursively, a large graph can be summarized in this fashion and fit into memory for query processing. This model is designed for generic graphs, and cannot be easily extended for RDF data, as edges in RDF data encode important semantics and relationships that cannot be ignored. Besides, such a summarization is not efficiently updatable.

Keyword search for RDF data has been recently studied in [109], with the same definition as ours. Both approaches are *schema-gnostic*. A schema to represent the relations in entities of distinct types is summarized from the RDF data. Keyword search in [48] also used the same summarization, but with a different scoring function. Both works extended the backward search method to RDF graphs as a baseline method for comparison; however, as we will study in Section 5.3, their version of the backward search method is problematic,

and they did not rigorously analyze the correctness of their backward search method.

More importantly, the proposed summarization [109] has a serious limitation as we show in this chapter: it bundles all the entities of the same type into one node in its summary, which loses too much information. For instance, in Figure 5.3, URI_6, URI_7, and URI_9 will be represented by a single vertex as *SpaceMission*. As a result, this summarization *generates erroneous results (both false positives and negatives)*, as we have *already illustrated in Figure 5.1*.

As another example, consider Figure 5.3, where all vertices of type *SpaceMission* are represented as one node in their summarization. Then, the edge *previousMission* connecting URI_3 and URI_8 results in a self-loop over this node in their summarization, which is incorrect since such a loop does not exist in the data. Furthermore, they do not support updates. While we also built our summarization using *type information*, our summarization leverages on completely different intuitions, which *guarantee* (a) the soundness of our results; and (b) the support of efficient updates.

There are other works related to keyword search on graphs. In [76], a 3-in-1 method is proposed to answer keyword search on structured, semistructured, and unstructured data. The idea is to encode the heterogeneous relations as a graph. Similar to [53, 63], it also needs to maintain a distance matrix. An orthogonal problem to keyword search on graph is the study of different ranking functions. This problem is studied in [48, 49]. In this work, we adopt the standard scoring function in previous work in RDF [109] and generic graphs [53].

5.8 Experiments

We implemented the proposed Backward and Summ methods in C++. We also implemented two representative approaches in keyword search on RDF/graph data, respectively, proposed in [109] (which also built a summarization) and [53] (which used distance matrix). We denote these two approaches as Schema and Blinks. All experiments were conducted on a 64-bit Linux machine with 6GB of memory. Note that in addition to results from this section, we have already shown the defects and limitations of these two methods in Figure 5.1 and Figure 5.2, respectively. In particular, the state-of-the-art method from [109] may return incorrect answers due to the limitations from its summarization, as shown in Figure 5.1.

5.8.1 Experiment Setups

We used four RDF datasets. With LUBM generator, we created a *default* dataset of 5 million triples and varied its size up to 28 million triples. The other datasets are real datasets: Wordnet, Barton, and DBpedia Infobox, which have about 2 million, 40 million, and 30 million triples, respectively. The number of distinct types are shown in Table 5.2. Notice that most RDF datasets are composed by less than a hundred types with DBpedia being the only exception.

We used the disk-based B^+-tree implementation from the TPIE library to build a Hexstore-like [114] index on the RDF datasets, which is adopted by the mainstream RDF engines. For the subgraph isomorphism tests in Algorithm 6, we used the VFLib. We assume each entity in the data has a type. For an entity that has multiple types, we bind the entity to its most popular type. To store and query RDF data with SPARQL, we use Sesame [30]. In all experiments, if not otherwise noted, we built the summarization for 3-hop neighbors, *i.e.,* $\alpha = 3$, and set $k = 5$ for top-k queries.

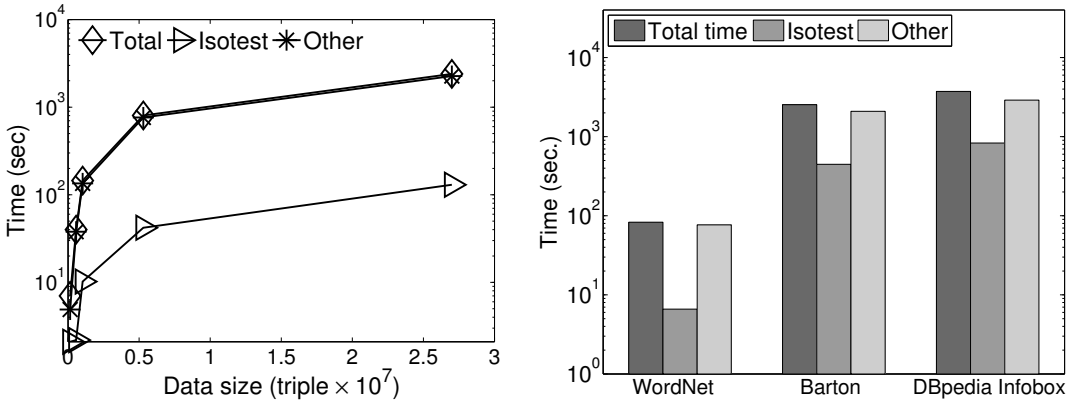
5.8.2 Evaluating Summarization Techniques

We start with a set of experiments to report the time (in log scale) in building a summarization. For LUBM, we vary the dataset size from 100 thousand triples to 28 million triples. In Figure 5.14(a), we plot the total time for building the summarization, which includes: the time spent to find homomorphic mappings (*i.e.,* performing subgraph isomorphism tests) and the time spent for the rest of operations (*e.g.,* partitioning the graph and constructing the inverted indexes). The latter cost dominates the summarization for all the cases in LUBM datasets. The same trend can also be observed in the three real datasets, as shown in Figure 5.14(b). The summarization is built once and thereafter incrementally updatable whenever the data get updated. By comparison, the summarization by the SCHEMA method cannot be incrementally updated, though it can be built faster.

As the SCHEMA method generates one (type) node in the summarization for all the nodes in the data that have the same type, the size of summarization (in terms of number of nodes) is equal to the number of distinct types from the data, as specified in Table 5.2. For our summarization, we plot the number of partitions and the number of summaries in Figures 5.15(a) and 5.15(b). In Figure 5.15(a) for LUBM, the summarization results in at

Table 5.2: Number of distinct types in the datasets

LUBM	Wordnet	Barton	DBpedia Infobox
14	15	30	5199



(a) (b)

Figure 5.14: Time for the summary construction (a) LUBM and (b) Real datasets

least two orders of magnitude less distinct structures w.r.t. the number of partitions. Even in the extreme case where the largest dataset is partitioned into about a million subgraphs, the number of distinct summaries remains under 100. In fact, it remains almost a constant after we increase the size of the dataset to 1 million triples. This is because LUBM data are highly structured [43].

On the other hand, real RDF datasets are known to have a high variance in their *structuredness* [43]. In Figure 5.15(b), we plot the number of distinct summaries for real datasets after summarization. For Wordnet and Barton datasets, the summarization distills a set of summaries that contains more than three orders of magnitude less structures compared to the respective set of partitions. Even in the most challenging case of DBpedia Infobox, the summarization achieves more than one order of magnitude less structures w.r.t. the partitions, as in Figure 5.15(b).

In Figures 5.16(a) and 5.16(b), we compare the number of triples stored in the partitions and in the summarization. Clearly, the results show that the distinct structures in the data partitions can be compressed with orders of magnitude less triples in the summarization, *e.g.,* more than one order of magnitude less for DBpedia Infobox and at least three orders of magnitude less for LUBM, Wordnet, and Barton. Since the summarization is small, this suggests that we can keep the summarization in main memory to process keyword query. Therefore, the upper and lower bounds for the distance traversed in a backward expansion (as used by our SUMM method) can be computed in memory.

In Figures 5.17(a) to 5.17(d), we report the impact of α (a parameter on the max number of hops in each partition; see Section 5.4.3) on the size of summarization. Intuitively, the smaller α is, the more similar the α-neighborhoods are, leading to less summaries in the summarization. This is indeed the case when we vary α for all the datasets. The smallest

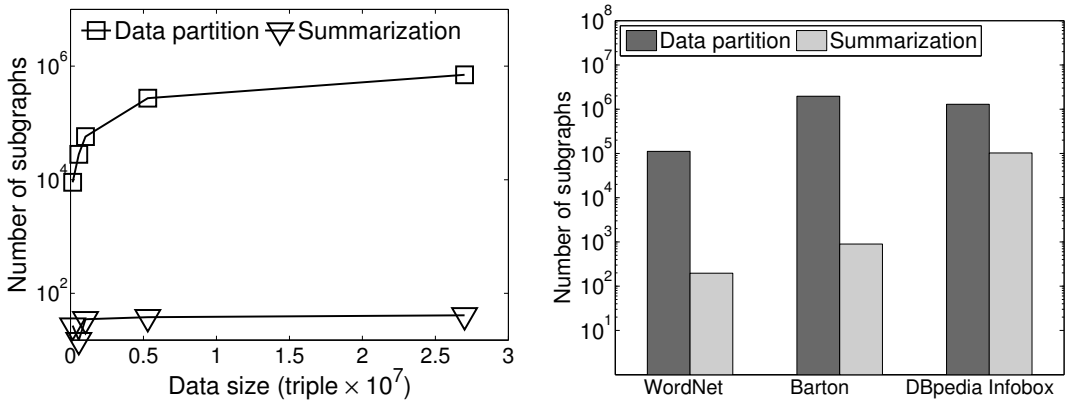


Figure 5.15: Number of subgraphs: partitions vs. summaries $S(G)$ (a) LUBM and (b) Real datasets

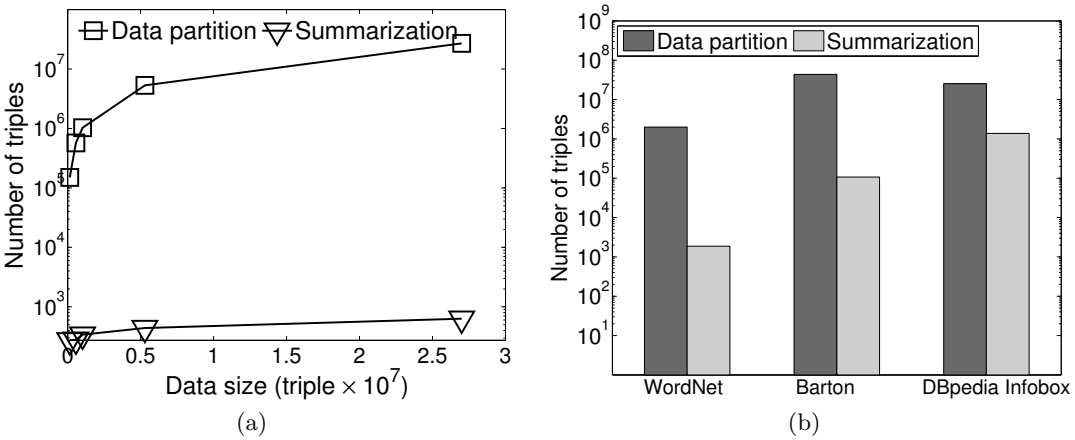


Figure 5.16: Number of triples: partitions vs. summaries $S(G)$ (a) LUBM and (b) Real datasets

summarizations are achieved when $\alpha = 1$ in Figures 5.17(a) to 5.17(d). Notice that there is a trade-off between the size of the summarization and the size of the auxiliary indexes. A smaller partition implies that more nodes become portals, which increases the size of auxiliary indexing structures. On the other hand, increasing α leads to larger partitions in general, which adds more variance in the structure of the partitions and inevitably leads to more summaries in the summarization. However in practice, since the partitions are constructed by directed traversals on the data, we observed that most of the directed traversals terminate after a few hops. For instance, in LUBM and Wordnet, most partitions stop growing when $\alpha > 3$. A similar trend is visible in Figures 5.17(c) and 5.17(d). When we increase α, the number of distinct structures changes moderately.

In Figures 5.18(a) and 5.18(b), we study the size of the auxiliary indexes. Figure 5.18(a)

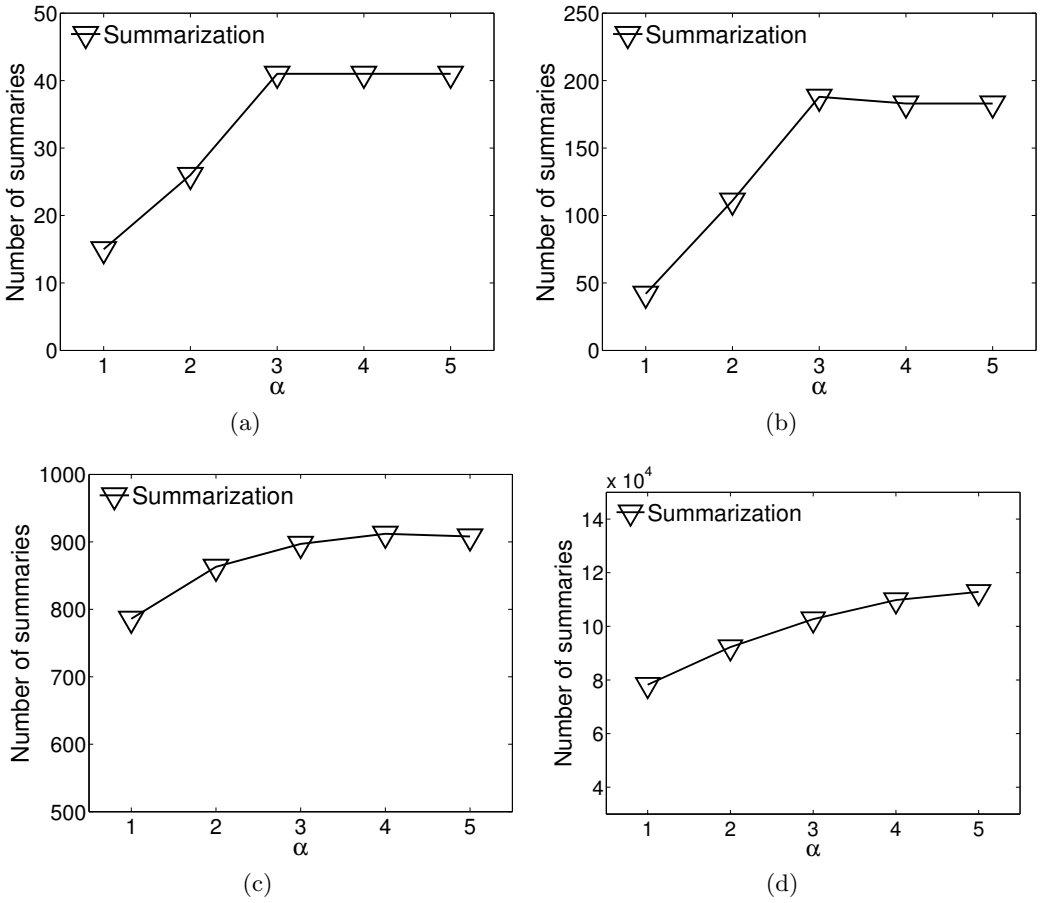
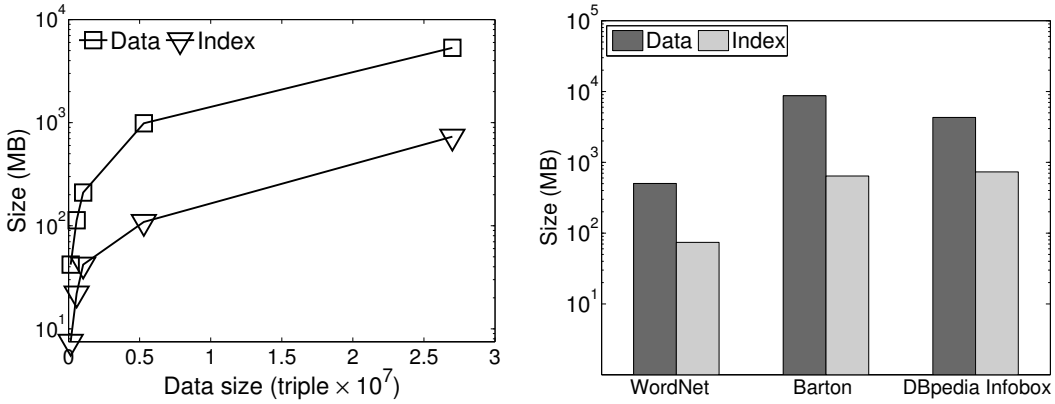


Figure 5.17: Impact of α to the number of summaries in $S(G)$: (a) LUBM (b) Wordnet (c) Barton and (d) DBPedia Infobox

shows that for LUBM, the size of the auxiliary indexes is one order of magnitude less than the data size. Similar trends can be observed in Figure 5.18(b) for the real datasets. The reason for these results is that for all indexes, we do not explicitly store the edges of the RDF data that usually dominate the cost in storing large graphs. In Figure 5.19, we report the breakdown of the inverted indexes for all the datasets. The most costly part is to store the mappings from the third inverted index (*i.e.,* 3rd_idx in the figure), whereas other mappings are small in size. Thus, to efficiently process query, we can keep the first and the second inverted indexes in main memory.

We also compare in Figure 5.20 the index overhead of different methods as we vary dataset sizes. Notice that BLINKS is the most costly method in terms of storage overhead (*i.e.,* demands one order of magnitude more space), as it builds a distance matrix, leading to a quadratic blowup in indexing size. BLINKS is no doubt the faster method for small data, but it clearly does not scale with large RDF datasets. Therefore, we do not report BLINKS in the evaluation of query performance.



(a) (b)

Figure 5.18: Size of the auxiliary indexes (a) LUBM and (b) Real datasets

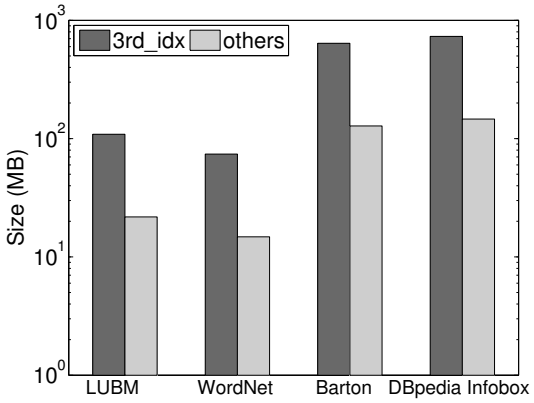


Figure 5.19: Breakdown

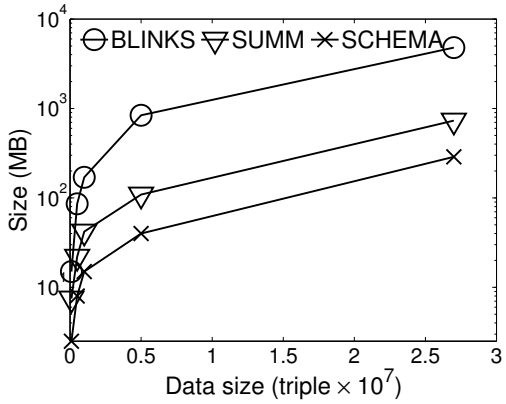


Figure 5.20: Index size

5.8.3 Query Performance

In this section, we study the performance of top-k keyword search using Summ and Backward. In particular, we compare the proposed methods with Schema, which is the only existing method that can scale to large RDF datasets. To this end, we design a query workload that has various characteristics. Table 5.3 lists 10 typical queries, together with the number of keyword occurrences in the datasets. For the LUBM data, all the keywords are selected from the first university in the dataset, except for keywords w.r.t. publications 17 and 18. For the two indicated keywords, we select one copy of each publication from the first university and pick the rest of them randomly from other universities. This is to simulate the cases in real datasets where not all the keywords in a query are close to each other. For the three real datasets, we pick two representative queries for each of them for evaluation. For long running queries, we terminate the executions after 1000 seconds. We plot the response times in log scale.

Table 5.3: Sample query workload

	Query	# nodes	Dataset
Q_1	[Pub_{19}, Lec_{13}]	(20,13)	L
Q_2	[$Research_5$, $FullProf_9$, Pub_{17}]	(9,4,83)	L
Q_3	[$FullProf_9$, $Grad_0$, Pub_{18}, Lec_6]	(4,15,40,5)	L
Q_4	[Dep_0, $Grad_1$, Pub_{18}, $AssocProf_0$]	(1,15,40,15)	L
Q_5	[Afghan, Afghanistan, al-Qaeda, al-Qa'ida]	(6,3,3,2)	W
Q_6	[3rdbase, 1stbase, baseball team, solo dance]	(14,13,17,4)	W
Q_7	[Knuth, Addison-Wesley, Number theory]	(1,1,35)	B
Q_8	[Data Mining, SIGMOD, Database Mgmt.]	(166,1,4)	B
Q_9	[M. Bloomberg, New York City, Manhatton]	(1,7,108)	I
Q_{10}	[George W. Bush, Saddam Hussein, Iraq]	(1,1,48)	I

L:LUBM W:Wordnet B:Barton I:DBpedia Infobox

We first use SCHEMA to answer the queries in Table 5.3. SCHEMA generates a set of k SPARQL queries for each keyword query consisting of multiple keywords. Evaluating these queries is supposed to return the top-k answers w.r.t. the scoring function. Here, even if we fix the incorrect termination condition in SCHEMA (as discussed in Section 5.3), our observation is that SCHEMA still returns empty results for *all* of the queries, as we have indicated in Figure 5.1. This can be explained by the way it summarizes the data, where all nodes of the same type are indistinguishably mapped to the same type node in the summarization. For instance, every FullProfessor has a publication in LUBM, whereas this does not mean that FullProfessor9 has a Publication17 for Q_2. Hence, the state-of-the-art method SCHEMA may return incorrect results for many queries. In what follows, we report the query performance of the BACKWARD and SUMM methods, both of which have provable guarantees on the correctness of the query results. The results are shown on Figures 5.21(a) and 5.21(b), respectively.

The efficiency of a keyword search on graph is known to be determined by a collection of factors [53], with no single factor being the most deterministic one. In particular, we observe that for selective keywords (*i.e.,* keywords that have few occurrences in the data) and especially those that are close to each other, *e.g.,* Q_1, Q_5, both BACKWARD and SUMM can answer the queries efficiently. In some cases, *e.g.,* Q_5, BACKWARD even outperforms SUMM due to the fact that SUMM uses a lower bound to decide its termination condition. This inevitably requires the SUMM approach to access more data than what is necessary to correctly answer the query.

However, being selective alone does not necessarily lead to better query performance, especially if the selective keyword corresponds to a hub node that has a large degree, which

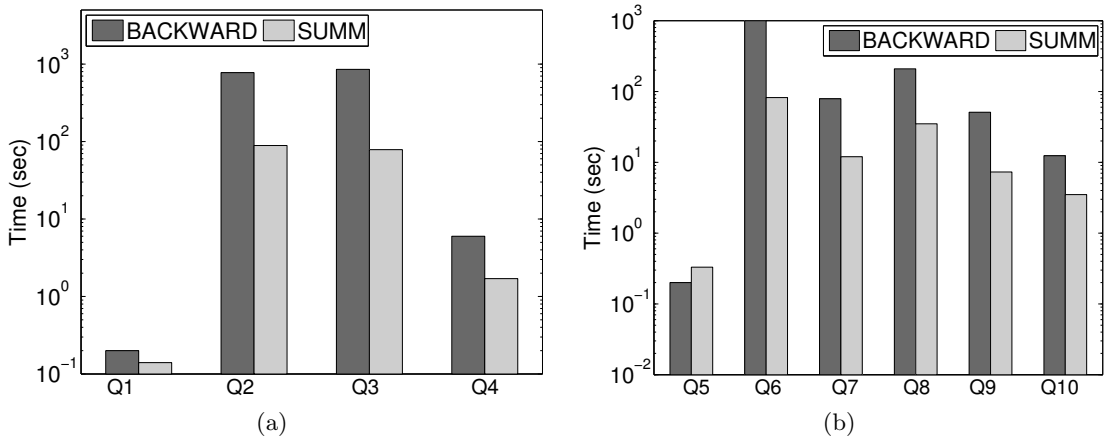


Figure 5.21: Query performance (a) LUBM and (b) Real datasets

quickly increases the size of the priority queue, *e.g.,* the Department0 in Q_4. On the other hand, as the keywords in the query become nonselective, *e.g.,* Q_3, or the keywords are far away from one another, *e.g.,* the solo dance and the baseball team in Q_6, the SUMM approach generally performs much better than the BACKWARD method.

For one thing, this is because only when connected partitions are found to contain all the keywords does SUMM need to access the whole subgraph on disk. This leads to savings in dealing with keywords that will not result in any answer, *e.g.,* most of the keyword nodes for publications 17 and 18 in Q_2–Q_4. For the other, at the partition level, the backward expansion in the SUMM approach can be done almost completely in memory as the major indexes for expansion are lightweight (as shown in Section 5.8.2) and therefore can be cached in memory for query evaluation. In such cases, *i.e.,* Q_2–Q_4 and Q_6–Q_{10}, we observe that the SUMM approach can result in much better performance.

5.9 Conclusion

After identifying the defects and limitations of existing methods, we studied the problem of scalable keyword search on big RDF data, and proposed a new solution based on summary: (i) we construct a concise summarization at the type level from RDF data; (ii) during query evaluation, we leverage the summarization to prune away a significant portion of RDF data from the search space, and formulate SPARQL queries for efficiently accessing data. Furthermore, the proposed summarization can be incrementally updated as the data get updated. Experiments on both RDF benchmark and real RDF datasets showed that our solution is efficient, scalable, and portable across RDF engines. An interesting future direction is to extend the summarization for optimizing generic SPARQL queries on large RDF datasets.

So far, we have investigated two types of queries and studied their optimization techniques for RDF data. Yet challenges also come from managing the constantly evolving RDF data. RDF data are known to contain rich temporal semantics. To cope with the ever-increasing temporal and multiversion data, an attractive way is to partition and store the RDF data in a distributed and parallel framework. A paramount concern in the distributed and parallel computation is to achieve load balancing. This naturally leads us to study an efficient strategy to find good partitions for large temporal RDF data. In the next chapter, we will present our work for finding the optimal splitters on large temporal and multiversion RDF data.

CHAPTER 6

OPTIMAL SPLITTERS IN TEMPORAL AND MULTIVERSION RDF DATA

6.1 Introduction

Increasingly, semantic web applications request the storage and processing of historical values in a database, to support various auditing, provenance, mining, and querying operations for better decision making. RDF data are known to contain rich temporal semantics. For instance, one of the popular datasets in the Linked Open Data project is the LinkedSensorData [6], which is an RDF dataset containing the historical sensor readings such as temperature, visibility, precipitation, *etc.* from about 20,000 weather stations in the United States. These sensor data originated from the MesoWest project [7] within the University of Utah, which has maintained the historical climate data since 1997. Together, the LinkedSensorData has more than 1 billion triples – at a scale that is very inefficient to store and process in a centralized database system [81].

Given the fast development in distributed and parallel processing frameworks, applications can now afford collecting, storing, and processing large amounts of multiversioned or temporal values from a long running history. Naturally, it leads to the development of multiversion databases and temporal databases for massive temporal RDF data. In these databases, an object o is associated with multiple disjoint temporal intervals in the form $[s, e]$, each of which is associated with the valid value(s) of o during the period of $[s, e]$.

Consider two specific examples as shown in Figure 6.1. Figure 6.1(a) shows a multiversion database [24, 77], and Figure 6.1(b) shows a temporal database, where each object is represented by a piecewise linear function. A multiversion database keeps all the historical values of an object. A new interval with a new value is created whenever an update or an insertion to an object has occurred. An existing interval with a (now) old value terminates when an item has been deleted or updated.

On the other hand, for large temporal or time-series data, we can represent the value of a temporal object as an arbitrary function $f : \mathbb{R} \to \mathbb{R}$ (time to value). In general, for arbitrary

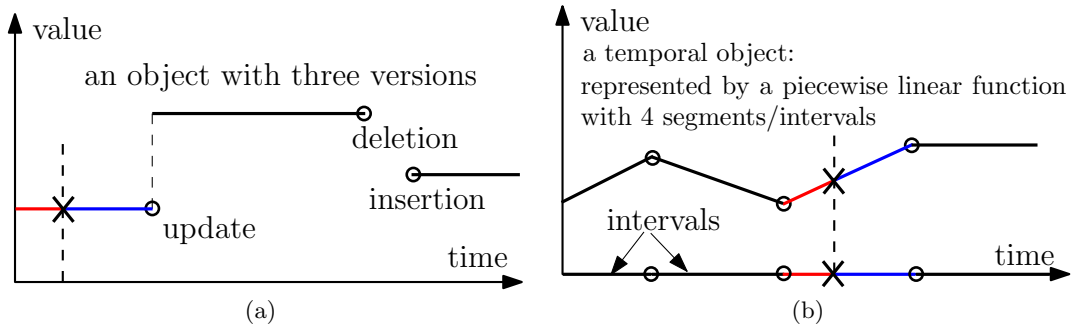


Figure 6.1: Databases with intervals (a) Multiversion database and (b) Temporal database

temporal data, f can be expensive to describe and process. In practice, applications often approximate f using a piecewise linear function g [19, 34, 66]. The problem of approximating an arbitrary function f by a piecewise linear function g has been extensively studied (see [19, 34, 66, 84] and references therein). Other functions can be used for approximation as well, such as a piecewise polynomial function, for better approximation quality. The key observations are: 1) more intervals lead to better approximation quality, but also are more expensive to represent; 2) adaptive methods, by allocating more intervals to regions of high volatility and less to smoother regions, are better than nonadaptive methods with a fixed segmentation interval.

How to approximate f with g for a temporal or time series object is beyond the scope of this dissertation, and we assume that the data have already been converted to a set of intervals, where each interval is associated with a mapping function (piecewise linear, or piecewise polynomial) by *any* segmentation method. In particular, we do not require objects in a dataset to have the same number of intervals nor do we require intervals from different objects to have aligned starting/ending points. Thus, it is possible that the data are collected from a variety of sources after applying different preprocessing modules.

Lastly, large interval data may also be produced by any time-based or range-based partitioning of an object, such as a log file or a spatial object, from a big dataset.

That said, we observe that in the aforementioned applications, users often have to deal with big interval data. Meanwhile, storing and processing big data in a cluster of (commodity) machines, to tap the power of parallel and distributed computation, is becoming increasingly important. Therefore, storing and processing the large number of intervals in a distributed store is a paramount concern in enabling the above applications to leverage the storage space and the computation power from a cluster.

Since most analytical tasks and user queries concerning interval data in a multiversion or temporal database are time-based, *e.g.,* find the object ids with valid values in $[50, 100]$ at

time-instance t, the general intuition is to partition the input dataset into a set of buckets based on their time-stamps. Intervals from one bucket are then stored in one node and processed by one core from a cluster of (commodity) machines. By doing so, user queries or transactions concerning a time instance or a time range can be answered in selected node(s) and core(s) independently without incurring excessive communication, which also dramatically improves the spatial and temporal locality of caching in query processing.

A challenge is to achieve load-balancing in this process, *i.e.,* no single node and core should be responsible for storing and processing too many intervals. In particular, *given the number of buckets to create, the size of the maximum bucket should be minimized.* This is similar to the concept of *optimal splitters* in databases with points [95] or array datasets [67, 79]. However, the particular nature of interval datasets introduces significant new challenges.

Specifically, a partitioning boundary (known as a splitter) may split an interval into two intervals. As a result, buckets on both sides of a splitter need to contain an interval that intersects with a splitter; see examples in Figure 6.1 where the dashed line with a cross represents a splitter. Furthermore, intervals from different objects may overlap with each other, which complicates the problem of finding the optimal splitters. Finally, a good storage scheme should also be capable to handle ad-hoc updates gracefully. In contrast, in a point or array dataset, any element from the original dataset will only lead to one element in one of the buckets, and elements do not overlap with each other.

Our contributions can be summarized as follows. Given n objects with a total of N intervals from all objects, and a user-defined budget k for the number of buckets to create, a baseline solution is a dynamic programming formulation with a cost of $O(kN^2)$. However, this solution is clearly not scalable for large datasets. Our goal is to design I/O and computation efficient algorithms that work well regardless if data fit in main memory or not. A design principle we have followed is to leverage on existing indexing structures whenever possible (so these methods can be easily adopted in practice). Specifically, we make the following contributions:

- We formulate the problem of finding optimal splitters for interval data in Section 6.2.
- We present a baseline method using a dynamic programming in Section 6.3.
- We design efficient methods for memory-resident data in Section 6.4. Our best method finds optimal splitters of a dataset for any budget values k in only $O(N \log N)$ cost.
- We investigate external memory methods for large disk-based data in Section 6.5. Our best method finds optimal splitters of a dataset for any budget value k in only

$O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ IOs, where B is the block size and M is the memory size.

- We extend our methods to work for the queryable version of the optimal splitters problem in Section 6.6, where k is the query parameter. We also discuss how to make our methods dynamic to handle ad-hoc updates in Section 6.6.
- We examine the efficiency of the proposed methods with extensive experiments in Section 6.7, where large real RDF datasets with hundreds of millions of intervals were tested.

We survey related work in Section 6.8, and conclude in Section 6.9. A part of this chapter also appears in our work [73].

6.2 Problem Formulation

Let the database D be a set of objects. Each object has an *interval attribute* (*e.g.,* time) whose universe U is the real domain representable in a computer. An object's interval attribute contains a sequence of nonoverlapping intervals, as demonstrated in Figure 6.1. Given an interval $[s, e]$, we refer to both s and e as its *endpoints*, and s (e) as its *starting* (*ending*) *value*. Although the intervals from one object are disjoint, the intervals of different objects may overlap. Let N be the total number of intervals from *all* objects in D. Denote by I the set of these N intervals.

The objective is to partition I into smaller sets so that they can be stored and processed in a distributed and parallel fashion. A *size-k partition P over I*, denoted as $P(I, k)$, is defined as follows:

1. P contains m *splitters*, where $0 \leq m \leq k$. Each splitter is a vertical line that is orthogonal to the interval dimension at a *distinct* value $\ell \in U$. We will use ℓ to denote the splitter itself when there is no confusion. Let the splitters in P be $\ell_1, \ldots, \ell_m$ in ascending order, and for convenience, also define $\ell_0 = -\infty$, $\ell_{m+1} = \infty$. These splitters induce $m + 1$ buckets $\{b_1, \ldots, b_{m+1}\}$ over I, where b_i ($1 \leq i \leq m + 1$) represents the interval $[\ell_{i-1}, \ell_i]$.
2. If $s \neq e$, an interval $[s, e]$ of I is assigned to a bucket b_i ($1 \leq i \leq m+1$), if $[s, e]$ has an intersection of *non-zero length* with the interval of b_i. That is, the intersection cannot be a point (which has a zero length).

   If $s = e$, $[s, e]$ degenerates into a point, in which case we assign $[s, e]$ to the bucket whose interval contains it. In the special case where $s = \ell_i$ for some $i \in [1, m]$ (*i.e.,* the point lies at a splitter), we follow the convention that $[s, e]$ is assigned to b_{i+1}.

3. We will regard b_i as a set, consisting of the intervals assigned to it. The size of bucket b_i, denoted as $|b_i|$, gives the number of such intervals.

Define the *cost of a partition* P with buckets $b_1, \ldots, b_{m+1}$ as the size of its maximum bucket:

$$c(P) = \max\{|b_1|, \ldots, |b_{m+1}|\}. \tag{6.1}$$

Since the goal is to partition I for storage and processing in distributed and parallel frameworks, a paramount concern is to achieve load-balancing, towards which a common objective is to minimize the maximum load on any node, so that there is no single bottleneck in the system. The same principle has been used for finding optimal splitters in partitioning points [95] and array datasets [67, 79]. That said, an optimal partition for I is formally defined as follows.

Definition 4 *An optimal partition of size k for I is a partition $P^*(I,k)$ with the smallest cost, i.e.,*

$$P^*(I,k) = \operatorname*{argmin}_{P \in \mathcal{P}(I,k)} c(P) \tag{6.2}$$

where $\mathcal{P}(I,k)$ is the set of all the size-k partitions over I.

$P^*(I,k)$ is thus referred to as an *optimal partition*. If multiple partitions have the same optimal cost, $P^*(I,k)$ may represent any one of them. In what follows, when the context is clear, we use P^* and P to represent $P^*(I,k)$ and $P(I,k)$, respectively.

Note that it is an equivalent definition if one defines $P(I,k)$ in step 2 such that bucket b_i gets assigned only the *intersection* of $[s,e]$ with b_i – namely, only a *portion* of $[s,e]$ is assigned to b_i, instead of the entire $[s,e]$. This, however, does not change the number of intervals stored at b_i, which therefore gives rise to the same partitioning problem. Keeping $[s,e]$ entirely in b_i permits conceptually cleaner and simpler discussion (because it removes the need of remembering to take intersection). Hence, we will stick to this problem definition in presenting our solutions.

Consider the example from Figure 6.2, where I contains 9 intervals from 3 objects. When $k = 2$, the optimal splitters are $\{\ell_1, \ell_2\}$, which yields 3 buckets b_1, b_2, b_3 with 3, 4, and 3 intervals, respectively; hence, $c(P^*) = 4$. An alternative partition of 2 splitters is also shown in Figure 6.2 with dashed lines. It induces 3 buckets with 3, 4, and 5 intervals, so its cost 5 is worse than the aforementioned optimal partition. Note that we use t^* to represent $c(P^*)$.

Depending on whether k is given apriori or not, there are two versions of the optimal splitters problem. In the first case, k is fixed and given at the same time with the input

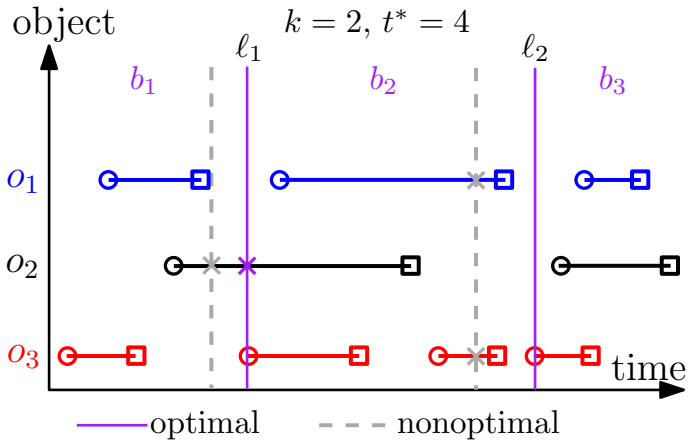


Figure 6.2: An example

set I. This fits the scenario where the number of nodes/cores in a cluster available for processing I is already known. In the second case, a user is not certain yet how many nodes/cores should be deployed. Thus, he/she wants to explore the cost of the optimal partition for different budget values of k. In this case, k is unknown, and we are allowed to preprocess I in order to quickly find the optimal splitters for any subsequent queries over I with different k values. Formally:

Problem 1 *The static interval splitters problem is to find P^* and $c(P^*)$ for an interval set I and a fixed budget value k.*

Problem 2 *The queryable interval splitters problem is to find P^* and $c(P^*)$ over an interval set I, for any value k that is supplied at the query time as a query parameter.*

Clearly, any solution to the queryable version can be applied to settle the static version, and vice versa (by treating I and k as a new problem instance each time a different k is supplied). However, such brute-force adaptation is unlikely to be efficient in either case. To understand why, first note that, in Problem 2, the key is to preprocess I into a suitable structure so that all subsequent queries can be answered fast (the preprocessing cost is *not* a part of a query's overhead). In Problem 1, however, such preprocessing cost must be counted in an algorithm's overall running time, and can be unworthy because we only need to concentrate on a single k, thus potentially avoiding much of the computation in the preprocessing aforementioned (which must target all values of k).

We investigate the static problem in Sections 6.3, 6.4, and 6.5, and the queryable problem in Section 6.6. In the queryable problem, handling updates in I becomes an important issue,

and presents additional challenges, which are also tackled in Section 6.6.

6.3 A Baseline Method

Let us denote the N intervals in I as $[s_1, e_1], \ldots, [s_N, e_N]$. Suppose, without loss of generality, that $s_1 \leq ... \leq s_N$. Let $S(I) = \{s_1, \ldots, s_N\}$, *i.e.,*, the set of starting values of the intervals in I. When I is clear from the context, we simply write $S(I)$ as S. For any splitter ℓ, denote by $\ell(1)$ the splitter that is placed at the smallest starting value in S that is larger than or equal to ℓ. If no such starting value exists, $\ell(1)$ is undefined. Note that if ℓ itself is a starting value, $\ell(1) = \ell$. It turns out that, to minimize the cost of a partition over I, it suffices to place splitters only at the values in S. This is formally stated in the next two lemmas:

Lemma 14 *Consider any partition P with distinct splitters $\ell_1, \ldots, \ell_m$ in ascending order, such that $\ell_m(1)$ is undefined. Let P' be a partition with splitters $\ell_1, \ldots, \ell_{m-1}$. Then, it always holds that $c(P') = c(P)$.*

Proof. Let b_m and b_{m+1} be the last two buckets in P (which are separated by ℓ_m), and b'_m be the last bucket of P' (which is to the right of ℓ_m). When $\ell_m(1)$ is undefined, there are no new intervals starting to the right of ℓ_m. Hence, it is not hard to show that $b_{m+1} \subseteq b_m = b'_m$. This proves the lemma because all the other buckets in P exist directly in P'. ■

If s_{max} is the largest starting value in S, the above lemma suggests that we can drop all those splitters greater than s_{max} without affecting the cost of the partition. Hence, it does not pay off to have such splitters.

Lemma 15 *Consider any partition P with distinct splitters $\ell_1, \ldots, \ell_m$ in ascending order, such that $\ell_m(1)$ exists. Let ℓ_i be the largest splitter that does not belong to S (i.e., $\ell_j \in S$ for all $j < i \leq m$). Define P' as a partition with splitters*

- *$\ell_1, \ldots, \ell_{i-1}, \ell_i(1), \ell_{i+1}, \ldots \ell_m$, if $\ell_i(1) \neq \ell_{i+1}$;*
- *$\ell_1, \ldots, \ell_{i-1}, \ell_{i+1}, \ldots \ell_m$, otherwise.*

Then, it always holds that $c(P') \leq c(P)$.

Proof. We consider only the first bullet because the case of the second bullet is analogous. Let b_i (b_{i+1}) be the bucket in P that is on the left (right) of ℓ_i. Similarly, let b'_i (b'_{i+1}) be the bucket in P' that is on the left (right) of $\ell_i(1)$. We will show that $b'_i \subseteq b_i$ and $b'_{i+1} \subseteq b_{i+1}$,

whose correctness implies $c(P') \leq c(P)$, because all the other buckets in P still exist in P', and vice versa.

We prove only $b'_i \subseteq b_i$ because a similar argument validates $b'_{i+1} \subseteq b_{i+1}$. Given an interval $[s, e]$ assigned to b'_i, we will show that it must have been assigned to b_i, too. Note that, by definition of $\ell_i(1)$, *no starting value can exist in* $[\ell_i, \ell_i(1))$. This means that $s < \ell_i$ because s would not be assigned to b'_i if $s = \ell_i(1)$.

If $s = e$, then s must fall in $[\ell_{i-1}, \ell_i)$, which means that s is also assigned to b_i. On the other hand, if $s \neq e$, a part of $[s, e]$ needs to intersect (ℓ_{i-1}, ℓ_i) so that $[s, e]$ can have intersection of non-zero length with b'_i. This means that $[s, e]$ also has non-zero-length intersection with b_i, and therefore, is assigned to b_i. ■

This shows that if a partition has a splitter that is not in S, we can always "snap" the splitter to a value in S, without increasing the cost of the partition (may decrease the cost of the partition). Next, we introduce a solution based on dynamic programming. Given a splitter ℓ, we define $I^-(\ell)$, $I^+(\ell)$, and $I^o(\ell)$ as the subset of intervals in I whose starting values are less than, greater than, and equal to ℓ, respectively:

$$\begin{aligned} I^-(\ell) &= \{[s_i, e_i] \in I \mid s_i < \ell\} \\ I^+(\ell) &= \{[s_i, e_i] \in I \mid s_i > \ell\} \\ I^o(\ell) &= \{[s_i, e_i] \in I \mid s_i = \ell\}. \end{aligned}$$

An interval $[s, e]$ is said to *strongly cover* a splitter ℓ if $s < \ell < e$ (note that both inequalities are strict). Let $I^\times(\ell)$ be the subset of intervals from I that strongly cover ℓ:

$$I^\times(\ell) = \{[s_i, e_i] \in I \mid s_i < \ell < e_i\}.$$

The following fact paves the way to a dynamic programming algorithm solving Problem 1:

Lemma 16 *If $k = 0$, $c(P^*(I, k)) = |I|$. For $k \geq 1$, $c(P^*(I, k)) =$*

$$\min\Big\{|I|,\ \min_{\ell \in S}\{\max\{c(P^*(I^-(\ell), k-1)), \lambda\}\}\Big\}. \tag{6.3}$$

where $\lambda = |I^o(\ell)| + |I^+(\ell)| + |I^\times(\ell)|$.

Proof. The case of $k = 0$ is obvious, so we concentrate on $k \geq 1$. A partition of size-k over I is allowed to use m splitters, where m ranges from 0 to k. If $m = 0$, then apparently the partition has cost $|I|$. The rest of the proof considers $m \geq 1$.

Lemmas 14 and 15 indicate that it suffices to consider partitions where all splitters fall in S. Let us fix a starting value $x \in S$. Consider an arbitrary partition $P(I, k)$ *whose*

last splitter ℓ is at position x. Let $m \in [1, k]$ be the number of splitters in $P(I, k)$. Let $P(I^-(x), k-1)$ represent the partition over $I^-(x)$ with the first $m-1$ splitters of $P(I, k)$. Denote by $b'_1, ..., b'_m$ the buckets of $P(I^-(x), k-1)$ in ascending order.

Let $b_1, ..., b_{m+1}$ be the buckets of $P(I, k)$ in ascending order. As $P(I, k)$ shares the first $m-1$ splitters with $P(I^-(x), k-1)$, we know $b_i = b'_i$ for $1 \leq i \leq m-1$. Next, we will prove:

- Fact 1: $b_m = b'_m$
- Fact 2: $|b_m + 1| = |I^o(x)| + |I^+(x)| + |I^\times(x)|$.

These facts indicate:

$$\begin{aligned} c(P(I,k)) &= \max\{|b_1|, ..., |b_{m+1}|\} \\ &= \max\{|b'_1|, ..., |b'_m|, |b_{m+1}|\} \\ &= \max\{c(P(I^-(x), k-1)), |b_{m+1}|\} \\ &\geq \max\{c(P^*(I^-(x), k-1)), |b_{m+1}|\} \end{aligned}$$

where the equality can be achieved by using an optimal partition $P^*(I^-(x), k-1)$ to replace $P(I^-(x), k-1)$. Then, the lemma follows by minimizing $c(P(I, k))$ over all possible x.

Proof of Fact 1. We will prove only $b_m \subseteq b'_m$ because a similar argument proves $b'_m \subseteq b_m$. Let $[x', x]$ be the interval of b_m. Accordingly, the interval of b'_m is $[x', \infty)$. Consider an interval $[s, e]$ of I that is assigned to b_m. If $s = e$, then it must hold that $x' \leq s < x$ (note that $s \neq x$; otherwise, $[s, e]$ is assigned to b_{m+1}), which means that $[s, e] \in I^-(x)$, and hence, $[s, e]$ is assigned to b'_m. On the other hand, if $s \neq e$, then $[s, e]$ has a non-zero-length intersection with $[x', x]$, implying that $[s, e]$ intersects (x', x). Hence, $s < x$, and $[s, e]$ has a non-zero-length intersection with $[x', \infty)$. This proves that $[s, e]$ is also assigned to b'_m.

Proof of Fact 2. By definition, an interval $[s, e]$ in I is assigned to b_{m+1} in three disjoint scenarios: 1) $s > x$, 2) strongly covers x, and 3) $s = x$. The numbers of intervals in these scenarios are given precisely by $I^+(x)$, $I^\times(x)$, and $I^o(x)$, respectively. ■

We are now ready to clarify our dynamic programming algorithm, which aims to fill in an N by k matrix, as shown in Figure 6.3. Cell $[i, j]$ (at the ith row, jth column) records the optimal cost for a subproblem $c(P^*(I(i), j))$, where $I(i)$ denotes the set of intervals in I with ids 1, ..., i (we index intervals in I in ascending order of their starting values, break ties first by ascending order of their ending values, and then arbitrarily). Thus, Cell $[i, j]$ represents the optimal cost of partitioning $I(i)$ using up to j splitters.

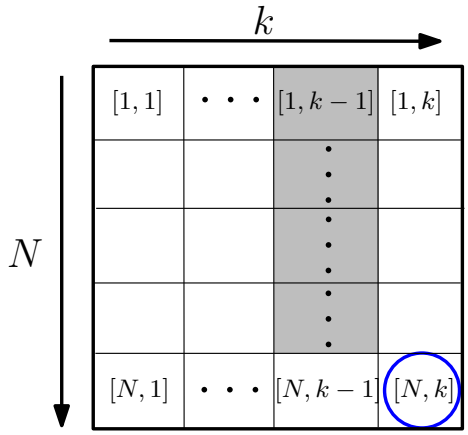


Figure 6.3: The DP method

Specifically, we fill the matrix starting from the top-left corner to the bottom-right corner. To fill the cost in Cell $[i, j]$, according to Lemma 16, the last splitter in a partition $P(I(i), j)$ may be placed at any value in $\{s_1, ..., s_i\}$. Hence, one needs to check $i - 1$ cells from the $(j - 1)$th column, *i.e.,* from $[1, j - 1]$ to $[i - 1, j - 1]$. For instance, in Figure 6.3, to find the value for the cell $[N, k]$, we need to check $N - 1$ cells from the previous column (the gray cells). We refer to this approach as the *DP* method.

The above algorithm can be slightly extended in the straightforward manner to remember also the partitions found, so that the DP method outputs both P^* and $c(P^*)$.

The cost of the dynamic programming method can be summarized as follows. Sorting I by starting values takes $O(N \log N)$ time. Next, we focus on the cost of DP. Given a splitter ℓ_j, using ideas to be explained in Section 6.4, we can determine λ in $O(1)$ time with the help of a structure that can be built in $O(N \log N)$ time. Hence, to fill in Cell $[i, j]$, it requires to check $i - 1$ cells in the preceding column plus $O(1)$ cost for obtaining λ. Completing an entire column thus incurs $O(\sum_{i=1}^{N} i) = O(N^2)$ time. As k columns need to be filled, the overall running time is $O(kN^2)$.

6.4 Internal Memory Methods

The DP method clearly does not scale well with the database size N due to its quadratic complexity. In this section, we develop a more efficient algorithm for Problem 1, assuming that the database can fit in memory.

The decision version of our problem is what we call the *cost-t splitters problem*: determine whether there is a size-k partition P with $c(P) \leq t$, where t is a positive integer given as an input parameter. If such P exists, t is *feasible*, or otherwise, *infeasible*. If t is feasible, we define $\bar{t}$ as an arbitrary value in $[1, t]$ such that there is a size-k partition P with $c(P) = \bar{t}$,

i.e.,

$$\bar{t} = \text{an arbitrary } x \in [1, t] \text{ s.t. } \exists P \in \mathcal{P}(k, I), c(P) = x. \tag{6.4}$$

When t is infeasible, define $\bar{t} = 0$. An algorithm solving the cost-t splitters problem is required to output $\bar{t}$, and if $\bar{t} > 0$ (*i.e.,* t is feasible), also a P with $c(P) = \bar{t}$. The following observation follows immediately the above definitions:

Lemma 17 *If t is infeasible, then any $t' < t$ is also infeasible.*

A trivial upper bound of t is N. Hence, the above lemma suggests that we can solve Problem 1 by carrying out a binary search to determine the smallest feasible t in $[1, N]$. This requires solving $O(\log N)$ instances of the cost-t splitters problem. In the sequel, we will show that each instance can be settled in $O(k)$ time, using a structure constructable in $O(N \log N)$ time. This gives an algorithm for Problem 1 with $O(N \log N + k \log N) = O(N \log N)$ overall running time.

As before, we denote the intervals in I as $[s_1, e_1], ..., [s_N, e_N]$ sorted in nondescending order of their starting values, break ties by nondescending order of their ending values first, and then arbitrarily. Interval $[s_i, e_i]$ is said to have *id* i. We consider that I is given in an array where the ith element is $[s_i, e_i]$ for $1 \leq i \leq N$.

In what follows, we introduce a concept named *stabbing-count array*. As will be clear shortly, the key to attacking the cost-t splitters problem is to construct a stabbing-count array A. For each $i \in [1, N]$, define $\sigma[i]$ as the number of intervals in I strongly covering the value s_i, *i.e.,*

$$\sigma[i] = |I^{\times}(s_i)|. \tag{6.5}$$

Furthermore, define $\delta[i]$ as the number of intervals in I with starting values equal to s_i but with ids less than i. Formally, if $I^o_<(s_i) = \{[s_j, e_j] \in I \mid s_j = s_i \wedge j < i\}$, then

$$\delta[i] = |I^o_<(s_i)|. \tag{6.6}$$

A stabbing-count array A is simply an array of size N where $A[i] = (\sigma[i], \delta[i])$, $1 \leq i \leq N$. Figure 6.4 shows an example where I has $N = 9$ intervals, and the values of $\sigma[i]$, $\delta[i]$ have been shown under the interval with id i. For instance, $\sigma[4] = 1$ because there is one interval $[s_3, e_3]$ strongly covering s_4, whereas $\delta[6] = 1$ because one interval $[s_5, e_5]$ has the same starting value as $[s_6, e_6]$, and yet, has a smaller id than $[s_6, e_6]$.

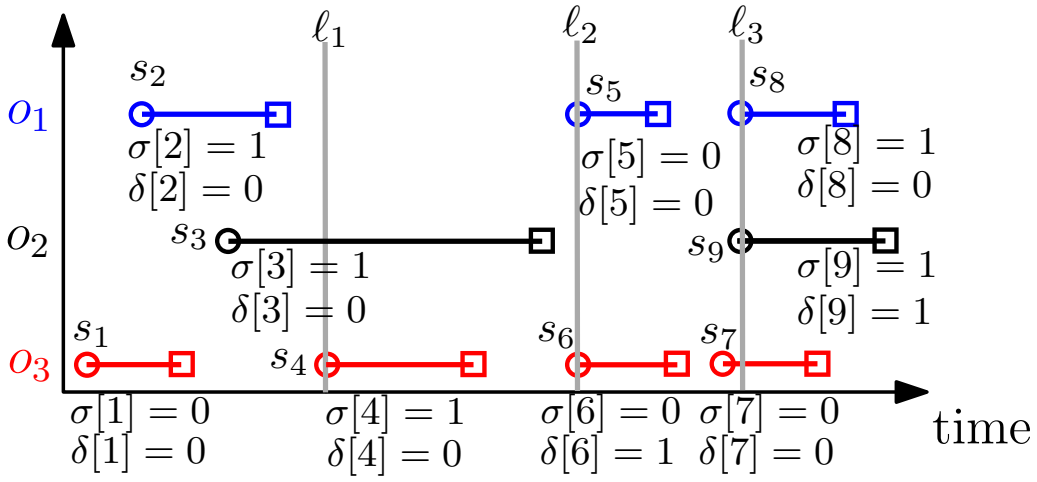


Figure 6.4: Stabbing-count array

Lemma 18 *The stabbing-count array can be built in $O(N \log N)$ time.*

Proof. This is done with a sweeping technique. First, use $O(N \log N)$ time to sort in ascending order the $2N$ endpoints (starting and ending values) of the N intervals in I, breaking ties as follows:

- A starting value is always put before an ending value;
- If both endpoints are starting values, the one belonging to an interval with a smaller id is put earlier (similarly, if both endpoints are ending values).

Denote by E the sorted list. For each endpoint $x \in E$, we associate it with the id i of the interval where x belongs, and if x is an ending value, also with the starting value s_i of that interval.

Next, we scan E once by its ordering. In this process, an interval $[s, e]$ is *alive* if s has been scanned, but e has not. At any moment, we keep track of 1) s_{last}: the last starting value scanned, 2) the number c of alive intervals, and 3) the number c_δ of alive intervals whose starting values equal s_{last}. As we will see, all this information can be updated in constant time per endpoint in E, and allows us to generate an entry of stabbing-count array A in constant time after a starting value is scanned.

Specifically, at the beginning of the scan, $c = c_\delta = 0$ and $s_{last} = -\infty$. Let x be the next endpoint to be scanned, and i the id of the interval to which x belongs. We proceed as follows:

- Case 1: x is an ending value e_i. Decrease c by 1. Furthermore, if $s_i = s_{last}$, also decrease c_δ by 1.

- Case 2: x is a starting value. Increase c by 1. Furthermore, if $x = s_{last}$, increase c_δ by 1; otherwise, set s_{last} to x, and reset c_δ to 1. We also generate an entry $(\sigma[i], \delta[i])$ of A, where $\sigma[i] = c - c_\delta$, and $\delta[i] = c_\delta - 1$.

The scan clearly takes $O(N)$ time, thus completing the proof. ■

Given the stab-counting array A, we now describe an algorithm, called *t-jump*, for solving the cost-t splitters problem in $O(k)$ time. If t is feasible, our algorithm outputs $\bar{t}$ and a partition P with splitters $\ell_1, ..., \ell_m$, where m is some integer between 0 and k, such that $c(P) = \bar{t} \leq t$. Each ℓ_j $(1 \leq j \leq m)$ is a starting value of a certain interval in I. We denote by $p(j)$ the id of that interval, namely, $\ell_j = s_{p(j)}$.

Algorithm 8 illustrates the details of t-jump. At a high level, the main idea is to place the splitters in ascending order, and particularly, in such a way that the next splitter is pushed as far away from the preceding one as possible, aiming to let the new bucket have size exactly t (hence, the name t-jump). However, as we will see, due to the overlapping among intervals, the aim is not always achievable. When it is not, we need to settle for a bucket with a smaller size, by moving the next splitter backwards, but just enough to make

Algorithm 8: t-jump (I, k, t)

```
1  if t ≥ N then
2      return t̄ = N, and an empty splitter set;
3  p(1) = t + 1 − δ[t + 1]; |b_1| = t − δ[t + 1];
4  if |b_1| = 0 then
5      return t̄ = 0;
6  t̄ = |b_1|;
7  for j = 2, ..., k do
8      x_j = p(j − 1) + t − σ[p(j − 1)];
9      if x_j > N then
10         |b_j| = N − p(j − 1) + 1 + σ[p(j − 1)]; t̄ = max(t̄, |b_j|);
11         return t̄, and splitters s_{p(1)}, ..., s_{p(j−1)};
12     p(j) = x_j − δ[x_j];
13     if p(j) = p(j − 1) then
14         return t̄ = 0;
15         /* p(j) < p(j − 1) cannot happen */
16     |b_j| = t − δ[x_j]; t̄ = max(t̄, |b_j|);
17 |b_{k+1}| = N − p(k) + 1 + σ[p(k)];
18 if |b_{k+1}| > t then
19     return t̄ = 0;
20 t̄ = max(t̄, |b_{k+1}|);
21 return t̄, and splitters s_{p(1)}, ..., s_{p(k)}.
```

the new bucket's size drop below t.

Now, let us walk through the algorithm. In the outset, lines 1-2 deal with the trivial case where $t \geq N$ (such t is always feasible, even if no splitter is used). Lines 3-6 place the first splitter ℓ_1 at the *largest starting value that guarantees* $|b_1| \leq t$. The rationale of these lines is from the next lemma:

Lemma 19 *We have:*

- *For $\ell_1 = s_{p(1)}$ where $p(1)$ is set as in line 3, $|b_1| = t - \delta[t+1] \leq t$. On the other hand, if $\ell_1 > s_{p(1)}$, $|b_1| > t$.*
- *If $|b_1|$ as set in line 3 equals 0, t is infeasible.*

Proof. To prove the first sentence in the first bullet, note that b_1 does not include the interval with id $p(1)$. The bucket includes exactly the intervals with ids 1, ..., $p(1) - 1$, namely, $t - \delta[t+1]$ of them. To prove the second sentence, if $\ell_1 > s_{p(1)}$, b_1 includes at least the first $t+1$ intervals, and hence, $|b_1| > t$.

Now we show that the second bullet is also true. In fact, since $t = \delta[t+1]$, the intervals with ids 1, ..., $t+1$ all have the same starting values. These intervals will be assigned to an identical bucket in any partition. This bucket must have a size at least $t+1$. ■

Hence, if $|b_1| = 0$, the algorithm terminates at line 5, indicating that t is infeasible. Otherwise (*i.e.,* $|b_1| > 0$), we know that the cost of the current partition is $\bar{t} = |b_1|$ (*i.e.,* equals to the largest bucket, line 6).

For every $j \in [2, k]$, lines 8-16 determine splitter ℓ_j, assuming $\ell_1, ..., \ell_{j-1}$ are already available. These lines set ℓ_j to *largest starting value that guarantees* $|b_j| \leq t$, based on the next lemma:

Lemma 20 *We have:*

- *For $\ell_j = s_{p(j)}$ where $p(j)$ is set as in line 12, $|b_j| = t - \delta[x_j] \leq t$. On the other hand, if $\ell_j > s_{p(j)}$, $|b_j| > t$.*
- *If $p(j) = p(j-1)$ at line 13, t is infeasible.*

Proof. Regardless of the position of ℓ_j, b_j *must* contain $\sigma[p(j-1)]$ intervals, *i.e.,* those in $I^{\times}(s_{p(j-1)})$ because they strongly cover $s_{p(j-1)}$, and hence, have non-zero-length intersection with b_j. If ℓ_j is placed at $s_{p(j)}$, then besides the intervals of $I^{\times}(s_{p(j-1)})$, b_j also includes those intervals with ids $p(j-1)$, ..., $p(j) - 1$. By definition of the stabbing-count array, $|I^{\times}(s_{p(j-1)})| = \sigma[p(j-1)]$. Hence, b_j contains in total $|I^{\times}(s_{p(j-1)})| + p(j) - p(j-1) = t - \delta[x_j]$ intervals. This proves the first sentence of the first bullet.

If $\ell_j > s_{p(j)}$, then besides $I^{\times}(s_{p(j-1)})$, b_j also includes at least the intervals with ids $p(j-1)$, ..., x_j (due to line 12 and the definition of the stabbing-count array for $\delta[i]$, $s_{p[j]} = s_{x[j]}$). In this case, $|b_j|$ is at least $x_j - p(j-1) + 1 + |I^{\times}(s_{p(j-1)})| = t + 1$. This proves the second sentence of the first bullet.

Next, we show that the second bullet is also true. Notice that $p(j) = p(j-1)$ implies $\delta[x_j] = t - \sigma[p(j-1)]$. By the way x_j is calculated, there are exactly $t - \sigma[p(j-1)] - 1$ starting values between $s_{p(j-1)}$ and s_{x_j}. Hence, since $\delta[x_j] > t - \sigma[p(j-1)] - 1$, we know that the interval with id $p(j-1)$ is counted by $\delta[x_j]$ (*i.e.,* the interval belongs to $I^o_<(x_j)$). This means that $s_{p(j-1)} = s_{x_j}$.

Now we have found $\sigma[p(j-1)]$ intervals of I strongly covering $s_{p(j-1)}$, in addition to $\delta(x_j) + 1$ intervals whose starting values are $s_{p(j-1)}$ (the $+1$ is due to the interval with id x_j). These $\sigma[p(j-1)] + \delta(x_j) + 1 = t + 1$ intervals will always be assigned to an identical bucket in any partition. Therefore, no partition can have cost at most t. ■

Hence, lines 13-15 declare t infeasible if $p(j) = p(j-1)$. Otherwise, line 16 correctly sets $|b_j|$, and updates the current partition cost $\bar{t}$ if necessary.

Now let us focus on lines 9-11. They handle the scenario where *less than* k splitters are needed to obtain a partition of cost at most t. Specifically, the final partition has only $j-1$ splitters. Line 10 determines the size of the last bucket, and adjusts the partition cost $\bar{t}$ accordingly. The algorithm terminates at line 11 by returning the results.

At line 17, we have obtained all the k splitters, and hence, the last bucket b_{k+1} has been automatically determined. The line computes its size, while lines 18-20 check its feasibility (a negative answer leads to termination at line 19), and update the partition cost if needed. Finally, line 21 returns the k splitters already found, as well as the cost $\bar{t}$ of the partition.

We illustrate the algorithm with the example in Figure 6.4, setting $t = k = 3$. To find the first splitter ℓ_1, we compute $p(1) = 4$, and hence, place ℓ_1 at s_4. To look for the second splitter ℓ_2, line 8 calculates $x_2 = p(1) + t - \sigma[p(1)] = 4 + 3 - 1 = 6$. However, as $\delta[6] = 1 > 0$, we move $p(2)$ back to $x_2 - \delta[x_2] = 6 - 1 = 5$. For the third splitter ℓ_3, we derive $p(3) = p(2) + t - \sigma[p(2)] = 5 + 3 - 0 = 8$. Finally, the algorithm checks the size of the last bucket, which is $N - p(3) + 1 + \sigma[p(3)] = 9 - 8 + 1 + 1 = 3$, namely, still within the target cost t. Hence, t-jump outputs $\bar{t} = 3$, and splitters $\ell_1 = s_4, \ell_2 = s_5$, and $\ell_3 = s_8$.

Lemma 21 *If t-jump does not return $\bar{t} = 0$, then the splitters output constitute a partition with cost $\bar{t} \leq t$. Otherwise (i.e., $\bar{t} = 0$), then t must be infeasible.*

Proof. The first sentence is easy to show, given that the sizes of all buckets have been explicitly given in Algorithm 8. Next, we focus on the case where $\bar{t} = 0$. The algorithm

may return $\bar{t} = 0$ at three places: lines 5, 14, and 19. Lemmas 19 and 20 have already shown that termination at lines 5 and 14 is correct. It thus remains to show that line 19 termination is also correct.

Assume, for the purpose of contradiction, that j-jump reports $\bar{t} = 0$ at line 19, but there exists a size-k partition P' over I such that $c(P') \leq t$. Suppose that P' has splitters $\ell'_1, \ldots, \ell'_m$ in ascending order for some $m \leq k$, which define $m+1$ buckets $b'_1, \ldots, b'_{m+1}$ again in ascending order. By Lemmas 14 and 15, we can consider that $\ell'_1, ..., \ell'_m$ are all starting values in S.

We will establish the following statement: *for each $j \in [1, m]$, t-jump always places the jth smallest splitter ℓ_j in such a way that $\ell_j \geq \ell'_j$* – referred to as the *key statement* henceforth. This statement will complete the proof of Lemma 21. To see this, notice that it indicates $\ell_k \geq \ell_m \geq \ell'_m$, which in turn means $|b_{k+1}| \leq |b'_{m+1}|$. However, as line 19 tells us $|b_{k+1}| > t$, it thus must hold that $|b'_{m+1}| > t$, thus contradicting the fact that $c(P') \leq t$.

We now prove the key statement by induction. As the base step, $\ell_1 = s_{p(1)} \geq \ell'_j$ because if not, then by Lemma 19 $|b'_1|$ must be strictly greater than t, violating $c(P') \leq t$.

Now assuming that the key statement is correct for $j = z$, next we show that it is also correct for $j = z + 1$. In fact, if $\ell_{z+1} = s_{p(z+1)} < \ell'_{z+1}$, then by Lemma 20, a bucket with interval $[\ell_z, \ell'_{z+1}]$ will have a size greater than t. However, since $\ell_z \geq \ell'_z$, we know that the interval $[l'_z, l'_{z+1}]$ of bucket b'_z contains $[\ell_z, \ell'_{z+1}]$, and therefore, $|b'_{z+1}|$ must also be greater than t, contradicting $c(P') \leq t$. This completes the proof of the key statement. ■

Algorithm t-jump in Algorithm 8 clearly runs in linear time to the number of splitters, *i.e.,* $O(k)$ time. Putting everything in this section together, we have arrived at the first main result of the chapter.

Theorem 6 *The static interval splitters problem can be solved in $O(N \log N)$ time in internal memory.*

6.5 External Memory Methods

This section discusses how to solve the static interval splitters problem I/O-efficiently when the input set I of intervals does not fit in memory. Our analysis will be carried out in the standard *external memory* model of computation [16]. In this model, a computer has M words of memory, and a disk has been formatted into *blocks* (a.k.a. pages) of size B words. An I/O operation either reads a block from the disk to memory, or conversely, writes a block in memory to the disk. The objective of an algorithm is to minimize the

number of I/Os. We assume $M \geq 3B$, *i.e.,* the memory is large enough to store at least 3 blocks of data.

Initially, the input set I is stored in a disk-resident array that occupies $O(N/B)$ blocks. When the algorithm finishes, we should have output the $m \leq k$ splitters of the final partition to a file of $O(k/B)$ blocks in the disk. Define:

$$SORT(N) \quad = \quad (N/B)\log_{M/B}(N/B).$$

It is well-known that sorting a file of N elements entails $O(SORT(N))$ I/Os by the textbook external sort algorithm. The rest of the section serves as the proof for the theorem below:

Theorem 7 *The static interval splitters problem can be solved using $O(SORT(N))$ I/Os in external memory.*

As before, denote the intervals in I as $[s_1, e_1]$, $[s_2, e_2]$, ..., $[s_N, e_N]$ in ascending order of s_i (break ties by ascending order of their ending values first, and then arbitrarily), $1 \leq i \leq N$, where $[s_i, e_i]$ is said to have id i. Henceforth, we consider that I is stored as a disk-resident array where the ith element is $[s_i, e_i]$. This can be fulfilled by simply sorting the original input I, whose cost is within the budget of Theorem 7.

A trivial solution is to adapt the main-memory algorithm. The previous section has settled the static interval splitters problem in $O(N \log N)$ time when the input I fits in memory. Recall that our algorithm has two steps: it first creates the stabbing-count array A in $O(N \log N)$ time, and then solves $O(\log N)$ instances of the cost-t splitters problem, spending $O(k)$ time on each instance.

In external memory, a straightforward adaptation gives an algorithm that performs $O(SORT(N) + \min(k, \frac{N}{B}) \log N)$ I/Os. Recall from Section 6.4 that the computation of the stabbing-count array A requires only sorting $2N$ values followed by a single scan of the sorted list. Hence, the first step can be easily implemented in $O(SORT(N))$ I/Os in external memory. The second step, on the other hand, trivially runs in $O(\min(k, \frac{N}{B}) \log N)$ I/Os, by simply treating the disk as virtual memory.

This algorithm is adequate when k is not very large. The term $\min(k, \frac{N}{B}) \log N$ is asymptotically dominated by $O(SORT(N))$ when $k = O\big(\frac{N}{B \log_2(M/B)}\big)$ (note that the base of the logarithm is 2). However, the solution falls short for our purpose of claiming a clean bound $O(SORT(N))$ for the entire range of $k \in [1, N]$ (as is needed for proving Theorem 7).

In practice, this straightforward solution can be expensive when k is large, which may happen in a cluster. Note that k could be the total number of cores in a cluster, when each

core is responsible for processing one bucket in a partition. So it is not uncommon to have k in a few thousand, or even tens of thousands in a cluster.

Next, we provide an alternative algorithm in the external memory that settles the the problem of finding optimal splitters using $O(SORT(N))$ I/Os.

In what follows, we study how to find the cost-t splitters in external memory efficiently. The *cost-t splitters problem* (defined in Section 6.4) determines whether there is a size-k partition P with cost $c(P) \leq t$; moreover, if P exists, an algorithm also needs to output such a partition (any P with $c(P) \leq t$ is fine). Assuming that the stabbing-count array A has been stored as a file of $O(N/B)$ blocks, next we explain how to solve this problem with $O(N/B)$ I/Os.

The algorithm implements the idea of our main-memory solution by scanning the arrays A and I synchronously once. Following the notations in Section 6.4, let $\ell_1, \ldots, \ell_m$ be the splitters of P (where $1 \leq m \leq k$), and $p(i)$ an interval id such that $\ell_i = s_{p(i)}$, $1 \leq i \leq m$. We start by setting $p(1)$ as in Line 3 of Algorithm 8, fetching the $p(1)$th interval of I, and writing it to an output file. Iteratively, having obtained $p(i)$, for $1 \leq i \leq m-1$, we forward the scan of A to $A[p(i)]$, and compute $p(i+1)$ according to Lines 8-16 in Algorithm 8. Then, we forward the scan of I to retrieve the $p(i+1)$th interval of I, and append it to the output file. Recall that the main-memory algorithm would declare the absence of a feasible P under several situations. In external memory, when any such situation occurs, we also terminate with the absence declaration, and destroy the output file.

The total cost is $O(N/B)$ I/Os because we never read the same block of I or A twice. The algorithm only requires keeping $O(1)$ information in memory. In particular, among $p(1), \ldots, p(i)$ (suppose $p(i+1)$ is not available yet), only $p(i)$ needs to be remembered. We will refer to $p(i)$ as the *front-line value* of the algorithm. By definition, once $p(i+1)$ is obtained, it becomes the new front-line value, thus allowing us to discard $p(i)$ from memory.

6.5.1 Cost-*t* Testing

Let us consider an easier variant of the cost-t splitters problem called *cost-t testing*, which is identical to the former problem except that it does not require an algorithm to output a partition in any case (*i.e.,* even if P exists). An algorithm outputs only a Boolean answer: yes (that is, P exists), or no.

Clearly, the cost-t testing problem can also be solved in $O(N/B)$ I/Os. For this purpose, we slightly modify our algorithm for cost-t splitter: (i) eliminate the entire part of the algorithm dealing with the output file (which is unnecessary for cost-t testing), and (ii) if

the algorithm declares the absence of a feasible P, we return no for cost-t testing; otherwise, *i.e.,* the algorithm terminates without such a declaration, we return yes.

What do we gain from such a modification, compared to using the cost-t splitter algorithm to perform cost-t testing directly? The answer is the avoidance of writing $O(k/B)$ blocks. Recall that the cost-t splitter algorithm would produce during its execution an output file whose length can reach k. Doing away with the output file turns out to be crucial in attacking a concurrent extension of cost-t testing, as discussed next, which is the key to proving Theorem 7.

6.5.2 Concurrent Cost-t Testing

The goal of this problem is to solve multiple instances of the cost-t testing problem simultaneously. Specifically, given h integers satisfying $1 \leq t_1 < t_2 < \ldots < t_h \leq N$, the *concurrent testing problem* settles h instances of cost-t testing for $t = t_1, ..., t_h$, respectively. Following the result in Lemma 17, cost-t testing obeys the monotonicity that if cost-t testing returns yes (or no), then cost-t' with any $t' > t$ (or $t' < t$, respectively) will also return yes (no). Therefore, the output of concurrent testing can be a single value τ, equal to the smallest t_j ($1 \leq j \leq h$) such that t_j-testing returns yes. Note that τ does not need to always exist: the algorithm returns nothing if t_h-testing returns negatively (in which case, the t_j-testings of all $j \in [1, h-1]$ must also return no).

Assuming $h \leq cM$ where $0 < c < 1$ is to be decided later, we can perform concurrent testing in $O(N/B)$ I/Os. We concurrently execute h cost-t testings, each of which sets t to a distinct t_j, $1 \leq j \leq h$. The concurrency is made possible by several observations on the cost-t testing algorithm we developed earlier:

1. Regardless of t, the algorithm scans I and A only *forwardly*, *i.e.,* it never reads any block that has already been passed.

2. The next block to be read from I (A) is uniquely determined by its front-line value $p(i)$. In particular, if $p(i)$ is larger, then the block lies further down in the array I (A).

3. As one execution of the algorithm requires only $c' = O(1)$ words of memory, by setting

$$c = \frac{1}{c'}\left(1 - \frac{2B}{M}\right) \tag{6.7}$$

we ensure that h concurrent threads of the algorithm demand at most $cM \cdot c' = M - 2B$ words of memory. This will always leave us with two available memory blocks, which

we deploy as the input buffers for reading I and A, respectively. Note that since $M \geq 3B$, we have $c \geq 1/(3c')$, indicating that $cM = \Omega(M)$.

In memory, the algorithm uses a min-heap H to manage the front-line values of the h threads of cost-t testing. At each step, it de-heaps the smallest value p from H. Suppose without loss of generality that p comes from the thread of cost-t_j testing, for some $j \in [1, h]$. We execute this thread until having obtained its new front-line value, which is then en-heaped in H. This continues until all threads have terminated, at which point we determine the output τ as explained before. A trivial improvement is to stop testing t_j's for $t_j > t_i$ if the testing on t_i returns that t_i is feasible. The fact that it performs only $O(N/B)$ I/Os follows directly from the preceding observations about each thread of cost-t testing.

6.5.3 Solving the Static Interval Splitters Problem

Our I/O efficient algorithm for the static interval splitters problem has three steps:

1. Construct the stabbing-count array A.
2. Obtain the minimum t^* such that cost-t^* testing returns yes.
3. Solve the cost-t^* splitters problem to retrieve the splitters of an optimal partition P^*.

We refer to this algorithm as the *concurrent t-jump* method, or in short, *ct-jump*. The correctness of the algorithm is obvious, noticing that Step 2 guarantees t^* to be the cost of an optimal partition.

We explained previously how to do Step 1 in $O(SORT(N))$ I/Os and Step 3 in $O(N/B)$ I/Os. Next, we will show that Step 2 requires only $O((N/B)\log_M N)$ I/Os. This will establish Theorem 7 because, for $N \geq M$, it holds that[1]

$$\frac{\log N}{\log M} \leq \frac{\log N - \log B}{\log M - \log B}$$

Note that the left-hand side is $\log_M N$ whereas the right-hand side is $\log_{M/B}(N/B)$.

The rest of the section will concentrate on Step 2. It is easy to see that t^* falls in the range $[1, N]$. We will gradually shrink this *permissible range* until eventually it contains only a single value, *i.e.,* t^*. We achieve the purpose by launching multiple rounds of concurrent testing such that, after each round, the permissible range will be shrunk to $O(1/M)$ of the original length. An example for performing concurrent testings to shrink the permissible range is shown in Figure 6.5.

[1]Let x, y, z be positive values such that $x \geq y > z$, then $\frac{x}{y} \leq \frac{x-z}{y-z}$.

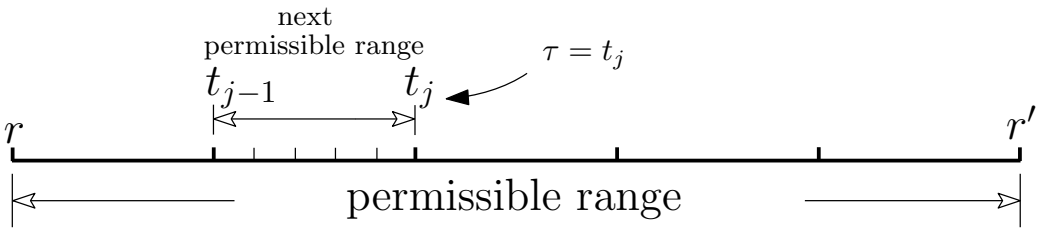


Figure 6.5: Concurrent testing on permissible ranges

Specifically, suppose that the permissible range is currently $[r, r']$ such that $r' - r \geq cM$, where c is the constant given in (6.7). We choose $h = cM$ integers $t_1, ..., t_h$ to divide $[r, r']$ as evenly as possible, namely: $t_j = r + \lceil j(r' - r)/(h + 1) \rceil$, for $j = 1, ..., h$. Then, we carry out concurrent testing with $t_1, ..., t_h$. Let τ be the output of the concurrent testing. Then, the permissible range can be shortened to:

- $[t_h + 1, r']$ if τ does not exist (*i.e.*,, the concurrent testing returned nothing).
- $[r, t_1]$ if $\tau = t_1$.
- $[t_{j-1} + 1, t_j]$ if $\tau = t_j$ for some $j > 1$.

It is easy to verify that the length of the new permissible range is at most $2/(cM) = O(1/M)$ that of the old one.

Finally, when the permissible range $[r, r']$ has length at most cM, we acquire the final t^* by one more concurrent testing with $h = r' - r + 1$ values $t_1, ..., t_h$, each of which is set to a distinct integer in $[r, r']$. The value of t^* equals the output τ of this concurrent testing.

It is clear that we perform $O(\log_M N)$ rounds of concurrent testing in total. As each round takes linear I/Os, we thus have obtained an algorithm that implements Step 2 in $O((N/B) \log_M N)$ I/Os. In other words, the algorithm *concurrent t-jump* has IO cost $O(SORT(N))$. This completes the proof of Theorem 7.

6.6 Queryable Interval Splitters and Updates

In this section, we tackle the challenges in solving the queryable interval splitters problem (*i.e.,* Problem 2 introduced in Section 6.2).

6.6.1 Queryable Interval Splitters

Our solutions for the static interval splitters problem also lead to efficient solutions to the queryable interval splitters problem.

In internal memory, we can use the t-jump algorithm for answering any queries with different k values. In a preprocessing step, we build the stabbing-count array A (which

occupies $O(N)$ space) in $O(N \log N)$ time. For subsequent queries, each query with a different k value takes $O(k \log N)$ time to answer, by solving $\log N$ cost-t splitters problems and each taking $O(k)$ time.

In external memory, the preprocessing cost of the *t-jump method* is $O(SORT(N))$ I/Os to build and maintain the disk-based stabbing-count array A. The size of the index (which is just A) is $O(N/B)$, and the query cost is $O(\min\{k, N/B\} \log N)$ I/Os.

The preprocessing step of the *concurrent t-jump method* is the same. Hence, it also takes $O(SORT(N))$ I/Os, and its index also uses $O(N/B)$ space. Each query takes $O((N/B) \log_M N)$ I/Os.

6.6.2 Dealing with Updates

In the queryable problem, another interesting challenge is to handle dynamic updates in the interval set I. Unfortunately, in this case, update costs in both the *t-jump* and *concurrent t-jump* methods are expensive. The indexing structure in both methods is the stabbing-count array A. To handle arbitrary updates, in the worst case, all elements in A need to be updated. Hence, the update cost is $O(N)$ time in internal memory, and $O(N/B)$ I/Os in external memory. This is too expensive for large datasets.

This limitation motivates us to explore update-efficient indexing structures and query methods for the queryable interval splitters problem. We again leverage the idea of solving $O(\log N)$ instances of the cost-t splitters problems (the decision version of our problem), in order to answer an optimal splitter query with any k value. The challenge boils down to designing an update-friendly indexing structure for answering a cost-t splitters query efficiently.

Observe that the key step in our algorithm for solving a cost-t splitters query in Section 6.4 is to figure out which starting value to use for placing the next splitter, which is given by Lines 8 and 12 in Algorithm 8. The critical part is to find out:

1) the number of intervals in I that strongly cover a starting value s, which is where a splitter has been placed, *i.e.,* $|I^{\times}(s)|$.
2) the number of intervals in I that share the same starting values as an interval $[s_i, e_i]$ $(1 \leq i \leq N)$, but with smaller ids less than i, *i.e.,* $|I^{o}_{<}(s_i)|$.

Hence, to solve a cost-t splitters problem instance, we just need to use an update-friendly index that answers any stabbing-count query efficiently, *i.e.,* an index that finds $|I^{\times}(s)|$ and $|I^{o}_{<}(s)|$ for any point s efficiently. This can be done efficiently using a *segment B-tree* [116]. In internal memory, this structure occupies $O(N)$ space, can be built in $O(N \log N)$ time,

answers a stabbing-count query in $O(\log N)$ time, and supports an insertion/deletion in $O(\log N)$ time. In external memory, the space, construction, query, and update costs are $O(N/B)$, $O((N/B)\log_{M/B}(N/B))$, $O(\log_B N)$, and $O(\log_B N)$, respectively.

Finally, answering a cost-t splitters query requires answering k different stabbing-count queries; and answering a queryable interval splitters problem then takes $O(\log N)$ cost-t splitters problem instances. Hence, the overall query cost of this approach is $O(k\log^2 N)$ time in internal memory, and $O(k\log_B N \cdot \log N)$ I/Os in external memory. We denote this method as the *stabbing-count-tree* method, or just *sc-tree*.

6.7 Experiments

We implemented all methods in C++. The external memory methods were implemented using the TPIE-library [21]. All experiments were performed on a Linux machine with an Intel Core i7-2600 3.4GHz CPU, a 4GB memory, and a 1TB hard drive.

6.7.1 Experiment Setups

We used several large real datasets. The first one *Temp* is from the LinkedSensorData project (originated from the MesoWest project). It contains temperature measurements from Jan 1997 to Oct 2011 from 26,383 distinct stations across the United States. There are almost 2.6 billion total readings from all stations with an average of 98,425 readings per station. For our experiments, we view each year of readings from a distinct station as *a distinct object*. Each new reading in an object is viewed as an update and creates *a new version of that object*. This leads to a multiversion database and every version of an object defines an interval in the database. *Temp* has 145,628 objects with an average of 17,833 versions per object. So *Temp* has approximately 2.6 billion intervals in total.

The second real dataset, *Meme*, was obtained from the Memetracker project. It tracks popular quotes and phrases which appear from various sources on the Internet. Each record has the URL of the website containing the memes, the time Memetracker observed the memes, and a list of the observed memes. We view each website as an object. Each record reports a new list of memes from a website, and it is treated as an update to the object representing that website, which creates a new version that is alive until the next update (a record containing the same website). *Meme* has almost 1.5 million distinct websites and an average of 67 records per website. Each version of an object in *Meme* also defines an interval, and we have approximately 100 million intervals in total.

We have also obtained the popular time series datasets [65] from the UCR Time Series website. In particular, we used three of the largest datasets from this collection, but they are

all much smaller than *Temp* and *Meme* as described above. All real datasets are summarized in Table 6.1, with their number of intervals.

We primarily used *Meme* in internal memory and *Temp* in external memory. In order to test the scalability, we randomly selected subsets from *Meme* and *Temp* to produce data with different number of intervals. Unless otherwise specified, the default values for important parameters in our experiments are summarized in Table 6.2. The page size is set to 4096 bytes by default. The default fill factor in the *sc-tree* is 0.7. In each experiment, we varied the value of one parameter of interest, while setting other parameters in their default values. Since the UCR time series datasets are all relatively small in size compared to *Meme* and *Temp*, we only used them for evaluating the internal memory methods.

6.7.2 Results from Internal Memory Methods

In this case, we focus on the results from the *static interval splitters* problem.

Figure 6.6 studies the effect of k and N. Clearly, the *DP* method is linear to k as shown in Figure 6.6(a) and quadratic to N, as shown in Figure 6.6(b). DP is 4-5 orders of magnitude more expensive than our t-jump method. On the other hand, even though the second step in t-jump is to solve $O(\log N)$ instances of cost-t splitters problem and each instance takes $O(k)$ time, the dominant cost for t-jump is the sorting operation in its step, which is to construct the stabbing-count array. Hence, its overall cost is $O(N \log N)$. As a result, its running time is almost not affected by k, as seen in 6.6(a), but (roughly) linearly affected by N as seen in 6.6(b). In conclusion, t-jump is extremely efficient for finding optimal splitters, and it is highly scalable (as cheap as main memory sorting methods). It takes only about 1 second to find optimal splitters for size-40 partitions over 2.5 million intervals, when they are not sorted. Note that t-jump will be even more efficient and scalable if data are already

Table 6.1: Number of intervals in real datasets tested

Temp	**Meme**	**CMU**	**Mallat**	**NHL**
2.6×10^9	1.0×10^8	1.57×10^5	2.39×10^6	1.41×10^6

Table 6.2: Default datasets and default values of key parameters

	Internal	**External**
Dataset	a subset of *Meme*	a subset of *Temp*
Size	$\sim$ 21 MB	$\sim$ 4.1 GB
N	$\sim$ 1 million	$\sim$ 200 million
k	40	5000
h	not applicable	5

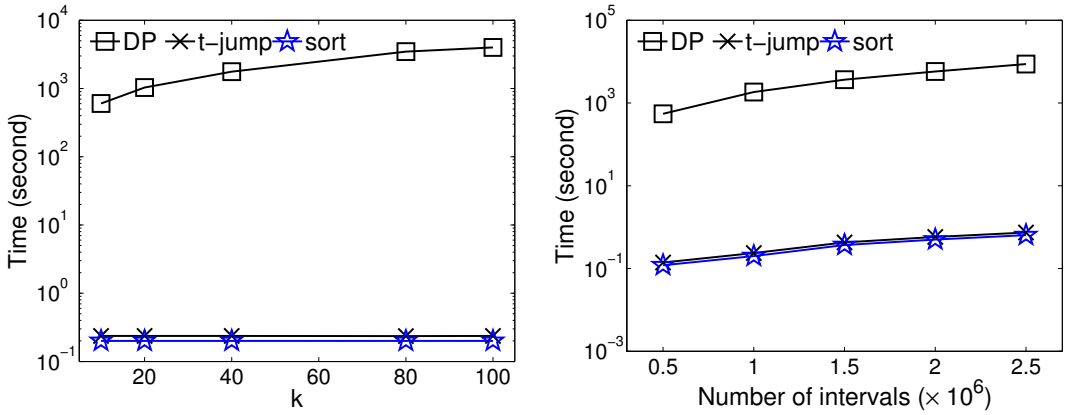


Figure 6.6: Running time of internal memory methods: (a) Vary k and (b) Vary N

sorted; in that case, its cost reduces to only $O(k \log N)$.

In Figure 6.7, we report the experimental results on the datasets from the UCR time series collection. The trend is similar to our observations from the *Meme* dataset. Our best solution t-jump consistently performs 3-4 orders of magnitudes faster than the DP approach in all three datasets. The dominant cost for t-jump is from sorting the input data, which is clearly shown in Figure 6.7.

6.7.3 Results from External Memory Methods

We first present the results for the *static interval splitters* problem, then analyze the results for the *queryable interval splitters* problem. In both problems, we need to study the effect of h from the second step in our concurrent t-jump method. Hence, we first evaluate the impact of h, as shown in Figure 6.8.

To isolate the impact of h, we show only running time from the second step of the *ct*-jump method, *i.e.,* we assume that the stabbing-count array has already been constructed. Therefore, we do not include its construction cost in Figure 6.8. We vary h between 1 to 10, and repeat the same experiment for $k = 2,000$, $k = 5,000$, and $k = 10,000$. What is interesting to observe from Figure 6.8(a) is that the running time initially deceases sharply and then slightly increases (very slowly), when we increase h. The same trend (albeit being less obvious and consistent) can also be observed for the number of IOs in Figure 6.8(b). This is because that the initial increment in h helps quickly reduce the permissible range, but subsequent increases in h lead to little gain (in shrinking the permissible range), but require more unnecessary testings. These results show that a small h value is sufficient for *ct*-jump to produce consistently good performance for a wide-range of k values. Hence, we set $h = 5$ as the default value for the rest of the experiments.

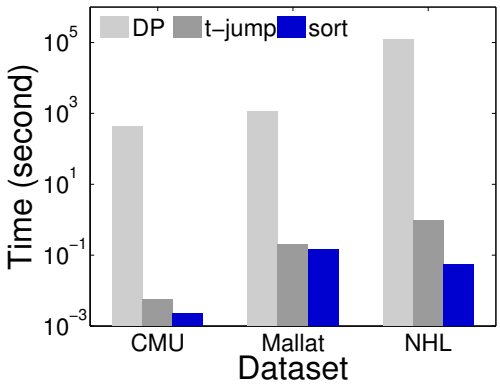


Figure 6.7: Results from the UCR datasets.

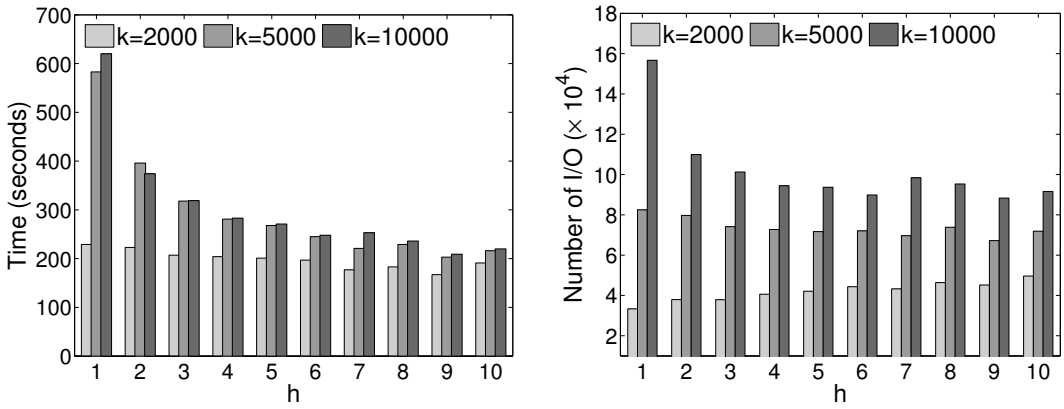


Figure 6.8: Effect of h in the second step of *ct*-jump: (a) Time and (b) IO

Next, we show the results for finding the static interval splitters. Recall that in this case, the overall cost includes the cost for building either the stabbing-count array or the stabbing-count tree, and the cost for finding the optimal splitters with the help of such a data structure. Also recall that the cost in this case, in terms of IOs, for *ct*-jump, *t*-jump, and *sc*-tree is $O(SORT(N))$, $O(SORT(N) + k \log N)$, and $O(SORT(N) + k \log_B N \log N)$, respectively.

Figure 6.9 studies the scalability of different methods by varying N from 50 million to 400 million. Not surprisingly, all methods have an almost linear dependence to N (*i.e.,* the number of intervals) in terms of both running time and IOs. But obviously, *ct*-jump achieves the best overall running time in Figure 6.9(a), and both *ct*-jump and *t*-jump have clearly outperformed the sc-tree method. The *ct*-jump method also has better IOs than *t*-jump, but the difference is not clearly visible in Figure 6.9(b), because the dominant IO cost for both methods is the external sort. Both methods have fewer IOs than sc-tree, especially

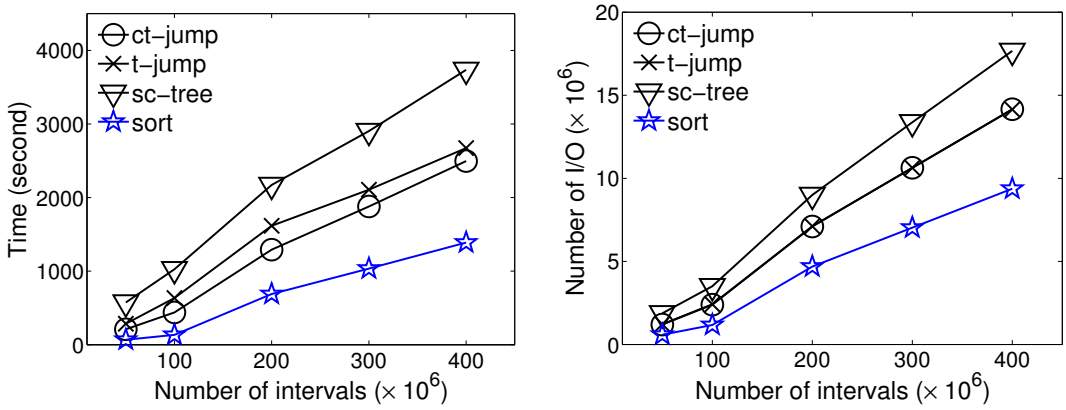


Figure 6.9: Static splitters, vary N: (a) Running time and (b) Total IO

for larger data. Overall, *ct*-jump is the best method that is almost as efficient and scalable as external sorting. With $N = 400$ million intervals, *ct*-jump achieves a 30% speedup over the *sc-tree* method and a 10% speedup over the *t*-jump method. Therefore, *ct*-jump is the most scalable method in this experiment.

We also investigated the impact of k by varying k from $2,000$ to $10,000$. As k becomes larger, both the running time and the number of IOs increase in all methods, especially for the sc-tree method, as shown in Figure 6.10. A larger k value increases the cost of solving a cost-t splitters problem, which is the essential step for all methods. However, the benefit of concurrent testing in *ct*-jump is clearly reflected in Figure 6.10(a) for larger k values. Its cost is still very much just the sorting cost, which is consistent with our theoretical analysis. In contrast, the *t*-jump method displays a clear increase in running time over larger k values. The *ct*-jump method is more than 2 times faster than sc-tree, and about 20-30% faster than the *t*-jump method, as we increase k.

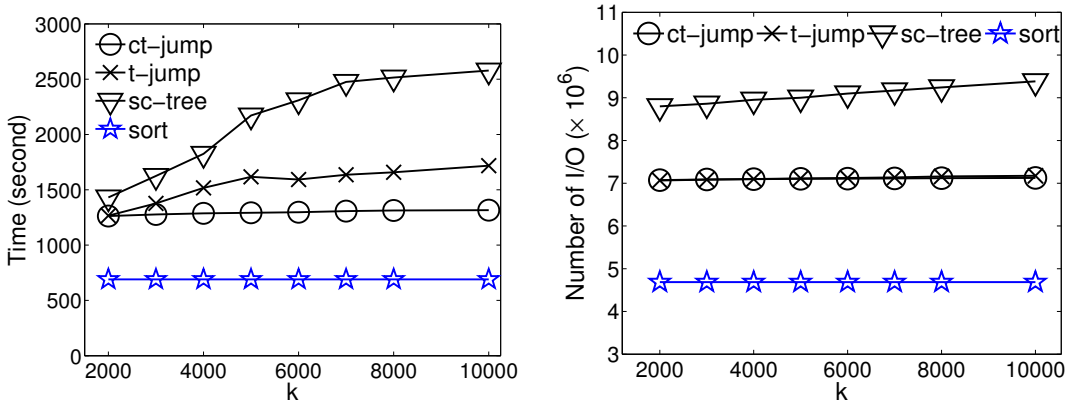


Figure 6.10: Static splitters, vary k: (a) Running time and (b) Total IO

In conclusion, *ct*-jump is the best method for the static interval splitters problem in external memory.

In what follows, we investigate the results for finding the queryable interval splitters. We first study the preprocessing cost to construct the indices for different external memory methods, which is to construct either the stabbing-count array in *ct*-jump and *t*-jump, or the stabbing-count tree in *sc*-tree. Figure 6.11 compares the size of the indices in different methods when we increase N from 50 million to 400 million. Both stabbing-count array and stabbing-count tree are linear to N, however, a stabbing-count tree does require much more space, almost by a factor 2 when N becomes 400 million. The stabbing-count array in both *ct*-jump and *t*-jump is the same in size as the size of all intervals. This means that it will be much smaller than the size of the dataset, which contains other values than just an interval for each record.

In terms of the construction cost of indices in these methods, Figure 6.12 shows that building a stabbing-count array is slightly cheaper than building a stabbing-count tree. But both are dominated by the external sorting cost, and obviously the *ct*-jump and the *t*-jump methods share the same preprocessing cost. All three methods have an almost linear construction cost to N.

We next study the query cost. Figure 6.13 investigates the impact of k when it changes from $2,000$ to $10,000$. The query time for all three methods increase, as shown in Figure 6.13(a), but the cost of *ct*-jump increases at the slowest rate. Specifically, we observe that *ct*-jump answers a query 3-4 times faster than *sc*-tree, and about 1-2 times faster than *t*-jump. A similar trend can be observed in the IO cost, as shown in Figure 6.13(b). However, in this case, the IO difference between *ct*-jump and *t*-jump appears to be much smaller, compared to their difference in query time.

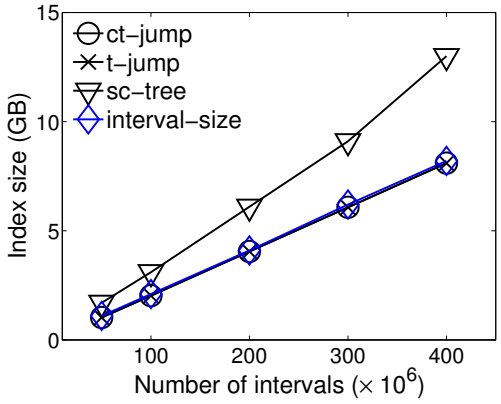


Figure 6.11: Index size

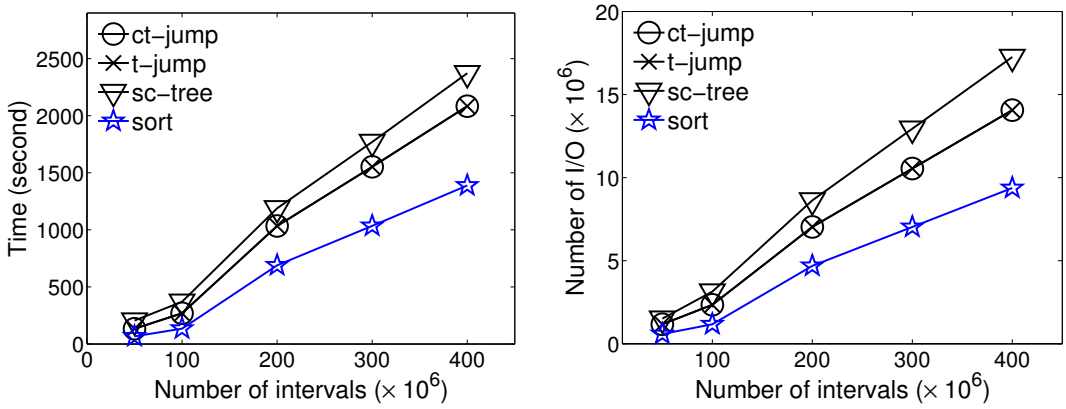


Figure 6.12: Preprocessing cost: (a) Preprocessing time and (b) Preprocessing IO

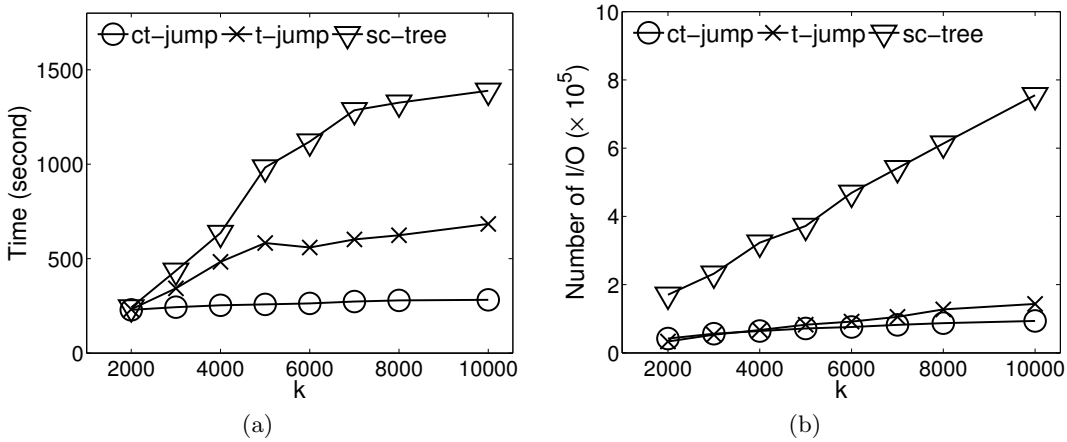
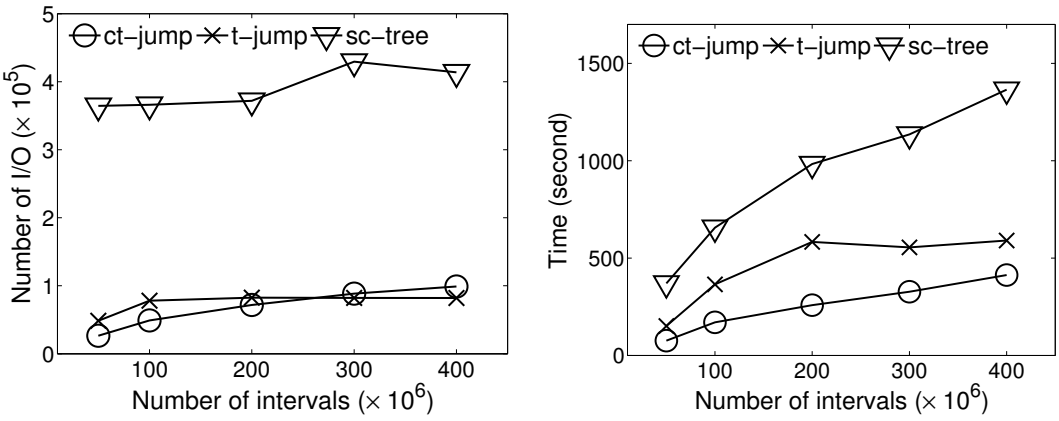


Figure 6.13: Queryable splitters, vary k: (a) Update time and (b) Update IO

So one may wonder why the query time of *ct*-jump is much better than *t*-jump, when the difference in terms of the number of IOs is not so significant. This is explained by the (much) better caching behavior in *ct*-jump. Recall that *ct*-jump performs concurrent testing of several cost-t testing instances. As a result, it makes very small leaps while scanning through the stabbing-count array, compared to the big leaps made by *t*-jump over the stabbing-count array. These small leaps result in much better hit rates in buffer and cache, leading to much better running time.

Next, we examine the scalability of our query methods when we increase the size of the data from 50 million to 400 million. Figure 6.14(a) reports the query cost in terms of number of IOs. Not surprisingly, larger N values only increase the query IOs by a small amount. In contrast to the linear dependence on k, the query cost of *ct*-jump and *t*-jump only depends on N by a $O(\log N)$ factor, and the query cost of *sc*-tree only depends on N by a $O(\log_B N \log N)$ factor. On the other hand, the increases in running time for all



(a) (b)

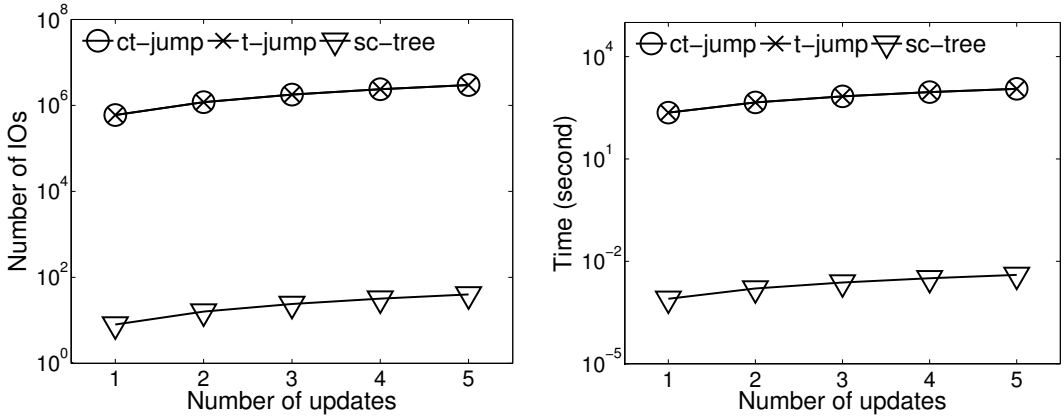
Figure 6.14: Queryable splitters, vary N: (a) Query IO and (b) Query time

three methods are more significant when N increases, as shown in Figure 6.14(b). This is because as N becomes larger, for a fixed k value, the distance between two consecutive splitters also becomes larger. Since all three methods essentially make k jumps over the data to solve a cost-t splitters problem instance, a larger jump in distance leads to poorer caching performance, which explains the more notable increase in query time than that in query IO when N increases. Nevertheless, *ct*-jump outperforms *sc*-tree by about 60% and *t*-jump by about 30% in time on average.

Lastly, updates to the data may happen for the queryable splitters problem. When dealing with updates is important, as we have analyzed earlier in Section 6.6, *sc*-tree becomes a much more attractive method than *ct*-jump and *t*-jump. The fundamental reason is that a stabbing-count tree is a dynamic indexing structure with an update cost of $O(\log_B N)$ IOs, while a stabbing-count array is not with an update cost of $O(N/B)$ IOs. Of course, the update-friendly property in a stabbing-count tree comes with the price of having more expensive query cost, as we have already shown.

That said, we generate a number of updates by randomly issuing a few insertions/deletions on the initial set of intervals. We report the average cost in IOs and time from 10 updates, as shown in Figure 6.15(a) and Figure 6.15(b). Clearly, the *sc-tree* method is much more efficient in handling updates than *t-jump* and *ct-jump*.

In summary, for the queryable splitter problem in external memory, when data are static (which is often the case for big data), the *ct-jump* method is the most efficient method. When dealing with dynamic updates is important, *sc-tree* works the best. Nevertheless, all three methods have excellent efficiency and scalability on big data, and the key idea behind all three methods is our observation on solving $O(\log N)$ cost-t problem (or testing) instances.



(a) (b)

Figure 6.15: The update cost for queryable splitters: (a) Update IO and (b) Update time

6.7.4 Optimal Point Splitters

A special case of our problem is when *all intervals degenerate to just points*, where *each interval* starts and ends at one same time instance.

Our solution can gracefully handle such special case, *i.e.,* to find the optimal splitters for a point set. The state-of-the-art method for finding the optimal splitters in a point set runs in $O(k \log^2 N)$ time [95], assuming the set of input points has already been sorted. We dub this approach the p-split method. The details of this study will be surveyed in Section 6.8. In contrast, t-jump runs in only $O(k \log N)$ time, saving a $O(\log N)$ factor compared to the state-of-the-art on a point dataset. Note that the investigation from [95] focuses only on the RAM model, *i.e., p*-split is an internal memory method. Therefore, in this experiment, we compare p-split against our internal memory method t-jump by using all the internal memory datasets. In particular, we represent each interval only by its starting value and report the running time for finding the optimal point splitters. The results are shown in Figure 6.16. Not surprisingly, t-jump performs 3-4 times faster than the state-of-the-art method p-split across all datasets.

6.7.5 Final Remark

It is also interesting to point out our algorithms, t-jump, ct-jump, and sc-tree also produce very balanced buckets while minimizing the size of the maximum bucket. This is due to the fact that they all use our algorithm for solving the cost-t splitters (or cost-t testing) problem as a basic building block. And the way we solve the cost-t splitters (or cost-t testing) problem is to attempt to produce each bucket with a size t. When this is not possible, our algorithm finds a bucket with a size that is as close as possible to t, before producing the next bucket. In other words, in the optimal partition P^* that our algorithms

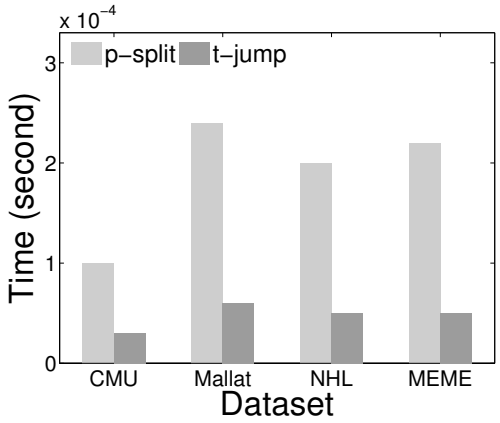


Figure 6.16: Comparison with p-split method in [95] to find optimal point splitters

find with the maximum bucket size being equal to t^*, the size of each bucket in P^* is in fact very close to t^*.

Figure 6.17 verifies this claim over both real datasets, when we vary k and show the average and the standard deviation for the sizes of all buckets in the optimal partition P^* produced by our algorithms, along with $t^* = c(P^*)$, the size of the maximum bucket in P^*. In all cases, the average bucket size is very close to t^*, and the standard deviation is very small (hundred to a thousand, compared to tens of or hundred thousand for the average bucket size). Furthermore, note that the average size of a bucket is not necessarily $\frac{N}{k+1}$ in our problem, since an interval may span over multiple buckets. In fact, it is always $\geq \frac{N}{k+1}$ and varies for different partitions depending on the dataset I, the budget k, and the partitioning algorithm used. Nevertheless, all three of our methods, t-jump, ct-jump, and sc-tree, produce the same P^* for a given k and I.

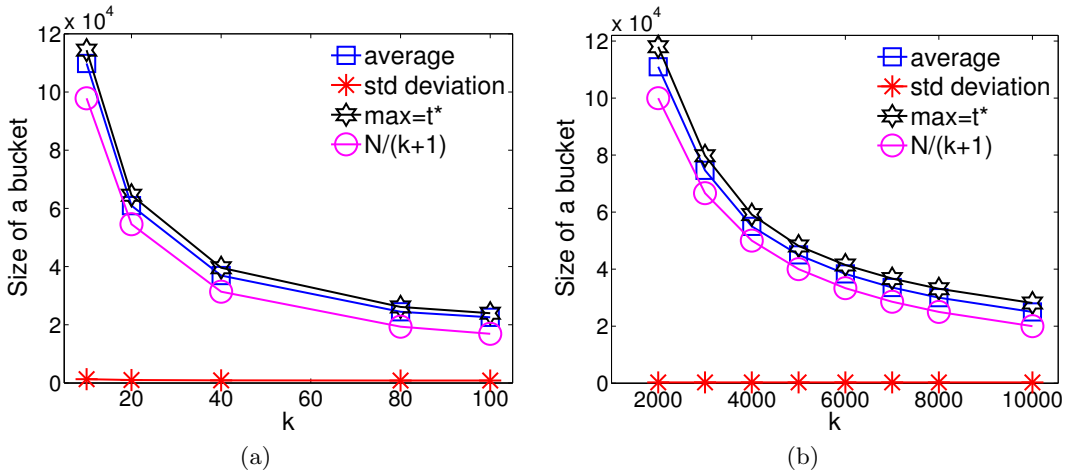


Figure 6.17: Balanced partitions produced by our algorithms on (a) Meme Data and (b) Temp data

6.8 Related Work

To the best of our knowledge, this is the first work that investigates the problem of finding optimal splitters for large interval data.

Ross and Cieslewicz studied the problem of finding optimal splitters for a set of one-dimensional points [95]. In their problem, a partition consists a set of disjoint buckets over the point dataset. The buckets are produced by k splitters. However, a set of k splitters will produce $(2k+1)$ buckets instead of just $(k+1)$ buckets: they count all points that have values equal to a splitter as a separate bucket, so k splitters form k distinct buckets by themselves (which we refer to as the *splitter buckets*), plus the other $(k+1)$ buckets formed by any two neighboring splitters (which we refer to as the *nonsplitter buckets*). Their goal, however, is to minimize the size of the maximum bucket from only the $(k+1)$ nonsplitter buckets. Their motivation was to disregard buckets that contain only a single (but many duplicates) value. This finds applications in incremental sorting, distributed and parallel processing (of some applications), etc., as they argued in [95]. They have proposed a $O(k \log^2 N)$ method for memory-resident point sets. Clearly, their problem is fundamentally different from the interval splitters problem. Interestingly, it is possible to extend our stabbing-count array based methods, namely, *t-jump* and *concurrent t-jump*, to solve this problem in internal and external memory, respectively. In that case, our solution leads to a $O(k \log N)$ method, which improves the results in [95] by a $O(\log N)$ factor for the point splitters problem.

On the other hand, the array partitioning problem is as follows. The input is an one-dimension array E of N non-negative numbers, and a *weight function* w that maps a contiguous segment of A to a non-negative integer. The k-partition of E is a division of E into $(k+1)$ contiguous segments, that is, setting dividers $\ell_0 = 0 < \ell_1 < \cdots < \ell_k < \ell_{k+1} = N$. Here, the ith segment is $E[\ell_{i-1} + 1 \cdots \ell_i]$. The MAX norm of a partition over E is $\max_{i=1}^{k+1} w(E[\ell_{i-1} + 1 \cdots \ell_i]$. The goal is to find a partition of size k that minimizes the MAX norm of any size-k partitions over E. Typical weight function includes addition and Hamming weight function [67, 79] among others. This problem can also be extended to 2-dimensional arrays, where a partition consists of a number of disjoint but complete-covering 2-dimensional blocks over the 2-dimensional array E. Khanna *et al.* studied this problem and gave an $O(N \log N)$ algorithm for memory resident arrays in 1d for arbitrary k. This problem is NP-hard in 2d and they gave approximation methods instead. More efficient and effective approximations were then given in [79]. This problem is related but certainly very different from our work. An interesting open problem is the interval array partitioning problem, which is defined similarly as the array partitioning problem, except that each

element in the array is an interval of values (*e.g.,* those from an uncertain object).

Our study may find interesting applications in parallel interval scheduling problems [47], where each job has a specified interval within which it needs to be executed. Each machine has a *busy interval* that contains all the intervals corresponding to the jobs it processes. Given the parallelism parameter $g \geq 1$, which is the maximal number of jobs that can be processed simultaneously by a single machine. The goal is to assign the jobs to machines such that the total busy time of the machines is minimized. This problem is known to be NP-hard for $g \geq 2$. Nevertheless, it is an intriguing future work to explore if our techniques can help design efficient approximate solutions for such problems.

Lastly, the DP method follows the general intuition of bucketization that finds application in optimal histogram constructions, *e.g.,* the DP method for constructing a V-optimal histogram [58, 89]. The construction of the stabbing count array is somewhat related to the prefix sum array that finds applications in data warehouses and histogram constructions, *e.g.,* [54, 69]. Our study is also generally related with the management of interval and temporal data [15, 22, 37, 70, 70, 106].

6.9 Conclusion

Temporal and multiversion RDF databases often generate massive amounts of data. Therefore, it becomes increasingly important to store and process these data in a distributed and parallel fashion. This work makes an important contribution in solving the optimal splitters problem, which is essential in enabling efficient distributed and parallel processing of such data. An interesting open problem is to extend our study to higher dimensions.

CHAPTER 7

OTHER WORKS

A theme that threads up this dissertation is efficient query processing and optimization for RDF data. Apart from this theme, we have also conducted research on other emerging problems, which include (I) ranking large temporal data; (II) security issues in data outsourcing. In what follows, we summarize the main ideas of these works.

We have studied the problem of ranking queries on large temporal data [75]. Temporal data arise in a large number of domains, such as time series data, transactional databases, spatio-temporal trajectories, and many others. The database community has devoted extensive amount of efforts to indexing and querying temporal data in the past decades. However, insufficient amount of attention has been paid to temporal ranking queries, arguably one of the most important query types. More precisely, given any time instance t, the query asks for the top-k objects at time t with respect to some score attribute. Some generic indexing structures based on R-trees do support ranking queries on temporal data, but as they are not tailored for such queries, the performance is far from satisfactory. To this end, we present the Seb-tree, a simple indexing scheme that supports temporal ranking queries much more efficiently than the R-tree-based generic methods in both theory and practice. The Seb-tree uses the B-tree as the only building block and hence is especially appealing in practice due to its simplicity.

We have also conducted research to investigate how to audit a database server's effort in evaluating clients' queries – an emerging problem arise under the database-as-a-service model [72]. We show that extending the classic techniques in the literature to outsourced databases with multiple data owners is highly inefcient. To cope with lazy servers in the distributed setting, we propose query access assurance (QAA) that focuses on IO-bound queries. The goal in QAA is to enable clients to verify that the server has honestly accessed all records that are necessary to compute the correct query answer, thus eliminating the incentives for the server to be lazy if the query cost is dominated by the IO cost in accessing these records. We formalize this concept for distributed databases, and present two efcient

schemes that achieve QAA with high success probabilities. Our design employs a few number theory techniques and successfully address the limitation in the classic techniques.

CHAPTER 8

CONCLUSION

In this dissertation, we have studied several emerging topics with regards to the management of linked data and its query optimization techniques, in an effort to support scalable data analytics and integration for large linked data.

A common observation during the dissertation research is that techniques from relational, XML, and graph databases oftentimes fall short in handling large, real RDF data sets and SPARQL queries. This is in part due to the facts that large RDF data are schema-less, distributed, and constantly evolving, which has put forward new challenges in building scalable RDF data management systems and necessitated significant research effort. To this end, we have revisited the classic problem of query rewriting over views in the context of SPARQL and RDF. We proposed the first sound and complete query rewriting algorithm for SPARQL, with novel optimizations. A follow-up problem we have addressed in this dissertation is how to perform multiquery optimization for SPARQL in a principled and declarative manner. Our study opens the gate to several interesting future researches, such as how to efficiently deal with variable predicates in query and view definition, how to partially materialize the view with the query rewriting in SPARQL to further improve the efficiency, and also how to include other SPARQL features such as FILTER and OPTIONAL into the algorithm. Beyond the structured query language SPARQL, we have studied keyword search for RDF data, which is an indispensable tool in data integration for the schemaless data. After identifying the defects and limitations of existing methods, we studied how to leverage the ontology of RDF data to optimize the query evaluation, which led to promising experimental results in answering keyword queries on real RDF data sets. An interesting future work is to optimize general SPARQL queries with the ontology of the data, *e.g.,* pruning empty rewritten queries from the query rewriting.

A few techniques we have proposed in this dissertation research can be seamlessly generalized beyond RDF and SPARQL. For instance, we have studied efficient solutions to find the optimal splitters for temporal and multiversion RDF data. The techniques we proposed

can also be applied to a more general setting for distributed and parallel computation. Interesting future work includes taking heavy hitters into consideration and extending our techniques to higher dimensions.

REFERENCES

[1] 4store - scalable RDF storage. http://4store.org/.

[2] Currently Alive SPARQL Endpoints. http://www.w3.org/wiki/SparqlEndpoints.

[3] DBpeida. http://dbpedia.org.

[4] Jena semantic web framework. http://jena.sourceforge.net.

[5] The link open data. http://linkeddata.org/.

[6] Linked sensor data. http://wiki.knoesis.org/index.php/LinkedSensorData.

[7] The mesowest project. http://mesowest.utah.edu.

[8] Resource Description Framework. http://www.w3.org/RDF/.

[9] SPARQL query language for RDF. http://www.w3.org/TR/rdf-sparql-query/.

[10] The TPC Benchmarks. http://www.tpc.org/.

[11] Virtuoso universal server. http://virtuoso.openlinksw.com.

[12] Abadi, D. J., Marcus, A., Madden, S. R., and Hollenbach, K. Scalable semantic web data management using vertical partitioning. In *VLDB* (Vienna, Austria, 2007).

[13] Abel, F., Coi, J. L. D., Henze, N., Koesling, A. W., Krause, D., and Olmedilla, D. Enabling advanced and context dependent access control in RDF stores. In *ISWC* (Busan, Korea, 2007).

[14] Abiteboul, S., Hull, R., and Vianu, V. *Foundations of databases*. Addison-Wesley, 1995.

[15] Agarwal, P. K., Arge, L., and Yi, K. An optimal dynamic interval stabbing-max data structure. In *SODA* (Vancouver, Canada, 2005).

[16] Aggarwal, A., and Vitter, J. S. The input/output complexity of sorting and related problems. *Communications of the ACM 31*, 9 (1988), 1116–1127.

[17] Agrawal, S., Chaudhuri, S., and Das, G. DBXplorer: enabling keyword search over relational databases. In *SIGMOD* (Madison, Wisconsin, 2002).

[18] Alon, N., Matias, Y., and Szegedy, M. The space complexity of approximating the frequency moments. In *STOC* (Philadelphia, USA, 1996).

[19] Anagnostopoulos, A., Vlachos, M., Hadjieleftheriou, M., Keogh, E., and Yu, P. S. Global distance-based segmentation of trajectories. In *KDD* (Philadelphia, USA, 2006).

[20] Angles, R., and Gutierrez, C. The expressive power of SPARQL. In *ISWC* (Karlsruhe, Germany, 2008).

[21] Arge, L., Procopiuc, O., and Vitter, J. S. Implementing I/O-efficient data structures using TPIE. In *ESA* (Rome, Italy, 2002).

[22] Arge, L., and Vitter, J. S. Optimal external memory interval management. *SIAM Journal on Computing 32*, 6 (2003), 1488–1508.

[23] Atre, M., Chaoji, V., Zaki, M. J., and Hendler, J. A. Matrix "bit" loaded: A scalable lightweight join query processor for RDF data. In *WWW* (Raleigh, USA, 2010).

[24] Becker, B., Gschwind, S., Ohler, T., Seeger, B., and Widmayer, P. An asymptotically optimal multiversion b-tree. *The VLDB Journal 5*, 4 (1996), 264–275.

[25] Beyer, K., Haas, P. J., Reinwald, B., Sismanis, Y., and Gemulla, R. On synopses for distinct-value estimation under multiset operations. In *SIGMOD* (Beijing, China, 2007).

[26] Bhalotia, G., Hulgeri, A., Nakhe, C., Chakrabarti, S., and Sudarshan, S. Keyword searching and browsing in databases using banks. In *ICDE* (San Jose, USA, 2002).

[27] Biggs, N., Lloyd, E., and Wilson, R. *Graph Theory*. Oxford University Press, 1986.

[28] Bizer, C., and Schultz, A. The Berlin SPARQL benchmark. *International Journal on Semantic Web and Information Systems 5*, 2 (2009), 1–24.

[29] Bornea, M. A., Dolby, J., Kementsietsidis, A., Srinivas, K., Dantressangle, P., Udrea, O., and Bhattacharjee, B. Building an efficient RDF store over a relational database. In *SIGMOD* (New York, USA, 2013).

[30] Broekstra, J., Kampman, A., and Harmelen, F. V. Sesame: A generic architecture for storing and querying RDF and RDF schema. In *ISWC* (Sardinia, Italy, 2002).

[31] Bruno, N., Gravano, L., Koudas, N., and Srivastava, D. Navigation- vs. index-based XML multiquery processing. In *ICDE* (Bangalore, India, 2003).

[32] Cautis, B., Deutsch, A., and Onose, N. XPath rewriting using multiple views: Achieving completeness and efficiency. In *WebDB* (Vancouver, Canada, 2008).

[33] Cautis, B., Deutsch, A., and Onose, N. Querying data sources that export infinite sets of views. In *ICDT* (St. Petersburg, Russia, 2009).

[34] Chen, Q., Chen, L., Lian, X., Liu, Y., and Yu, J. X. Indexable PLA for efficient similarity search. In *VLDB* (Vienna, Austria, 2007).

[35] Chen, Y., Wang, W., and Liu, Z. Keyword-based search and exploration on databases. In *ICDE* (Hannover, Germany, 2011).

[36] Chen, Y., Wang, W., Liu, Z., and Lin, X. Keyword search on structured and semistructured data. In *SIGMOD* (Providence, USA, 2009).

[37] Chomicki, J., Toman, D., and Böhlen, M. H. Querying ATSQL databases with temporal logic. *ACM Transactions on Database Systems 26*, 2 (2001), 145–178.

[38] Correndo, G., Salvadores, M., Millard, I., Glaser, H., and Shadbolt, N. SPARQL query rewriting for implementing data integration over linked data. In *EDBT* (Lausanne, Switzerland, 2010).

[39] Cyganiak, R., and Jentzsch, A. The LOD diagram. `http://lod-cloud.net`.

[40] Dalvi, B. B., Kshirsagar, M., and Sudarshan, S. Keyword search on external memory data graphs. In *VLDB* (Auckland, New Zealand, 2008).

[41] Dalvi, N. N., Sanghai, S. K., Roy, P., and Sudarshan, S. Pipelining in multiquery optimization. In *PODS* (Santa Barbara, USA, 2001).

[42] Diwan, A. A., Sudarshan, S., and Thomas, D. Scheduling and caching in multiquery optimization. In *COMAD* (Delhi, India, 2006).

[43] Duan, S., Kementsietsidis, A., Srinivas, K., and Udrea, O. Apples and oranges: a comparison of RDF benchmarks and real RDF datasets. In *SIGMOD* (Athens, Greece, 2011).

[44] Fan, W., Chan, C.-Y., and Garofalakis, M. Secure XML querying with security views. In *SIGMOD* (Paris, France, 2004).

[45] Fan, W., Geerts, F., Jia, X., and Kementsietsidis, A. Rewriting regular XPath queries on XML views. In *ICDE* (Istanbul, Turkey, 2007).

[46] Finkelstein, S. Common expression analysis in database applications. In *SIGMOD* (Orlando, USA, 1982).

[47] Flammini, M., Monaco, G., Moscardelli, L., Shachnai, H., Shalom, M., Tamir, T., and Zaks, S. Minimizing total busy time in parallel scheduling with application to optical networks. In *IPDPS* (Rome, Italy, 2009).

[48] Fu, H., and Anyanwu, K. Effectively interpreting keyword queries on RDF databases with a rear view. In *ISWC* (Bonn, Germany, 2011).

[49] Golenberg, K., Kimelfeld, B., and Sagiv, Y. Keyword proximity search in complex data graphs. In *SIGMOD* (Vancouver, Canada, 2008).

[50] Guo, Y., Pan, Z., and Heflin, J. LUBM: A benchmark for OWL knowledge base systems. *Journal of Web Semantics 3*, 2 (2005), 158–182.

[51] Gutierrez, C., Hurtado, C., and Vaisman, A. Temporal RDF. In *ESWC* (Paris, France, 2005).

[52] Halevy, A. Y. Answering queries using views: A survey. *The VLDB Journal 10*, 4 (2001), 270–294.

[53] He, H., Wang, H., Yang, J., and Yu, P. S. Blinks: ranked keyword searches on graphs. In *SIGMOD* (Beijing, China, 2007).

[54] Ho, C.-T., Agrawal, R., Megiddo, N., and Srikant, R. Range queries in OLAP data cubes. In *SIGMOD* (Tucson, USA, 1997).

[55] Hong, M., Demers, A. J., Gehrke, J., Koch, C., Riedewald, M., and White, W. M. Massively multiquery join processing in publish/subscribe systems. In *SIGMOD* (Beijing, China, 2007).

[56] Hristidis, V., Gravano, L., and Papakonstantinou, Y. Efficient IR-style keyword search over relational databases. In *VLDB* (Berlin, Germany, 2003).

[57] Ianni, G., Krennwallner, T., Martello, R., and Polleres, A. Dynamic querying of mass-storage RDF data with rule-based entailment regimes. In *ISWC* (Washington D.C., USA, 2009).

[58] Jagadish, H. V., Koudas, N., Muthukrishnan, S., Poosala, V., Sevcik, K. C., and Suel, T. Optimal histograms with quality guarantees. In *VLDB* (New York, USA, 1998).

[59] Jain, A. K., Murty, M. N., and Flynn, P. J. Data clustering: a review. *ACM Computing Survey 31*, 3 (1999), 264–323.

[60] Jeavons, P. On the algebraic structure of combinatorial problems. *Theoretical Computer Science 200*, 1 (1998), 185–204.

[61] Kacholia, V., Pandit, S., Chakrabarti, S., Sudarshan, S., Desai, R., and Karambelkar, H. Bidirectional expansion for keyword search on graph databases. In *VLDB* (Trondheim, Norway, 2005).

[62] Kalyanasundaram, B., and Schintger, G. The probabilistic communication complexity of set intersection. *SIAM Journal on Discrete Mathematics 5*, 4 (1992), 545–557.

[63] Kargar, M., and An, A. Keyword search in graphs: Finding r-cliques. In *VLDB* (Seattle, USA, 2011).

[64] Kementsietsidis, A., Neven, F., de Craen, D. V., and Vansummeren, S. Scalable multiquery optimization for exploratory queries over federated scientific databases. In *VLDB* (Auckland, New Zealand, 2008).

[65] Keogh, E., Xi, X., Wei, L., and Ratanamahatana., C. The UCR timeseries datasets. `http://www.cs.ucr.edu/~eamonn/time_series_data/`.

[66] Keogh, E. J., Chu, S., Hart, D., and Pazzani, M. J. An online algorithm for segmenting time series. In *ICDM* (San Jose, USA, 2001).

[67] Khanna, S., Muthukrishnan, S., and Skiena, S. Efficient array partitioning. In *ICALP* (Bologna, Italy, 1997).

[68] Koch, I. Enumerating all connected maximal common subgraphs in two graphs. *Theoretical Computer Science 250*, 1 (2001), 1–30.

[69] Koudas, N., Muthukrishnan, S., and Srivastava, D. Optimal histograms for hierarchical range queries. In *PODS* (Dallas, USA, 2000).

[70] Kriegel, H.-P., Pötke, M., and Seidl, T. Managing intervals efficiently in object-relational databases. In *VLDB* (Cairo, Egypt, 2000).

[71] Le, W., Duan, S., Kementsieditis, A., Li, F., and Wang, M. Rewriting queries on SPARQL views. In *WWW* (Hyderabad, India, 2011).

[72] Le, W., and Li, F. Query access assurance in outsourced databases. *IEEE Transactions on Services Computing 5*, 2 (2012), 178–191.

[73] Le, W., Li, F., Tao, Y., and Christensen, R. Optimal splitters for temporal and multiversion databases. In *SIGMOD* (New York, USA, 2013).

[74] Lenzerini, M. Data integration: A theoretical perspective. In *PODS* (Madison, USA, 2002).

[75] Li, F., Yi, K., and Le, W. Top-k queries on temporal data. *The VLDB Journal 19*, 5 (2010), 715–733.

[76] Li, G., Ooi, B. C., Feng, J., Wang, J., and Zhou, L. EASE: Efficient and adaptive keyword search on unstructured, semistructured and structured data. In *SIGMOD* (Vancouver, Canada, 2008).

[77] Lomet, D., Hong, M., Nehme, R., and Zhang, R. Transaction time indexing with version compression. In *VLDB* (Auckland, New Zealand, 2008).

[78] Luo, Y., Wang, W., and Lin, X. SPARK: A keyword search engine on relational databases. In *ICDE* (Cancun, Mexico, 2008).

[79] Muthukrishnan, S., and Suel, T. Approximation algorithms for array partitioning problems. *Algorithms 54*, 1 (2005), 85–104.

[80] Neumann, T., and Weikum, G. RDF-3X: a RISC-style engine for RDF. In *VLDB* (Auckland, New Zealand, 2008).

[81] Neumann, T., and Weikum, G. Scalable join processing on very large RDF graphs. In *SIGMOD* (Providence, USA, 2009).

[82] O'Gorman, K., Agrawal, D., and Abbadi, A. E. Multiple query optimization by cache-aware middleware using query teamwork. In *ICDE* (San Jose, USA, 2002).

[83] Östergård, P. R. A fast algorithm for the maximum clique problem. *Discrete Applied Mathematics 120*, 1 (2002), 195–205.

[84] Palpanas, T., Vlachos, M., Keogh, E., Gunopulos, D., and Truppel, W. Online amnesic approximation of streaming time series. In *ICDE* (Boston, USA, 2004).

[85] Papakonstantinou, Y., and Vassalos, V. Query rewriting for semistructured data. In *SIGMOD* (Philadelphia, USA, 1999).

[86] Park, J., and Segev, A. Using common subexpressions to optimize multiple queries. In *ICDE* (Los Angeles, USA, 1988).

[87] Pérez, J., Arenas, M., and Gutierrez, C. Semantics and complexity of SPARQL. *ACM Transactions on Database Systems 34*, 3 (2009), 1–45.

[88] Polleres, A. From SPARQL to rules (and back). In *WWW* (Banff, Canada, 2007).

[89] Poosala, V., Ioannidis, Y. E., Haas, P. J., and Shekita, E. J. Improved histograms for selectivity estimation of range predicates. In *SIGMOD* (Montreal, Canada, 1996).

[90] Pottinger, R., and Halevy, A. MiniCon: A scalable algorithm for answering queries using views. *The VLDB Journal 10*, 2 (2001), 182–198.

[91] Preda, N., Suchanek, F. M., Kasneci, G., Neumann, T., Yuan, W., and Weikum, G. Active knowledge: Dynamically enriching RDF knowledge bases by web services. In *SIGMOD* (Singapore, 2010).

[92] Quilitz, B., and Leser, U. Querying distributed RDF data sources with SPARQL. In *ESWC* (Tenerife, Spain, 2008).

[93] Raymond, J. W., and Willett, P. Maximum common subgraph isomorphism algorithms for the matching of chemical structures. *Journal of Computer-Aided Molecular Design 16*, 7 (2002), 521–533.

[94] Rizvi, S., Mendelzon, A., Sudarshan, S., and Roy, P. Extending query rewriting techniques for fine-grained access control. In *SIGMOD* (Paris, France, 2004).

[95] Ross, K. A., and Cieslewicz, J. Optimal splitters for database partitioning with size bounds. In *ICDT* (St. Petersburg, Russia, 2009).

[96] Roy, P., Seshadri, S., Sudarshan, S., and Bhobe, S. Efficient and extensible algorithms for multiquery optimization. In *SIGMOD* (Dallas, USA, 2000).

[97] Schmidt, M., Hornung, T., Lausen, G., and Pinkel, C. SP^{2}Bench: A SPARQL performance benchmark. In *ICDE* (Shanghai, China, 2009).

[98] Schmidt, M., Meier, M., and Lausen, G. Foundations of SPARQL query optimization. In *ICDT* (Lausanne, Switzerland, 2010).

[99] Sellis, T., and Ghosh, S. On the multiple-query optimization problem. *IEEE Transactions on Knowledge and Data Engineering 2*, 2 (1990), 262–266.

[100] Sellis, T. K. Multiple-query optimization. *ACM Transactions on Database Systems 13*, 1 (1988), 23–52.

[101] Shamir, R., and Tsur, D. Faster subtree isomorphism. *Journal of Algorithms 33*, 2 (1999), 267–280.

[102] Shim, K., Sellis, T. K., and Nau, D. Improvements on a heuristic algorithm for multiple-query optimization. *Data and Knowledge Engineering 12*, 2 (1994), 197–222.

[103] Stocker, M., Seaborne, A., and Bernstein, A. SPARQL basic graph pattern optimization using selectivity estimation. In *WWW* (Beijing, China, 2008).

[104] Stuckenschmidt, H., Vdovjak, R., Houben, G.-J., and Broekstra, J. Index structures and algorithms for querying distributed rdf repositories. In *WWW* (New York, USA, 2004).

[105] Sun, C., Chan, C. Y., and Goenka, A. K. Multiway SLCA-based keyword search in xml data. In *WWW* (Banff, Canada, 2007).

[106] Tao, Y., and Papadias, D. MV3R-Tree: A spatio-temporal access method for timestamp and interval queries. In *VLDB* (Rome, Italy, 2001).

[107] Tappolet, J., and Bernstein, A. Applied temporal RDF: Efficient temporal querying of RDF data with SPARQL. In *ESWC* (Heraklion, Greece, 2009).

[108] Tomita, E., and Seki, T. An efficient branch-and-bound algorithm for finding a maximum clique. In *DMTCS* (Dijon, France, 2003).

[109] Tran, T., Wang, H., Rudolph, S., and Cimiano, P. Top-k exploration of query candidates for efficient keyword search on graph-shaped (RDF) data. In *ICDE* (Shanghai, China, 2009).

[110] Trigoni, N., Yao, Y., Demers, A. J., Gehrke, J., and Rajaraman, R. Multiquery optimization for sensor networks. In *DCOSS* (Marina del Rey, USA, 2005).

[111] Ullman, J. D. Information integration using logical views. In *ICDT* (Delphi, Greece, 1997).

[112] Vismara, P., and Valery, B. Finding maximum common connected subgraphs using clique detection or constraint satisfaction algorithms. In *MCO* (Metz, France, 2008).

[113] Wang, Q., Yu, T., Li, N., Lobo, J., Bertino, E., Irwin, K., and won Byun, J. On the correctness criteria of fine-grained access control in relational databases. In *VLDB* (Vienna, Austria, 2007).

[114] Weiss, C., Karras, P., and Bernstein, A. Hexastore: sextuple indexing for semantic web data management. In *VLDB* (Auckland, New Zealand, 2008).

[115] Yang, H. Z., and Larson, P. Query transformation for PSJ-queries. In *VLDB* (Brighton, England, 1987).

[116] Yang, J., and Widom, J. Incremental computation and maintenance of temporal aggregates. *The VLDB Journal 12*, 3 (2003), 262–283.

[117] Zhao, P., and Han, J. On graph query optimization in large networks. In *VLDB* (Singapore, 2010).

[118] Zhao, Y., Deshpande, P., Naughton, J. F., and Shukla, A. Simultaneous optimization and evaluation of multiple dimensional queries. In *SIGMOD* (Seattle, USA, 1998).