Efficient Big Data query answering in the presence of constraints

Damián Bursztyn

NNT : 2016SACLS567

THÈSE DE DOCTORAT
DE
L'UNIVERSITÉ PARIS-SACLAY
PRÉPARÉE À
L'UNIVERSITÉ PARIS-SUD

AU SEIN DE L'INRIA SACLAY ET DU LABORATOIRE D'INFORMATIQUE DE L'ÉCOLE POLYTECHNIQUE (LIX)

ÉCOLE DOCTORALE N°580
Sciences et technologies de l'information et de la communication

Spécialité de doctorat : Informatique

Par

M. Damián BURSZTYN

Répondre efficacement aux requêtes Big Data en présence de contraintes

Thèse présentée et soutenue à Saclay, le 15 décembre 2016 :

Composition du Jury :

Mme Christine Froidevaux	Professeur, Université Paris-Sud	Président
M. Thomas Neumann	Professeur, Technische Universität München	Rapporteur
Mme Marie-Christine Rousset	Professeur, Université de Grenoble	Rapporteur
M. Serge Abiteboul	Directeur de recherche, INRIA & ENS Cachan	Examinateur
M. Diego Calvanese	Professeur, Free University of Bozen-Bolzano	Examinateur
M. Dario Colazzo	Professeur, Université Paris-Dauphine	Examinateur
Mme Ioana Manolescu	Directeur de recherche, INRIA & Ecole Polytechnique	Directrice de thèse
M. François Goasdoué	Professeur, Université de Rennes 1 & INRIA	Co-directeur de thèse

université PARIS-SACLAY
ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Résumé

Répondre efficacement aux requêtes Big Data en présence de contraintes

Les contraintes sont les artéfacts fondamentaux permettant de donner un sens aux données. Elles garantissent que les données sont conformes aux besoins des applications. L'objet de cette thèse est d'étudier deux problématiques liées à la gestion efficace des données en présence de contraintes.

Nous abordons le problème de répondre efficacement à des requêtes portant sur des données, en présence de *contraintes déductives*. Cela mène à des données *implicites* dérivant de données explicites et de contraintes. Les données implicites requièrent une étape de *raisonnement* afin de calculer les réponses aux requêtes. Le raisonnement par *reformulation* des requêtes compile les contraintes dans une requête modifiée qui, évaluée à partir des données explicites uniquement, génère toutes les réponses fondées sur les données explicites et implicites. Comme les requêtes reformulées peuvent être complexes, leur évaluation est souvent difficile et coûteuse.

Nous étudions l'optimisation de la technique de réponse aux requêtes par reformulation dans le cadre de *l'accès aux données à travers une ontologie*, où des requêtes conjonctives SPARQL sont posées sur un ensemble de faits RDF sur lesquels des contraintes RDF Schema (RDFS) sont exprimées. La thèse apporte les contributions suivantes. (i) Nous généralisons les langages de reformulation de requêtes précédemment étudieés, afin d'obtenir un *espace de reformulations* d'une requête posée plutôt qu'une unique reformulation. (ii) Nous présentons des *algorithmes* effectifs et efficaces, fondés sur un modèle de coût, permettant de sélectionner une requête reformulée ayant le plus faible coût d'évaluation. (iii) Nous montrons expérimentalement que notre technique améliore significativement la performance de la technique de réponse aux requêtes par reformulation.

Au-delà de RDFS, nous nous intéressons aux langages d'ontologie pour lesquels répondre à une requête peut se réduire à l'évaluation d'une certaine formule de la Logique du Premier Ordre (obtenue à partir de la requête et de l'ontologie), sur les faits explicites uniquement. (iv) Nous généralisons la technique de reformulation optimisée pour RDF, mentionnée ci-dessus, aux formalismes pour répondre à une requête LPO-réductible.

(v) Nous appliquons cette technique à la Logique de Description DL-Lite$_{\mathcal{R}}$ sous-jacente au langage OWL2 QL du W3C, et montrons expérimentalement ses avantages dans ce contexte.

Nous présentons également, brièvement, un travail en cours sur le problème consistant à fournir des chemins d'accès efficaces aux données dans les systèmes Big Data. Nous proposons d'utiliser un ensemble de systèmes de stockages hétérogènes afin de fournir une meilleure performance que n'importe lequel d'entre eux, utilisé individuellement. Les données stockées dans chaque système peuvent être décrites comme des vues matérialisées sur les données applicatives. Répondre à une requête revient alors à réécrire la requête à l'aide des vues disponibles, puis à décoder la réécriture produite comme un ensemble de requêtes à exécuter sur les systèmes stockant les vues, ainsi qu'une requête les combinant de façon appropriée.

Mots clés— Web sémantique, Optimisation des requêtes, Répondre à des requêtes en présence de contraintes, Reformulation des requêtes, Polystores

université PARIS-SACLAY
ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Abstract

Efficient Big Data query answering in the presence of constraints

Constraints are the essential artefact for giving meaning to data, ensuring that it fits real-life application needs, and that its meaning is correctly conveyed to the users. This thesis investigates two fundamental problems related to the efficient management of data in the presence of constraints.

We address the problem of efficiently answering queries over data in the presence of *deductive constraints*, which lead to *implicit* data that is entailed (derived) from the explicit data and the constraints. Implicit data requires a *reasoning* step in order to compute complete query answers, and two main query answering techniques exist. Data *saturation* compiles the constraints into the database by making all implicit data explicit, while query *reformulation* compiles the constraints into a modified query, which, evaluated over the explicit data only, computes all the answer due to explicit and/or implicit data. So far, reformulation-based query answering has received significantly less attention than saturation. In particular, reformulated queries may be complex, thus their evaluation may be very challenging.

We study optimizing reformulation-based query answering in the setting of *ontology-based data access*, where SPARQL conjunctive queries are answered against a set of RDF facts on which constraints hold. When RDF Schema is used to express the constraints, the thesis makes the following contributions. (i) We generalize prior query reformulation languages, leading to a *space of reformulated queries* we call JUCQs (joins of unions of conjunctive queries), instead of a single fixed reformulation. (ii) We present effective and efficient *cost-based algorithms* for selecting from this space, a reformulated query with the lowest estimated cost. (iii) We demonstrate through experiments that our technique drastically improves the performance of reformulation-based query answering while always avoiding "worst-case" performance.

Moving beyond RDFS, we consider the large and useful set of ontology languages enjoying FOL *reducibility of query answering*: answering a query can be reduced to evaluating a certain first-order logic (FOL) formula (obtained from the query and ontology) against only the explicit facts. (iv) We generalize the above-mentioned JUCQ-based optimized reformulation technique to improve performance in any FOL-reducible setting, and (v) we instantiate this framework to the DL-Lite$_{\mathcal{R}}$ Description Logic underpinning

the W3C's OWL2 QL ontology language, demonstrating significant performance advantages in this setting also.

We also report on current work regarding the problem of providing efficient data access paths in Big Data stores. We consider a setting where a set of different, heterogeneous storage systems can be used side by side to provide better performance than any of them used individually. In such a setting, the data stored in each system can be described as views over the application data. Answering a query thus amounts to rewrite the query using the available views, and then to decode the rewriting into a set of queries to be executed on the systems holding the views, and a query combining them appropriately.

Keywords— Semantic Web, Query optimization, Query answering under constraints, Query reformulation, Hybrid stores

Acknowledgements

I am deeply grateful to all those who helped and supported me during these years, leading to the completion of this thesis.

I would like to start by thanking my advisors Ioana Manolescu and François Goasdoué for sharing their passion and enthusiasm with me, as well as dedication and help through this journey. Completing this thesis would not have been possible, nor as fun as it was without them. It has been more than 3 years since they welcomed me in the OAK team as an intern. Many things changed, François moved to Lannion, the team name changed to CEDAR, people graduated and new people joined the team, etc., however their support, guidance and hard-work attitude only growed with time. They were advisors in every meaning of the word, always encouraging me to give my best and improve myself, and guiding me during the process. Doing a PhD is a long-term, hard-working journey, which I was lucky to share with Ioana and François. They were always demanding, but equally commited.

I am grateful to Thomas Neumann and Marie-Christine Rousset for thoroughly reading my thesis and for their valuable feedback. Further, I would like to thank Serge Abiteboul, Diego Calvanese, Dario Colazzo, and Christine Froidevaux for being part of my examining committee; I am very honored.

During the PhD, I was also fortunate to have productive collaborations with my colleagues from Inria, Université Paris-Sud and École polytechnique Stamatis Zampetakis, Francesca Bugiotti, Alessandro Solimando, Michaël Thomazo, Ioana Ileana and Alexandra Roatis. They are all contributors to this thesis. In addition, it was a pleasure to serve École polytechnique under the guidance of Yanlei Diao and Ioana. I keep great memories of my days of teaching service.

Furthermore, I had the opportunity to be in contact with many people from other institutions and visit them. I want to thank Diego Calvanese for his invitation to visit KRDB at Free University of Bozen-Bolzano, and Martin Rezk, Davide Lanti, and Guohui Xiao for their valuable comments and feedbacks on efficient `FOL` reducible query answering. I want to thank Julien Leblay for inviting me to visit AIST Tokyo, it was a great experience discussing with him and the other people of his group about our research projects. Special thanks to Alin Deutsch for inviting me to visit UCSD and with whom we worked together in the Scalable Hybrid Stores project. Working with him has been a truly enriching experience, during which he provided a more formal perspective on the problems

we were tackling, always in a patient and didactic way. Also want to thank Rana, it was a pleasure to work together both in UCSD and in Paris. They are also contributors to this thesis.

Also want to thank to all the great people I was lucky to meet and share experiences with during my time in OAK and CEDAR. To Asterios, Jesus, Katerina, Juan, Raphael, Andrés, Danai, Zoi, Benjamin, Sofoklis, Sejla, Oscar, Tien Duc, Enhui, Soudip, Paul, Benoit, Tushar, Guillaume, Despoina, Javier, Swen and Gianluca... Thank you!

I would like to thank Esteban, Sabri, Migue, Eli, Rafa, Nati, Gutes, Romi, Nacho, Ali, Fav, Fran, Maria, Diego and Alan, as well as all my other friends back in Buenos Aires. Despite the distance, they were always near.

Want to thank those friends in Paris, which I met over these years, Carolina, Pete, André, Carmen, Quentin, Mary, George, Ioana, Fab, Nelly and Dimitri.

Especially, I would like to thank Héloïse for her support throughout the (hard) period of time while I wrote the thesis.

Last, but not least, I wish to thank Nico, Laura, Rosalia, Viviana and Ariel, who encouraged and helped me during these past years; Jaco, my sisters, Michelle and Gisela and my parents, Jorge and Silvia who raised me, supported me, and loved me.

To my family, for their love, support and encouragement.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Big Data

The Web 2.0, as it is popularly called, refers to a second stage of World Wide Web whose main ingredients are: (i) the apparition of dynamic content pages, services and applications providing the users a richer experience, (ii) the wide acceptance and (huge) growth of Software as a Service (SaaS), through lightweight integration protocols, web APIs, etc., (iii) massive participation, with a penetration of more than 46% of the world population, and (iv) a change in the user involvement and consumption paradigme, not being simple consumers but also content producers (e.g., Wikipedia, an online encyclopedia on which anyone can write or edit articles; Blogger, a blog-publishing service that allows users to create blogs entries as well as comment on them; Twitter, an online social networking service that enables users to share 140-character messages; Youtube, a video sharing plataform where users can also comment and rate the videos; WhatsApp Messenger, an instant message client to exchange text messages, audio messages, documents, images, video, etc. between two or more users; Facebook, a social networking service with more than 1.65 billion monthly active users as of March 31, 2016[1] etc.).

From a data management perspective, the Web 2.0 was disruptive in terms of data volume. We create 2.5 exabytes of data every day! In other words, 90% of the world's data was created in just the past 24 months[2]. At the same time, hardware evolution brought cheaper and a continuous increase in the memory capacity, the emergence of a variety of computation and storage technologies, such as GPUs, FPGAs, etc. [94], high-speed networks (e.g., Infinibad), and finally but not less important cloud computing emerged.

These events did not go unnoticed by the machine learning branch of the artificial intelligence community, nor by the the data management and distributed systems communities, which provided the systems and solutions (e.g., Hadoop, Spark, Flink, Kafka, etc.)

[1] `https://newsroom.fb.com/company-info/`

[2] `https://www-01.ibm.com/software/data/bigdata/what-is-big-data.html`

needed to support this new generation of Big Data (centric) applications.

1.2 Semantic Web

We live in times of the Third Generation Web: Web 3.0, also known as Semantic Web. Unlike traditional knowledge-representation systems, typically centralized [18], the Semantic Web is meant to be a world-wide distributed architecture; a step closer to Tim Berners-Lee dream back in 2000 [17]: where computers become capable of analyzing all the data on the Web.

The Semantic Web is an extension of the World Wide Web, in which the content is given structure, thus enabling a reliable way for the computers to process the semantics and therefore manipulate the data in a meaningful way. However, to accomplish such vision a set of data models and formats for specifying semantic descriptions of Web resources is needed. For such purpose the World Wide Web Consortium (W3C) presented the Resource Description Framework (RDF), on top of which the Semantic Web stack is built. RDF is a flexible data model that allows to express statements about resources (uniquely identified by their URI) in the form of subject-predicate-object expressions. To enhance the descriptive power of RDF datasets, the W3C proposed the RDF Schema (RDFS) [105] and Web Ontology Language (OWL) [102], facilitating the representation of semantic constraints (a.k.a. ontological constraints) between the classes and the properties used. The set of facts together with the logical statements on memberships of individuals and in classes or relationships between individuals form a *knowledge base*.

In a nutshell, current popularity and usage of ontologies in the Web is due to four major reasons [4]:

- Their flexible and natural way of structuring documents in multidimensional ways, allowing to find relevant information through very large documents collections.
- The logical formal semantics of ontologies provide means of inference, enabling reasoning. Therefore, it is possible for an ontology to be interpreted and processed by machines.
- Ontologies allow making concepts precise, improving Web search. For example, when searching for the word “aleph”, we could specialize the concept in an ontology, *book:aleph*, resulting in the book written by J. L. Borges, the one from P. Coelho, etc., and avoiding unwanted answers where the term is used with another connotation (like those referring to the letter with the same name or to the sequence of numbers used to represent the cardinality of infinite sets that can be well-ordered).
- Ontologies serve as local join between heterogeneous information sources. Moreover, their inference potential helps to automatically integrate different data sources.

For instance, if the famous argentinian writer J. L. Borges appears in a document, then he has an associated URI, through which all the other resources (books, articles, prizes, etc.) refer to him. Then, the RDF version of the British National Bibliography[3] developed by British Library, and expressing relevant information about books, authors, publishers, etc., will contain the books written by Borges, the publishers of such books, publication place and year. At the same time, GeoNames[4] provides further information regarding the publication places, and DBPedia [5] (the semantic counterpart of Wikipedia [6]) will add more information about Borges, like his political opinion, family, etc. All the above-mentioned interlinked datasets are part of Linked Open Data[7]

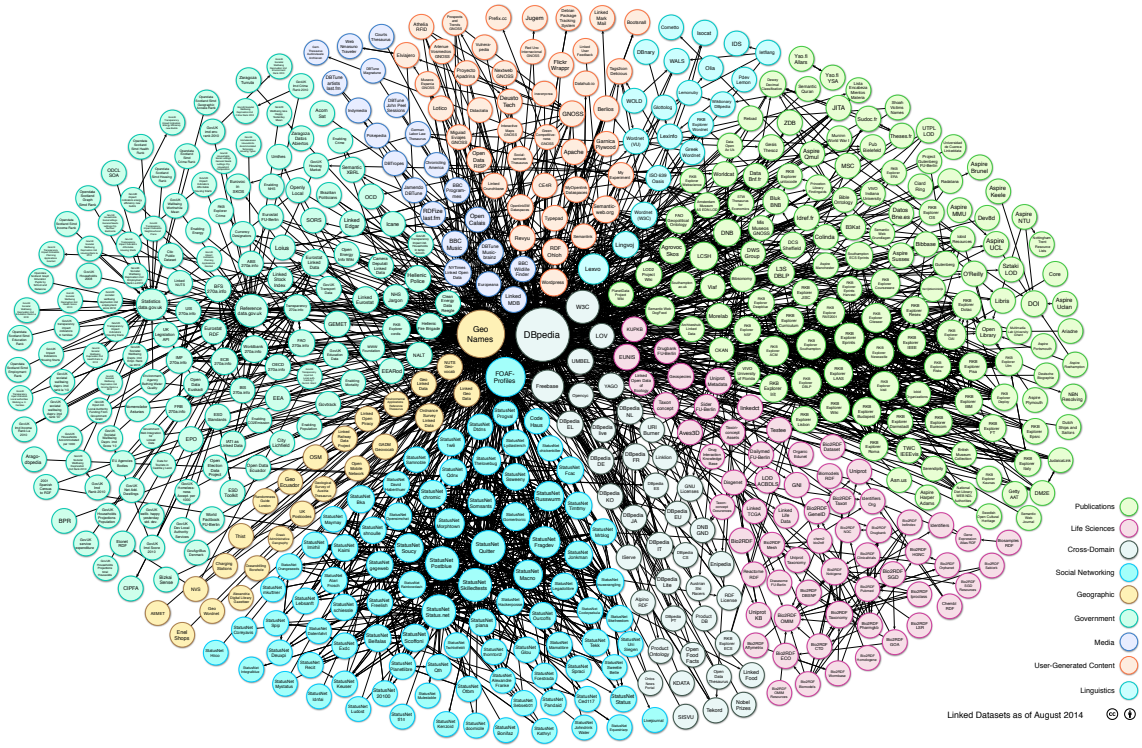

Figure 1.1: The Linked Open Data cloud as of April 2014.

Figure 1.1 [90] illustrates some of the most well known (RDF) datasets in the Linked Open Data cloud as of April 2014. Each node in the graph corresponds to one RDF dataset, while the node diameter reflects the size of the respective dataset. In addition, there is an edge between two nodes if the two datasets have some URIs in common (i.e., the datasets are interlinked). As reported in [90], the size of the Semantic Web is growing, almost doubling its size every year.

[3] http://www.bl.uk/bibliographic/datafree.html
[4] www.geonames.org/
[5] http://wiki.dbpedia.org/
[6] https://en.wikipedia.org/wiki/Main_Page
[7] http://lod-cloud.net/

To exploit this wealth of data, the SPARQL query language has been defined [106]; subsequently, novel techniques and algorithms have been proposed for the processing of SPARQL queries, based on vertical partitioning [1], indexing [107], efficient join processing [82], view selection [49], RDF management systems and optimized triple stores [108, 110, 113, 83], to name a few.

1.3 Motivations and studied problem

In this thesis, we consider the setting of ontology-based data access (OBDA) [72], which aims at exploiting a database, i.e., *facts*, on which hold ontological constraints, i.e., *deductive* constraints modeling the application domain under consideration. For instance, an ontology may specify that any author is a human, has a name, and must have authored some papers. Ontological constraints may greatly increase the *usefulness* of a database: for instance, a query asking for all the humans must return all the authors, just because of a constraint stating they are human; one does not need to store a human tuple in the database for each author. The data interpretations enabled by the presence of constraints has made OBDA a technique of choice when modeling complex real-life applications. For instance, in the medical domain, Snomed Clinical Terms is a biomedical ontology providing a comprehensive clinical terminology; the British Department of Health has a roadmap for standardizing medical records across the country, using this ontology etc..

While query answering under constraints is a classical database topic [3], research on OBDA has bloomed recently through many ontological constraints languages, e.g., Datalog$^{\pm}$ [31], Description Logics [13] and Existential Rules [14], or RDF Schema for RDF graphs. OBDA *query answering* is the task of computing the answer to the given query, by taking into account both the facts and the constraints holding on them. In contrast, query evaluation as performed by database servers leads to computing only the answers derived from the data (facts), while ignoring the constraints.

Two main methods exist for OBDA query answering, both of which consists of a *reasoning* step, either on data or on queries, followed by a *query evaluation* step. A first reasoning method is *data saturation* (a.k.a. *closure*). This consists of pre-computing and adding to the data all its implicit information, to make it explicit. Answering queries through saturation, then, amounts to evaluating the queries on the saturated data. While saturation leads to efficient query processing, it requires time to be computed, space to be stored, and must be recomputed upon updates. Moreover, data saturation may not always be an option as it is infinite in many OBDA settings. The alternative reasoning step is *query reformulation*. This consists in turning a query into a *reformulated query*, which, evaluated against non-saturated data, yields the exact answers to the original query. Since reformulation takes place at query time, it is intrinsically robust to updates; the query reformulation process in itself is also typically very fast, since it only operates on the query, not on the data. However, reformulated queries are often much more complex than the original ones, thus their evaluation may be costly or even unfeasible.

This thesis addresses the problem of *efficient OBDA query answering* in RDF, the prominent W3C standard for the Semantic Web, and in the DL-Lite$_\mathcal{R}$ description logic which underpins OWL2 QL, the W3C standard for semantic-rich data management.

Saturation-based query answering has received much attention in RDF, compared to reformulation-based query answering. In contrast, in DL-Lite$_\mathcal{R}$, since data saturation may be infinite, the focus is on reformulation-based query answering. In both data models, all the existing reformulation-based query answering techniques amounts to reformulating an incoming query with ontological constraints, so as to obtain a *single* reformulated query whose evaluation yields the correct answers. Importantly, the fact that there exists a single reformulated query results from the adopted query reformulation language.

The idea developed in this thesis to generalize the query reformulation languages used in the literature, in order to (i) encompass all the current reformulation-based query answering settings in RDF and DL-Lite$_\mathcal{R}$, and (ii) to allow a space of equivalent yet different reformulated queries, among which we can pick one with lowest (estimated) evaluation cost.

1.4 Contributions and outline

The report and the contributions to the above OBDA query answering problem are organized as follows:

Chapter 2 provides background about two well-known W3C's Semantic Web standards, the popular RDF data model and its associated SPARQL query languages, as well as the DL-Lite$_\mathcal{R}$ description logic and conjunctive queries that underpins OWL2 QL for semantic-rich data management.

Chapter 3 considers optimizing reformulation-based query answering in a setting where SPARQL conjunctive queries are posed against RDF facts on which constraints expressed by an RDF Schema hold. The contributions of this chapter are the following:

- To generalize prior query reformulation languages, leading to investigating a space of reformulated queries, instead of a single reformulation.
- To introduce an effective and efficient cost-based algorithm for selecting from this space, the reformulated query with the lowest estimated cost.
- To present extensive experiments showing that our technique enables reformulation-based query answering where the state-of-the-art approaches are simply unfeasible, while it may decrease its cost by orders of magnitude in other cases.

Chapter 4 generalizes the idea developed in the preceding chapter to devise a *novel optimization framework for reformulation-based query answering in First Order Logic (*FOL*) ontology-based data access settings*. The contributions of this chapter are the following:

- To extend the language of FOL query reformulations beyond those considered so far in the literature, and investigate *several (equivalent)* FOL *query reformulations* of a given query, out of which we pick one likely to lead to the best evaluation performance.
- To apply the above mentioned framework to the DL-Lite$_\mathcal{R}$ Description Logic underpinning the W3C's OWL2 QL ontology language.
- To demonstrate through experiments the framework performance benefits when two leading SQL systems, one open-source and one commercial, are used for evaluating the query reformulations.

Chapter 5 concludes this thesis and presents ongoing work as well as perspectives for future work.

Chapter 2

Preliminaries

This chapter presents the background needed by the presentation of the research work performed in the thesis. First, in Section 2.1 we introduce RDF, the graph-based data model promoted by the W3C standard for Semantic Web applications. Then, in Section 2.2 we revisit the DL-Lite$_\mathcal{R}$ description logic underpinning the W3C's OWL 2 QL standard for semantic-rich data management.

2.1 RDF

The Resource Description Framework (RDF) is a graph-based data model promoted by the W3C as the standard for Semantic Web applications. Its associated query language is SPARQL. RDF graphs are often *large* and *heterogeneous*, i.e., resources described in an RDF graph may have very different sets of properties.

An *RDF graph* (or *graph*, in short) is a set of *triples* of the form $\mathtt{s}\ \mathtt{p}\ \mathtt{o}$. A triple states that its *subject* $\mathtt{s}$ has the *property* $\mathtt{p}$, and the value of that property is the *object* $\mathtt{o}$. We consider only well-formed triples, as per the RDF specification [104], using uniform resource identifiers (URIs), typed or un-typed literals (constants) and blank nodes (unknown URIs or literals).

Blank nodes are essential features of RDF allowing to support *unknown URI/literal tokens*. These are conceptually similar to the variables used in incomplete relational databases based on *V-tables* [3, 58], as shown in [48].

Notations. We use $\mathtt{s}$, $\mathtt{p}$, $\mathtt{o}$ and $\_{:}b$ in triples as placeholders. Literals are shown as strings between quotes, e.g., "$string$". Finally, the set of values – URIs (U), blank nodes (B), and literals (L) – of an RDF graph $\mathtt{G}$ is denoted $\mathtt{Val}(\mathtt{G})$.

Figure 2.1 (top) shows how to use triples to describe resources, that is, to express class (unary relation) and property (binary relation) assertions. The RDF standard [104] provides a set of built-in classes and properties, as part of the rdf: and rdfs: pre-defined

Assertion	Triple	Relational notation
Class	s rdf:type o	o(s)
Property	s p o	p(s, o)

Constraint	Triple	OWA interpretation
Subclass	s rdfs:subClassOf o	$\mathtt{s} \subseteq \mathtt{o}$
Subproperty	s rdfs:subPropertyOf o	$\mathtt{s} \subseteq \mathtt{o}$
Domain typing	s rdfs:domain o	$\Pi_{\text{domain}}(\mathtt{s}) \subseteq \mathtt{o}$
Range typing	s rdfs:range o	$\Pi_{\text{range}}(\mathtt{s}) \subseteq \mathtt{o}$

Figure 2.1: RDF (top) & RDFS (bottom) statements.

namespaces. We use these namespaces exactly for these classes and properties, e.g., rdf:type specifies the class(es) to which a resource belongs.

Example 1 (**RDF graph**). *The RDF graph* G *below comprises information about a book, identified by* doi_1*: its author (a blank node* $\_:b_1$ *related to the author name, which is a literal), title and date of publication.*

$$\mathtt{G} = \begin{array}{l}\{\text{doi}_1 \text{ rdf:type Book},\\ \text{doi}_1 \text{ writtenBy } \_:b_1,\\ \text{doi}_1 \text{ hasTitle “El Aleph”},\\ \_:b_1 \text{ hasName “J. L. Borges”},\\ \text{doi}_1 \text{ publishedIn “1949”}\}\end{array}$$

An alternative, and sometimes more intuitive, way to visualize and represent all the triples information is using a graph, where there is a node for each (distinct) subject or object, labeled with its value; a triple is represented as a directed edge, labeled with the property value, between the subject node and the object node. Figure 2.2 presents an equivalent representation of G. Resources (URIs) are represented by blue rounded-nodes, blank nodes are illustrated with black rounded-nodes while literals are depicted with black square-nodes; black directed edges are use to exhibit the properties.

2.1.1 RDF Schema and entailment

A valuable feature of RDF is RDF Schema (RDFS) that allows enhancing the descriptions in RDF graphs. RDFS triples declare *semantic constraints* between the classes and the properties used in those graphs.

Figure 2.1 (bottom) shows the allowed constraints and how to express them; *domain* and *range* denote respectively the first and second attribute of every property. The RDFS constraints (Figure 2.1) are interpreted under the open-world assumption (OWA) [3].

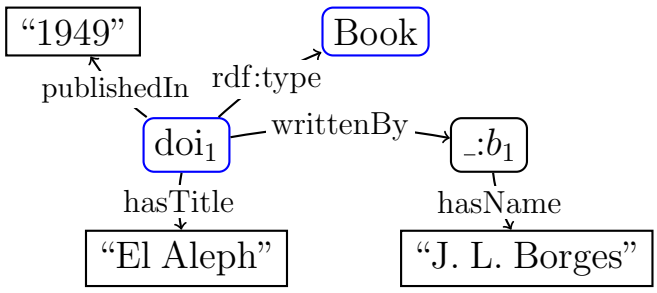

Figure 2.2: Sample RDF graph.

More specifically, when working with the RDF data model, if the triples hasFriend rdfs:domain Person and Anne hasFriend Marie hold in the graph, then so does the triple Anne rdf:type Person. The latter is due to the rdfs:domain constraint in Figure 2.1.

RDF entailment. *Implicit triples* are an important RDF feature, considered part of the RDF graph even though they are not explicitly present in it, e.g., Anne rdf:type Person above. W3C names *RDF entailment* the mechanism through which, based on a set of explicit triples and some *entailment rules*, implicit RDF triples are derived. We denote by $\vdash^{i}_{\text{RDF}}$ *immediate entailment*, i.e., the process of deriving new triples through a *single* application of an entailment rule. More generally, a triple `s p o` is entailed by a graph `G`, denoted $\texttt{G} \vdash_{\text{RDF}} \texttt{s p o}$, if and only if there is a sequence of applications of immediate entailment rules that leads from `G` to `s p o` (where at each step of the entailment sequence, the triples previously entailed are also taken into account).

Example 2 (RDFS). *Assume that the RDF graph* `G` *in Example 1 is extended with the following constraints.*

- *books are publications:*
  Book rdfs:subClassOf Publication
- *writing something means being an author:*
  writtenBy rdfs:subPropertyOf hasAuthor
- *books are written by people:*
  writtenBy rdfs:domain Book
  writtenBy rdfs:range Person

The resulting graph is depicted in Figure 2.3. Constraints are represented by blue edges while its implicit triples are those depicted by gray dashed-line edges.

Saturation. The immediate entailment rules allow defining the finite *saturation* (a.k.a. closure) of an RDF graph `G`, which is the RDF graph $\texttt{G}^{\infty}$ defined as the fixed-point obtained by repeatedly applying $\vdash^{i}_{\text{RDF}}$ rules on `G`.

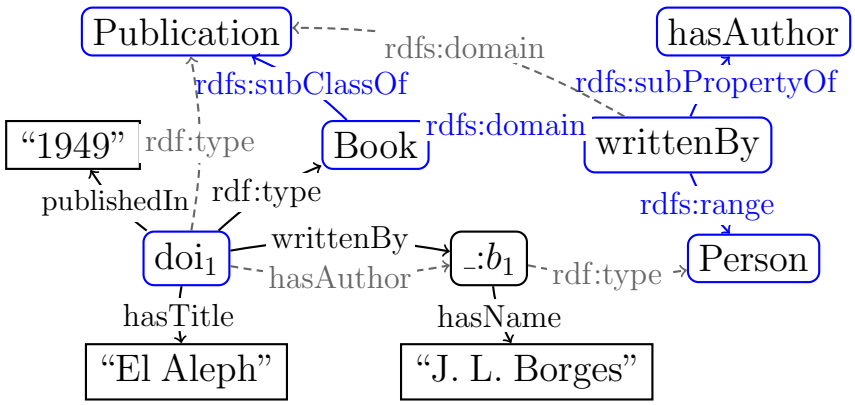

Figure 2.3: Sample RDF graph with RDFS constraints.

The saturation of an RDF graph is unique (up to blank node renaming), and does not contain implicit triples (they have all been made explicit by saturation). An obvious connection holds between the triples entailed by a graph $\mathtt{G}$ and its saturation: $\mathtt{G} \vdash_{\text{RDF}} \mathtt{s\ p\ o}$ if and only if $\mathtt{s\ p\ o} \in \mathtt{G}^{\infty}$.

RDF entailment is part of the RDF standard itself; in particular, *the answers to a query posed on* $\mathtt{G}$ *must take into account all triples in* $\mathtt{G}^{\infty}$, since *the semantics of an RDF graph is its saturation* [106].

2.1.2 BGP Queries

We consider the well-known subset of SPARQL consisting of (unions of) *basic graph pattern* (BGP) queries, modeling the SPARQL conjunctive queries. Subject of several recent works [48, 93, 49, 87], BGP queries are the most widely used subset of SPARQL queries in real-world applications [87]. A BGP is a set of *triple patterns*, or triples/atoms in short. Each triple has a subject, property and object, some of which can be variables.

Notations. In the following we use the conjunctive query notation $q(\bar{x})$:- $t_1, \ldots, t_\alpha$, where $\{t_1, \ldots, t_\alpha\}$ is a BGP; the query head variables $\bar{x}$ are called *distinguished variables*, and are a subset of the variables occurring in $t_1, \ldots, t_\alpha$; for boolean queries $\bar{x}$ is empty. The head of q is $q(\bar{x})$, and the body of q is $t_1, \ldots, t_\alpha$. We use x, y, z, etc. to denote variables in queries. We denote by $\mathtt{VarBl}(q)$ the set of variables *and* blank nodes occurring in the query q.

Query evaluation. Given a query q and an RDF graph $\mathtt{G}$, the *evaluation of* q *against* $\mathtt{G}$ is:

$$q(\mathtt{G}) = \{\bar{x}_\mu \mid \mu : \mathtt{VarBl}(q) \to \mathtt{Val}(\mathtt{G}) \text{ is a total assignment such that } t_1^\mu \in \mathtt{G}, t_2^\mu \in \mathtt{G}, \ldots, t_\alpha^\mu \in \mathtt{G}\}$$

where we denote by t^μ the result of replacing every occurrence of a variable or blank node $e \in \texttt{VarBl}(q)$ in the triple t, by the value $\mu(e) \in \texttt{Val}(\texttt{G})$.

Note that evaluation *treats the blank nodes in a query exactly as it treats non-distinguished variables* [4]. Thus, in the sequel, without loss of generality, we consider queries where all blank nodes have been replaced by (new) distinct non-distinguished variables.

Query answering. The evaluation of q against $\texttt{G}$ uses only $\texttt{G}$'s explicit triples, thus may lead to an incomplete answer set. The (complete) *answer set* of q against $\texttt{G}$ is obtained by the evaluation of q against $\texttt{G}^\infty$, denoted by $q(\texttt{G}^\infty)$.

Example 3 (Query answering). *The following query asks for the names of authors of books somehow connected to the literal 1949:*

$q(x_3)$:- x_1 hasAuthor x_2, x_2 hasName x_3, x_1 x_4 "1949"

Its answer against the graph in Figure 2.3 is $q(\texttt{G}^\infty) = \{\langle$"J L. Borges"$\rangle\}$. *The answer results from* $\texttt{G} \vdash_{\text{RDF}} \text{doi}_1$ hasAuthor $\_{:}b_1$ *and the assignment* $\mu = \{x_1 \leftarrow \text{doi}_1, x_2 \leftarrow \_{:}b_1, x_3 \leftarrow$ "J L. Borges", $x_4 \leftarrow \text{publishedIn}\}$. *Observe that evaluating* q *directly against* $\texttt{G}$ *leads to the empty answer, which is obviously incomplete.*

2.1.3 Query answering techniques

Answering queries over data in the presence of *deductive constraints* requires a *reasoning* step in order to compute complete query answers. Two main query answering techniques exist:

Saturation-based query answering. Compiles the constraints into the database by making all implicit data explicit. This method is straightforward and easy to implement. Its disadvantages are that dataset saturation requires computation time and storage space for all the entailed triples; moreover, the saturation must be recomputed upon every update. Incremental algorithms for saturation maintenance had been proposed in previous work [48]. However, the recursive nature of entailment makes this process costly (in time and space) and this method not suitable for datasets with a high rate of updates. Further, for some ontology languages saturation *may be infinite* (see 2.2).

Table 2.1, extracted from the cited work [48], presents the characteristics of well-known datasets and shows that saturation adds between 10% and 41% to the dataset size, while multiset-based saturation (required for the incrementally maintaining the saturation technique presented by the authors) increase the size between 116% and 227%.

Reformulation-based query answering. Compiles the constraints into a modified query, which, evaluated over the explicit data only, computes all the answer due to explicit and/or implicit data. The main advantage of this method is that its robust to update, there is no need to (re)compute the closure of the dataset. On the other hand, in gen-

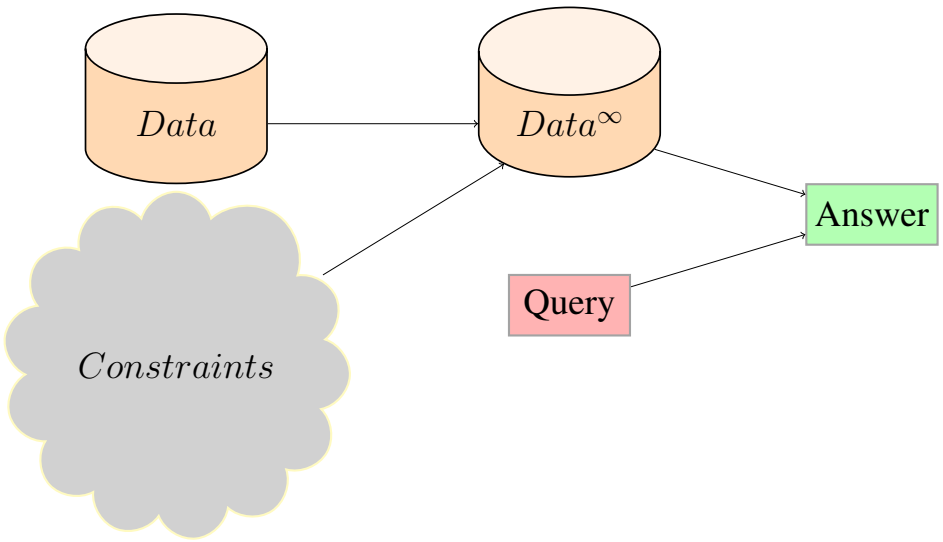

Figure 2.4: Saturation-based query answering overview.

Graph	#Schema	#Instance	#Saturation	Saturation increase (%)	#Multiset	Multiset increase (%)
Barton [15]	101	33,912,142	38,969,023	14.91	73,551,358	116.89
DBpedia [39]	5666	26,988,098	29,861,597	10.65	66,029,147	227.37
DBLP [38]	41	8,424,216	11,882,409	41.05	18,699,232	121.97

Table 2.1: Datasets and saturation characteristics [48].

eral, reformulated queries are complex, hence inefficiently handled by modern query processing engines.

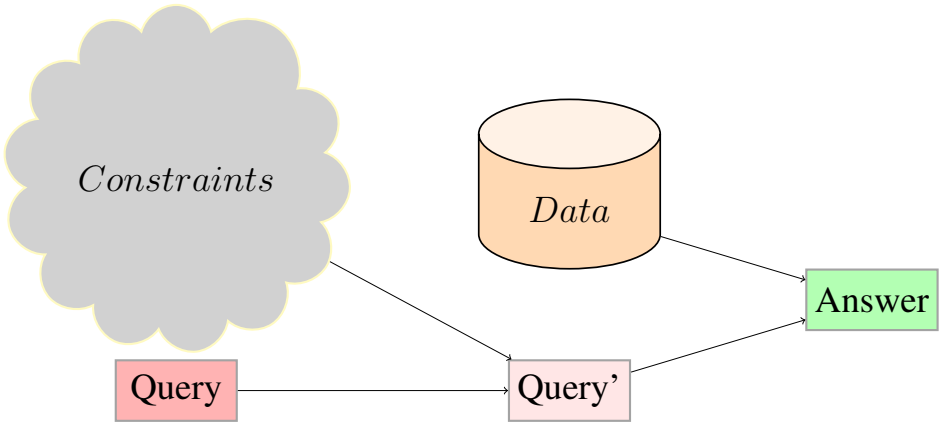

Figure 2.5: Reformulation-based query answering overview.

2.1.4 The database fragment of RDF

The *database (DB)* fragment of RDF [48] is, to the best of our knowledge, the most expressive RDF fragment for which both *saturation-* and *reformulation-based RDF query answering* has been defined and practically experimented. This DB fragment is defined by:

- *Restricting RDF entailment* to the RDF Schema constraints only (Figure 2.1), a.k.a. RDFS entailment. Consequently, the DB fragment focuses only on the application domain knowledge, a.k.a. ontological knowledge, and not on the RDF meta-model knowledge which mainly begets high-level typing of subject, property and object values found in triples with abstract RDF built-in classes, e.g., $\mathrm{rdf{:}Resource}$, $\mathrm{rdfs{:}Class}$, etc.

- *Not restricting RDF graphs in any way*. In other words, any triple allowed by the RDF specification is also allowed in the DB fragment.

In this DB fragment of RDF, *Saturation-based query answering* amounts to precomputing the saturation of a database $\mathtt{db}$ using its RDFS constraints in a forward-chaining fashion, so that the *evaluation* of every incoming query q against the saturation yields the correct answer set [48]: $q(\mathtt{db}^{\infty}) = q(\mathbf{Saturate}(\mathtt{db}))$. This technique follows directly from the definitions in Section 2.1 and Section 2.1.2, and the W3C's RDF and SPARQL recommendations.

Reformulation-based query answering, in contrast, leaves the database $\mathtt{db}$ untouched and reformulates every incoming query q using the RDFS constraints in a backward-chaining fashion, $\mathbf{Reformulate}(q, \mathtt{db}) = q^{\mathrm{ref}}$, so that the *relational evaluation* of this reformulation against the (non-saturated) database yields the correct answer set [48]: $q(\mathtt{db}^{\infty}) = q^{\mathrm{ref}}(\mathtt{db})$. The $\mathbf{Reformulate}$ algorithm, introduced in [49] and extended in [48], exhaustively applies a set of 13 reformulation rules. Starting from the incoming BGP query q to answer against $\mathtt{db}$, the algorithm produces a *union of BGP queries* retrieving the correct answer set from the database, even if the latter is not saturated.

Example 4 (Query reformulation). *The reformulation of* $q(x, y)$*:-* x $\mathrm{rdf{:}type}$ y *w.r.t. the database* $\mathtt{db}$ *(obtained from the RDF graph* $\mathtt{G}$ *depicted in Figure 2.3), asking for all resources and the classes to which they belong, is:*

$$
\begin{array}{ll}
(0) & q(x,y) \text{:- } x \text{ rdf:type } y \quad \cup \\
(1) & q(x,\text{Book}) \text{:- } x \text{ rdf:type Book} \quad \cup \\
(2) & q(x,\text{Book}) \text{:- } x \text{ writtenBy } z \quad \cup \\
(3) & q(x,\text{Book}) \text{:- } x \text{ hasAuthor } z \quad \cup \\
(4) & q(x,\text{Publication}) \text{:- } x \text{ rdf:type Publication} \quad \cup \\
(5) & q(x,\text{Publication}) \text{:- } x \text{ rdf:type Book} \quad \cup \\
(6) & q(x,\text{Publication}) \text{:- } x \text{ writtenBy } z \quad \cup \\
(7) & q(x,\text{Publication}) \text{:- } x \text{ hasAuthor } z \quad \cup \\
(8) & q(x,\text{Person}) \text{:- } x \text{ rdf:type Person} \quad \cup \\
(9) & q(x,\text{Person}) \text{:- } z \text{ writtenBy } x \quad \cup \\
(10) & q(x,\text{Person}) \text{:- } z \text{ hasAuthor } x
\end{array}
$$

The terms (1)*,* (4) *and* (8) *result from* (0) *by instantiating the variable* y *with classes from* db*, namely* $\{\text{Book}, \text{Publication}, \text{Person}\}$*. Item* (5) *results from* (4) *by using the subclass constraint between books and publications.* (2)*,* (6) *and* (9) *result from their direct predecessors in the list, and are due to the domain and range constraints. Finally,* (3)*,* (7) *and* (10) *result from their direct predecessors and the sub-property constraint present in the database.*

Evaluating this reformulation against db *returns the same answer as* $q(\mathtt{G}^\infty)$*, i.e., the answer set of* q*.*

2.2 DL-Lite$_\mathcal{R}$

As commonly known, a Description Logic *knowledge base (KB)* $\mathcal{K}$ consists of a TBox $\mathcal{T}$ (ontology, or axiom set) and an ABox $\mathcal{A}$ (database, or fact set), denoted $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$, with $\mathcal{T}$ expressing constraints on $\mathcal{A}$.

Most popular Description Logic dialects [13], and in particular DL-Lite$_\mathcal{R}$ [32], build $\mathcal{T}$ and $\mathcal{A}$ from a set N_C of *concept names* (unary predicates), a set N_R of *role names* (binary predicates), and a set N_I of *individuals* (constants). The *ABox* consists of a finite number of *concept assertions* of the form $A(a)$ with $A \in N_C$ and $a \in N_I$, and of *role assertions* of the form $R(a, b)$ with $R \in N_R$ and $a, b \in N_I$. The *TBox* is a set of axioms, whose expressive power is defined by the ontology language. DL-Lite$_\mathcal{R}$ description logic [32], which is the first order logic foundation of the W3C's OWL2 QL standard for managing semantic-rich Web data, is a significant extension of the subset of RDF (comprising RDF Schema) which can be translated into description logics, a.k.a. the DL fragment of RDF; DL-Lite$_\mathcal{R}$ is also a fragment of Datalog$^\pm$ [30].

Given a role R, its *inverse*, denoted R^-, is the set: $\{(b, a) \mid R(a, b) \in \mathcal{A}\}$. We denote $N_R^\pm$ the set of roles made of all role names, together with their inverses: $N_R^\pm = N_R \cup \{r^- \mid r \in N_R\}$. For instance, supervisedBy and supervisedBy$^-$, whose meaning is *supervises*, are in $N_R^\pm$. A DL-Lite$_\mathcal{R}$ TBox constraint is either:

$(T1)$	PhDStudent	$\sqsubseteq$	Researcher
$(T2)$	$\exists$worksWith	$\sqsubseteq$	Researcher
$(T3)$	$\exists$worksWith$^-$	$\sqsubseteq$	Researcher
$(T4)$	worksWith	$\sqsubseteq$	worksWith$^-$
$(T5)$	supervisedBy	$\sqsubseteq$	worksWith
$(T6)$	$\exists$supervisedBy	$\sqsubseteq$	PhDStudent
$(T7)$	PhDStudent	$\sqsubseteq$	$\neg\exists$supervisedBy$^-$

Table 2.2: Sample TBox $\mathcal{T}$.

(i) a **concept inclusion** of the form $C_1 \sqsubseteq C_2$ or $C_1 \sqsubseteq \neg C_2$, where each of C_1, C_2 is either a concept from N_C, or $\exists R$ for some $R \in N_R^{\pm}$, and $\neg C_2$ is the complement of C_2. Here, $\exists R$ denotes the set of constants occurring in the first position in role R (i.e., the projection on the first attribute of R). For instance, $\exists$supervisedBy is the set of those supervised by somebody, while $\exists$supervisedBy$^-$ is the set of all supervisors (i.e., the projection on the first attribute of supervisedBy$^-$, hence on the second of supervisedBy);

(ii) a **role inclusion** of the form $R_1 \sqsubseteq R_2$ or $R_1 \sqsubseteq \neg R_2$, with $R_1, R_2 \in N_R^{\pm}$.

Observe that the left-hand side of the constraints are negation-free; in DL-Lite$_\mathcal{R}$, negation can only appear in the right-hand side of a constraint. Constraints featuring negation allow expressing a particular form of *integrity constraints*: *disjointness* or *exclusion* constraints. The next example illustrates DL-Lite$_\mathcal{R}$ KBs.

Example 5 (DL-Lite$_\mathcal{R}$ KB). *Consider the DL-Lite$_\mathcal{R}$ TBox $\mathcal{T}$ in Table 2.2 expressing contraints on the* Researcher *and* PhDStudent *concepts, and the* worksWith *and* supervisedBy *roles. It states that PhD students are researchers* $(T1)$*, researchers work with researchers* $(T2)(T3)$*, working with someone is a symmetric relation* $(T4)$*, being supervised by someone implies working with her/him* $(T5)$*, only PhD students are supervised* $(T6)$ *and they cannot supervise someone* $(T7)$*.*

Now consider the ABox $\mathcal{A}$ below, for the same concepts and roles:

$(A1)$ worksWith$(Ioana, Francois)$
$(A2)$ supervisedBy$(Damian, Ioana)$
$(A3)$ supervisedBy$(Damian, Francois)$

It states that Ioana works with François $(A1)$*, Damian is supervised by both Ioana* $(A2)$ *and François* $(A3)$*.*

The semantics of inclusion constraints is defined, as customary, in terms of their FOL interpretations. Tables 2.3 and 2.4 provide the FOL and relational notations expressing these constraints equivalently.

A KB $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$ is *consistent* if the corresponding FOL theory, consisting of the $\mathcal{A}$ facts and of the FOL constraints corresponding to $\mathcal{T}$, has a model. In this case, we

DL constraint	**FOL constraint**	**Relational constraint (under Open World Assumption)**
$A \sqsubseteq A'$	$\forall x[A(x) \Rightarrow A'(x)]$	$A \subseteq A'$
$A \sqsubseteq \exists R$	$\forall x[A(x) \Rightarrow \exists y R(x,y)]$	$A \subseteq \Pi_1(R)$
$A \sqsubseteq \exists R^-$	$\forall x[A(x) \Rightarrow \exists y R(y,x)]$	$A \subseteq \Pi_2(R)$
$\exists R \sqsubseteq A$	$\forall x[\exists y R(x,y) \Rightarrow A(x)]$	$\Pi_1(R) \subseteq A$
$\exists R^- \sqsubseteq A$	$\forall x[\exists y R(y,x) \Rightarrow A(x)]$	$\Pi_2(R) \subseteq A$
$\exists R' \sqsubseteq \exists R$	$\forall x[\exists y R'(x,y) \Rightarrow \exists z R(x,z)]$	$\Pi_1(R') \subseteq \Pi_1(R)$
$\exists R' \sqsubseteq \exists R^-$	$\forall x[\exists y R'(x,y) \Rightarrow \exists z R(z,x)]$	$\Pi_1(R') \subseteq \Pi_2(R)$
$\exists R'^- \sqsubseteq \exists R$	$\forall x[\exists y R'(y,x) \Rightarrow \exists z R(x,z)]$	$\Pi_2(R') \subseteq \Pi_1(R)$
$\exists R'^- \sqsubseteq \exists R^-$	$\forall x[\exists y R'(y,x) \Rightarrow \exists z R(z,x)]$	$\Pi_2(R') \subseteq \Pi_2(R)$
$R \sqsubseteq R'^-$ or $R^- \sqsubseteq R'$	$\forall x,y[R(x,y) \Rightarrow R'(y,x)]$	$R \subseteq \Pi_{2,1}(R')$ or $\Pi_{2,1}(R) \subseteq R'$
$R \sqsubseteq R'$ or $R^- \sqsubseteq R'^-$	$\forall x,y[R(x,y) \Rightarrow R'(x,y)]$	$R \subseteq R'$ or $\Pi_{2,1}(R) \subseteq \Pi_{2,1}(R')$

Table 2.3: DL-Lite$_\mathcal{R}$ inclusion constraints *without* negation in FOL, and in relational notation; A, A' are concept names while R, R' are role names. For the relational notation, we use 1 to designate the first attribute of any atomic role, and 2 for the second.

DL constraint	**FOL constraint**	**Relational constraint (under Open World Assumption)**
$A \sqsubseteq \neg A'$	$\forall x[A(x) \Rightarrow \neg A'(x)]$	$A \cap A' \subseteq \bot$
$A \sqsubseteq \neg\exists R$	$\forall x[A(x) \Rightarrow \neg\exists y R(x,y)]$	$A \cap \Pi_1(R) \subseteq \bot$
$A \sqsubseteq \neg\exists R^-$	$\forall x[A(x) \Rightarrow \neg\exists y R(y,x)]$	$A \cap \Pi_2(R) \subseteq \bot$
$\exists R \sqsubseteq \neg A$	$\forall x[\exists y R(x,y) \Rightarrow \neg A(x)]$	$A \cap \Pi_1(R) \subseteq \bot$
$\exists R^- \sqsubseteq \neg A$	$\forall x[\exists y R(y,x) \Rightarrow \neg A(x)]$	$A \cap \Pi_2(R) \subseteq \bot$
$\exists R' \sqsubseteq \neg\exists R$	$\forall x[\exists y R'(x,y) \Rightarrow \neg\exists z R(x,z)]$	$\Pi_1(R') \cap \Pi_1(R) \subseteq \bot$
$\exists R' \sqsubseteq \neg\exists R^-$	$\forall x[\exists y R'(x,y) \Rightarrow \neg\exists z R(z,x)]$	$\Pi_1(R') \cap \Pi_2(R) \subseteq \bot$
$\exists R'^- \sqsubseteq \neg\exists R$	$\forall x[\exists y R'(y,x) \Rightarrow \neg\exists z R(x,z)]$	$\Pi_2(R') \cap \Pi_1(R) \subseteq \bot$
$\exists R'^- \sqsubseteq \neg\exists R^-$	$\forall x[R'(y,x) \Rightarrow \neg\exists z R(z,x)]$	$\Pi_2(R') \cap \Pi_2(R) \subseteq \bot$
$R \sqsubseteq \neg R'^-$ or $R^- \sqsubseteq \neg R'$	$\forall x,y[R(x,y) \Rightarrow \neg R'(y,x)]$	$R \cap \Pi_{2,1}(R') \subseteq \bot$ or $\Pi_{2,1}(R) \cap R' \subseteq \bot$
$R \sqsubseteq \neg R'$ or $R^- \sqsubseteq \neg R'^-$	$\forall x,y[R(x,y) \Rightarrow \neg R'(x,y)]$	$R \cap R' \subseteq \bot$ or $\Pi_{2,1}(R) \cap \Pi_{2,1}(R') \subseteq \bot$

Table 2.4: DL-Lite$_\mathcal{R}$ inclusion constraints *with* negation, i.e., disjointness constraints, in FOL and relational notation. $\bot$ denotes the empty relation.

say also that $\mathcal{A}$ is $\mathcal{T}$*-consistent*. *In the absence of negation, any KB is consistent*, as negation-free constraints merely lead to infering more facts. If some constraints feature negation, $\mathcal{K}$ is consistent iff none of its (explicit or inferred) facts contradicts a constraint with negation. An inclusion or assertion α is *entailed* by a KB $\mathcal{K}$, written $\mathcal{K} \models \alpha$, if α is satisfied in all the models of the FOL theory corresponding to $\mathcal{K}$.

Example 6 (DL-Lite$_\mathcal{R}$ entailment). *The KB $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$ from Example 5 entails many constraints and assertions. For instance:*

- $\mathcal{K} \models \exists\texttt{supervisedBy} \sqsubseteq \neg\exists\texttt{supervisedBy}^-$, *i.e., the two attributes of* supervisedBy *are disjoint, due to* $(T6) + (T7)$*;*
- $\mathcal{K} \models \texttt{worksWith}(Francois, Ioana)$, *i.e., François works with Ioana, due to* $(T4)+(A1)$*;*
- $\mathcal{K} \models \texttt{PhDStudent}(Damian)$, *i.e., Damian is a PhD student, due to* $(A2) + (T6)$*;*
- $\mathcal{K} \models \texttt{worksWith}(Francois, Damian)$, *i.e., François works with Damian, due to* $(A3) + (T5) + (T4)$.

Finally remark that $\mathcal{A}$ is $\mathcal{T}$-consistent, i.e., there is no violation of its only constraint using negation $(T7)$*, since the KB $\mathcal{K}$ does not entail that some PhD student supervises another.*

2.2.1 Queries

DL-Lite$_\mathcal{R}$ knowledge bases are queried with FOL queries. A FOL query is of the form $q(\bar{x})$:- $\phi(\bar{x})$ where $\phi(\bar{x})$ is a FOL formula whose free variables are $\bar{x}$; the query *name* is q, its *head* is $q(\bar{x})$, while its *body* is $\phi(\bar{x})$. The *answer set* of a query q against a knowledge base $\mathcal{K}$ is: $ans(q, \mathcal{K}) = \{\bar{t} \in (N_I)^n \mid \mathcal{K} \models \phi(\bar{t})\}$, where $\mathcal{K} \models \phi(\bar{t})$ means that every model of $\mathcal{K}$ is a model of $\phi(\bar{t})$. If q is Boolean, $ans(q, \mathcal{K}) = \{\langle\rangle\}$ encodes true, with $\langle\rangle$ the empty tuple, while $ans(q, \mathcal{K}) = \emptyset$ encodes false. In keeping with the literature on query answering under ontological constraints, *our queries have set semantics*.

Example 7 (Query answering). *Consider the* FOL *query q asking for the PhD students with whom someone works:*

$$q(x) \text{:- } \exists y\ \texttt{PhDStudent}(x) \wedge \texttt{worksWith}(y, x)$$

Given the KB $\mathcal{K}$ of Example 5, the answer set of this query is $\{Damian\}$, since $\mathcal{K} \models$ $\texttt{PhDStudent}(Damian)$ and $\mathcal{K} \models \texttt{worksWith}(Francois, Damian)$ hold. Observe that evaluating q against $\mathcal{K}$'s ABox only yields no answer.

CQ	$q(\bar{x})$:- $a_1 \wedge \cdots \wedge a_n$
SCQ	$q(\bar{x})$:- $(a_1^1 \vee \cdots \vee a_1^{k_1}) \wedge \cdots \wedge (a_n^1 \vee \cdots \vee a_n^{k_n})$
UCQ	$q(\bar{x})$:- $\texttt{CQ}_1(\bar{x}) \vee \cdots \vee \texttt{CQ}_n(\bar{x})$
USCQ	$q(\bar{x})$:- $\texttt{SCQ}_1(\bar{x}) \vee \cdots \vee \texttt{SCQ}_n(\bar{x})$
JUCQ	$q(\bar{x})$:- $\texttt{UCQ}_1(\bar{x_1}) \wedge \cdots \wedge \texttt{UCQ}_n(\bar{x_n})$
JUSCQ	$q(\bar{x})$:- $\texttt{USCQ}_1(\bar{x_1}) \wedge \cdots \wedge \texttt{USCQ}_n(\bar{x_n})$

Table 2.5: FOL query dialects.

To simplify the reading, in what follows, we omit the quantifiers of existential variables, and simply write the above query as $q(x)$:- $\texttt{PhDStudent}(x) \wedge \texttt{worksWith}(y, x)$.

Query dialects. We will need to refer to several FOL query dialects, whose general forms are schematized in Table 2.5. *Conjunctive Queries* (CQs), a.k.a. select-project-join queries, are conjunctions of *atoms*, where an atom is either $A(t)$ or $R(t, t')$, for some t, t' variables or constants. *Semi-Conjunctive Queries* (SCQs) are joins of unions of single-atom CQs with the same arity, where the atom is either of the form $A(t)$ or of the form $R(t, t')$ as above; the bound variables of SCQs are also existentially quantified. *Unions of* CQ*s* (UCQs) are disjunctions of CQs with same arity. *Unions of* SCQ*s* (USCQs), *Joins of* UCQ*s* (JUCQs), and finally *Joins of* USCQ*s* (JUSCQs) are built on top of simpler languages by adding unions, respectively joins. All the above dialects directly translate into SQL, and thus can be evaluated by an RDBMS.

Notations. Unless otherwise specified, we systematically use q to refer to a CQ query, $a_1, \ldots, a_n$ to designate the atoms in the body of q, $\mathcal{T}$ to designate a DL-Lite$_\mathcal{R}$ TBox, and $\mathcal{A}$ for an ABox.

2.2.2 Query answering techniques

In constrast to the RDF case, data saturation is not an option in DL-Lite$_\mathcal{R}$, as the saturation of a KB may be infinite. Indeed, think of a KB with TBox $\mathcal{T} = \{\exists R^- \sqsubseteq \exists R, A \sqsubseteq \exists R\}$ and ABox $\mathcal{A} = \{A(a)\}$. Clearly, the saturation of $\mathcal{A}$ w.r.t. the constraints in $\mathcal{T}$ is infinite: $\{A(a), R(a, w_0)\} \cup \bigcup_{i=0}^{\infty}\{R(w_i, w_{i+1})\}$ where the w_i's are existential variables. Settings in which query answering is said FOL-reducible have therefore been investigated, i.e., settings in which query reformulation is possible.

FOL-reducible query answering. In a setting where query answering is FOL-reducible, there exists a FOL query $q^{\texttt{FOL}}$ (computable from q and $\mathcal{T}$) such that:

$$ans(q, \langle \mathcal{T}, \mathcal{A} \rangle) = ans(q^{\texttt{FOL}}, \langle \emptyset, \mathcal{A} \rangle)$$

for any $\mathcal{T}$-consistent ABox $\mathcal{A}$. Thus, query answering reduces to: a first reasoning step to produce the FOL query from q and $\mathcal{T}$ (this is also known as *reformulating* the query using the constraints), and a second step which evaluates the *reformulated query* $q^{\texttt{FOL}}$ in the standard fashion, *only on the ABox* (i.e., disregarding the TBox constraints). This

$q^1(x)$:- $\texttt{PhDStudent}(x) \wedge \texttt{worksWith}(y, x)$
$q^2(x)$:- $\texttt{PhDStudent}(x) \wedge \texttt{worksWith}(x, y)$
$q^3(x)$:- $\texttt{PhDStudent}(x) \wedge \texttt{supervisedBy}(y, x)$
$q^4(x)$:- $\texttt{PhDStudent}(x) \wedge \texttt{supervisedBy}(x, y)$
$q^5(x)$:- $\texttt{supervisedBy}(x, z) \wedge \texttt{worksWith}(y, x)$
$q^6(x)$:- $\texttt{supervisedBy}(x, z) \wedge \texttt{worksWith}(x, y)$
$q^7(x)$:- $\texttt{supervisedBy}(x, z) \wedge \texttt{supervisedBy}(y, x)$
$q^8(x)$:- $\texttt{supervisedBy}(x, z) \wedge \texttt{supervisedBy}(x, y)$
$q^9(x)$:- $\texttt{supervisedBy}(x, x)$
$q^{10}(x)$:- $\texttt{supervisedBy}(x, y)$

Table 2.6: Union terms in CQ-to-UCQ reformulation (Example 8).

can be done for instance by translating it into SQL and delegating the evaluation to an RDBMS. From a knowledge base perspective, this allows to take advantage of highly optimized data stores and query evaluation engines to answer queries. From the database perspective, this two-step approach enhances the power of RDBMSs, as it allows to compute answers based only on data stored in the ABox (i.e., the database), but also taking into account the deductive constraints and all their consequences (entailed facts and constraints).

As DL-Lite$_\mathcal{R}$ query answering is FOL reducible [32], the literature provides techniques for computing FOL reformulations of a CQ in settings related to DL-Lite$_\mathcal{R}$. These techniques produce (i) a UCQ w.r.t. a DL-Lite$_\mathcal{R}$ TBox, e.g., [32, 2, 86, 36, 100], or extensions thereof using existential rules [67] or Datalog$^\pm$ [101, 51], (ii) a USCQ [96] w.r.t. a set of existential rules generalizing a DL-Lite$_\mathcal{R}$ TBox, and (iii) a set of alternative equivalent JUCQs w.r.t. an RDF database [48], whose RDF Schema constraints are the following four, out of the twenty-two, DL-Lite$_\mathcal{R}$ ones: (1) $A \sqsubseteq A'$, (4) $\exists R \sqsubseteq A$, (5) $\exists R^- \sqsubseteq A$ and (11) $R \sqsubseteq R'$.

CQ-to-UCQ reformulation for DL-Lite$_\mathcal{R}$ [32]. We present the pioneering CQ-to-UCQ technique on which we rely to establish our results. *These results extend to any other* FOL *reformulation techniques for DL-Lite$_\mathcal{R}$*, e.g., the above-mentioned ones, *since they produce equivalent* FOL *queries*.

The technique of [32] relies on two operations: *specializing a query atom into another* by applying a negation-free constraint (recall Table 2.3) in the backward direction, and *specializing two atoms into their most general unifier* (mgu, in short). These operations are exhaustively applied to the input CQ; each operation generates a new CQ *contained in* the input CQ w.r.t. the TBox, because the new CQ was obtained by specializing one or two atoms of the previous CQ. The same process is then applied on the new CQs, and so on recursively until the set of generated CQs reaches a fixpoint. The finite union of the input CQ and of the generated ones forms the UCQ reformulation of the input CQ w.r.t. the TBox.

Example 8 (CQ**-to-**UCQ **reformulation).** *Consider the query* $q(x)$*:-* $\texttt{PhDStudent}(x) \wedge \texttt{worksWith}(y, x)$ *and KB* $\mathcal{K}$ *of the preceding examples. The* UCQ *reformulation of* q *is:* $q^{\texttt{UCQ}}(x)$*:-* $\bigvee_{i=1}^{10} q^i(x)$ *where* q^1*-*q^{10} *appear in Table 2.6. In the table,* $q^1(x)$ *has exactly the body of* q*.* $q^2(x)$ *is obtained from* q^1 *by applying the constraint* $(T4)$*:* $\texttt{worksWith} \sqsubseteq \texttt{worksWith}^-$*, which is of the form (10) listed in Table 2.3.* $(T4)$ *is* applied backward*, in the following sense: the query asks for* $\texttt{worksWith}(y, x)$*, and* $(T4)$ *tells us that* one of the possible reasons why this may hold*, is if* $\texttt{worksWith}(x, y)$ *holds. Thus,* q^2 *is contained within* q^1*, in the sense that if* q^2 *holds,* q^1 *is also sure to hold, but the opposite is not true; intuitively, "*q^1 *may hold for other reasons (thanks to other specializations of its atoms)" - and it is exactly the set of such other specializations which the technique explores.*

Similarly, q^3 *is obtained from* q^1 *by applying the constraint* $(T5)$ *backward on the atom* $\texttt{worksWith}(y, x)$*, and* q^4 *from* q^2 *by applying* $(T5)$ *on* $\texttt{worksWith}(x, y)$*. To obtain* q^5 *to* q^8*, we apply* $(T6)$ *backward on the atom* $\texttt{PhDStudent}(x)$ *in* q^1 *to* q^4*. Finally,* q^9 *is obtained from* q^7 *through the mgu of its two atoms, namely* $\texttt{supervisedBy}(x, z)$ *and* $\texttt{supervisedBy}(y, x)$*;* q^{10} *is similarly obtained from* q^8*.*

Beyond FOL**-reducible query answering**. The so-called *combined approach* of [68] computes a finite approximation of a (possibly infinite) KB' saturation, and then reformulates queries so that erroneous answers introduced by approximating the saturation are not returned.

Chapter 3

Efficient query answering in the presence of RDFS constraints

Answering queries over data in the presence of *deductive constraints*, which lead to *implicit* data that is entailed (derived) from the explicit data and the constraints, requires a *reasoning* step in order to compute complete query answers. Two main query answering techniques exist: data *saturation* compiles the constraints into the database by making all implicit data explicit, while query *reformulation* compiles the constraints into a modified query, which, evaluated over the explicit data only, computes all the answers due to explicit and/or implicit data. So far, reformulation-based query answering has received significantly less attention than saturation. In particular, reformulated queries may be complex, thus their evaluation may be very challenging.

In this chapter we focus on optimizing reformulation-based query answering in the setting of *ontology-based data access*, where SPARQL conjunctive queries are answered against a set of RDF facts on which RDFS constraints hold.

We consider the setting in which *conjunctive queries (*`CQ`*), once reformulated into unions of conjunctive queries (*`UCQ`*) or semi-conjunctive queries (*`SCQ`*), are handled for evaluation to a query evaluation engine*, which can be an RDBMS, a dedicated RDF storage and query processing engine, or more generally any system capable of evaluating *selections*, *projections*, *joins* and *unions*. As our experiments show, the evaluation of reformulated queries may be very challenging even for well-established relational or native RDF processors, which may handle them inefficiently or simply fail to handle them, even on moderate-size datasets.

The approach we take is the following: given a SPARQL conjunctive query q and a query reformulation algorithm $\mathcal{A}$ which turns a `CQ` into a `UCQ`, we explore a novel, large *space of alternative reformulations* of q that we term `JUCQ` (for *joins of unions of conjunctive queries*, which captures the `UCQ` and `SCQ` reformulations, and from which we pick a `JUCQ` reformulation with lowest estimated cost. Each `JUCQ` reformulation is obtained based on a carefully chosen set of invocations of the algorithm $\mathcal{A}$, guided by our cost model.

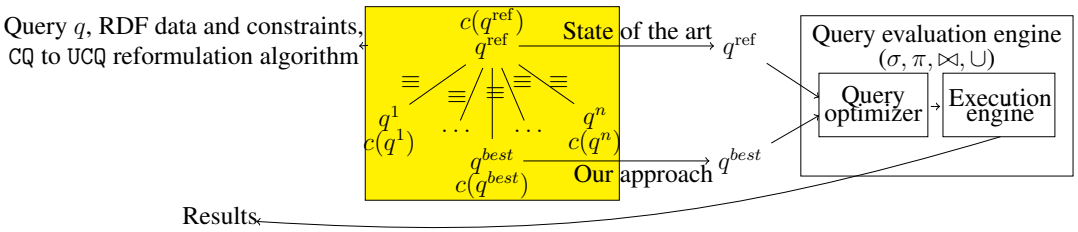

Figure 3.1: Outline of our approach for efficiently evaluating reformulated SPARQL conjunctive queries.

Contributions. The contributions we bring to the problem of efficiently answering SPARQL queries, through reformulation, can be outlined as follows (see Figure 3.1):

1. We generalize the query reformulation approach, by considering a large *space of alternative (equivalent)* JUCQ *reformulations*. This space corresponds to the yellow-background box in Figure 3.1; it includes and significantly generalizes the prior works based on UCQ or SCQ reformulation. We characterize the size of our space of alternatives, and show that it is oftentimes too large to be completely explored.

2. We define a *cost model* for estimating the evaluation performance of our reformulated queries through a relational engine; other functions can be used instead, and we show that an RDBMSs' internal cost model can easily be used, too.

3. We devise a novel *algorithm* which selects one alternative reformulated query, namely q^{best} in Figure 3.1, which (i) computes the same result as the UCQ or SCQ reformulated query q^{ref}, and (ii) reduces significantly the query evaluation cost (or simply makes it possible when evaluating q^{ref} fails!)

4. We implemented this algorithm and deployed it on top of three well-established RDBMSs, which we show differ significantly in their ability to handle UCQ and SCQ reformulations. Our experiments confirm that our algorithm *makes the most out of each of these engines* by leveraging their strengths and avoiding their weaknesses thanks to the usage of our cost model, which we calibrate separately for each system. This makes reformulation feasible when UCQ and/or SCQ fail, and brings performance improvements of several orders of magnitude w.r.t. UCQ .

5. Finally, we put our efficient reformulation-based query answering technique in perspective by comparing it against saturation-based query answering, both based on PostgreSQL and through the dedicated Semantic Web data management platform Virtuoso. These experiments confirm the robustness and performance of our technique, showing in particular that in some cases its performance approaches that of saturation-based query answering.

Triple	#answers	#reformulations	#answers after reformulation
(t_1)	$18,999,082$	188	$33,328,108$
(t_2)	0	4	$3,223$
(t_3)	396	3	683

Table 3.1: Characteristics of the sample query q_1.

In the sequel, we characterize our solution search space and formalize our problem in statementSection 3.1. Section 3.2 introduces our cost model and solution search space exploration technique, which we evaluate through experiments in Section 3.3. We discuss related work in Section 3.4, then we conclude.

The work reported here is based on the EDBT paper [25], the VLDB demonstration [26] and ICDE tutorial [29].

3.1 Motivation

We first introduce, by examples, the performance issues raised by the evaluation of state-of-the-art reformulated queries. We then introduce our novel reformulation search space and formalize our optimization problem.

Motivating Example 1. *Consider the three triples query q_1 shown below:*

$$\begin{array}{lll} q_1(x,y) \text{ :- } & x \text{ rdf:type } y, & (t_1) \\ & x \texttt{ ub:degreeFrom } \text{“}http://www.Univ532.edu\text{”}, & (t_2) \\ & x \texttt{ ub:memberOf } \text{“}http://www.Dept1.Univ7.edu\text{”} & (t_3) \end{array}$$

Table 3.1 gives some intuition on the difficulty of answering q_1 over an 10^8 triples LUBM [52] benchmark dataset.

The state-of-the-art CQ *to* UCQ *reformulation-based query answering needs to evaluate a reformulated query q'_1, which is a union of $2,256$ conjunctive queries, each of which consists of three triples (one for the reformulation of each triple in the original q_1). This query appears in Table 3.2, where all the triples t_1, t_2, t_3 are reformulated together by a* CQ *to* UCQ *reformulation algorithm denoted $(.)^{ref}$. Observe that in q'_1, many sub-expressions are repeated; for instance, the join over the single triples resulting from the reformulation of triples (t_2) and (t_3) will appear for each of the 188 reformulations of triple (t_1). Evaluating q'_1 on the 100 million triples LUBM dataset takes more than **6 seconds**.*

Alternatively, one could consider the equivalent query $q''_1 = (t_1)^{ref} \bowtie (t_2)^{ref} \bowtie (t_3)^{ref}$, which joins the CQ *to* UCQ *reformulation of each query's triple. In other terms, q''_1 first*

	Joins of UCQs	#reformulations	exec.time (ms)
q'_1	$(t_1, t_2, t_3)^{ref}$	$2,256$	$6,387$
q''_1	$(t_1)^{ref} \bowtie (t_2)^{ref} \bowtie (t_3)^{ref}$	195	$1,074,026$
	$(t_1, t_2)^{ref} \bowtie (t_3)^{ref}$	755	$1,968$
	$(t_1)^{ref} \bowtie (t_2, t_3)^{ref}$	200	$846,710$
q'''_1	$(t_1, t_3)^{ref} \bowtie (t_2)^{ref}$	568	554
	$(t_1, t_2)^{ref} \bowtie (t_1, t_3)^{ref}$	$1,316$	$2,734$
	$(t_1, t_2)^{ref} \bowtie (t_2, t_3)^{ref}$	764	$2,289$
	$(t_1, t_3)^{ref} \bowtie (t_2, t_3)^{ref}$	576	588

Table 3.2: Sample reformulations of q_1.

reformulates each triple (into, respectively, a union of 188*,* 4*, and* 3 *queries), and then joins these unions. This query corresponds to the simple semi-conjunctive queries (*SCQ*) alternative proposed in [96]. While this avoids the repeated work, its performance is much worse: in the same experimental setting, it takes about* ***1074 seconds*** *to evaluate.*

Let us now consider the following equivalent query $q'''_1 = (t_1, t_3)^{ref} \bowtie (t_2)^{ref}$ *where* t_1, t_2, t_3 *are the triples of the query* q_1*. Evaluating* q'''_1 *in the same experimental setting takes* ***554 ms****, more than* ***10 times faster*** *than the initial reformulation. The performance improvement of* q'''_1 *over* q''_1 *is due to the intelligent grouping of the triples* t_1 *and* t_3 *together. Such grouping of triples reduce the cardinality of the respective reformulated queries. Thus,* $(t_1, t_3)^{ref}$ *has* $2,045$ *answers and* 564 *reformulations. Table 3.2 shows the number of reformulations and execution time for all the eight possible combinations of triples.*

Motivating Example 2. *Consider now the six triples query* q_2 *shown below:*

$$
\begin{aligned}
q_2(x, u, y, v, z) \text{ :- } & x \text{ rdf:type } u, && (t_1)\\
& y \text{ rdf:type } v, && (t_2)\\
& x \text{ ub:}mastersDegreeFrom \text{ “}http://www.Univ532.edu\text{”}, && (t_3)\\
& y \text{ ub:}doctoralDegreeFrom \text{ “}http://www.Univ532.edu\text{”}, && (t_4)\\
& x \texttt{ub:memberOf } z && (t_5)\\
& y \texttt{ub:memberOf } z && (t_6)
\end{aligned}
$$

Statistics on the query triples, when evaluated over a 100 *million triples LUBM dataset, appear in Table 3.3. The* CQ *to* UCQ *reformulation of* q_2*, on the other hand, leads to a query* q'_2 *corresponding to a union of* $318,096$ *six triples queries. Due to its complexity,* q'_2 ***could not be evaluated*** *in the same experimental setting*[1].

[1]Concretely, a *stack depth limit exceeded error* was thrown by the DBMS. Further, other queries presented I/O exceptions thrown by the DBMS, in connection with a failed attempt to materialize an intermediary result. While it may be possible to tune some parameters to make the evaluation of such queries possible, the same error was raised by many large-reformulation queries, a signal that their peculiar shape is problematic.

Triple	#answers	#reformulations	#answers after reformulation
(t_1)	$18,999,082$	188	$33,328,108$
(t_2)	$18,999,082$	188	$33,328,108$
(t_3)	476	1	476
(t_4)	509	1	509
(t_5)	$7,299,701$	3	$7,803,096$
(t_6)	$7,299,701$	3	$7,803,096$

Table 3.3: Characteristics of the sample query q_2.

Now consider the query $q_2'' = (t_1)^{ref} \bowtie (t_2)^{ref} \bowtie (t_3)^{ref} \bowtie (t_4)^{ref} \bowtie (t_5)^{ref} \bowtie (t_6)^{ref}$*, where* $t_1, \ldots, t_6$ *are the triples of* q_2*; again, this corresponds to the* SCQ *reformulation proposed in [96].* q_2'' *is equivalent to* q_2^{ref}*, and in the same experimental setting, it is evaluated in* ***229 seconds****. This is due to the large results of the (syntactically small) subqueries* $(t_1)^{ref}, \ldots, (t_6)^{ref}$ *(especially the first two, each with* $33,328,108$ *results), which required some time to join.*

Finally, consider the query $q_2''' = (t_1, t_3)^{ref} \bowtie (t_3, t_5)^{ref} \bowtie (t_2, t_4)^{ref} \bowtie (t_4, t_6)^{ref}$*, also equivalent to* q_2'*. Evaluating* q_2''' *takes* ***524 ms****, more than* ***430 times faster*** *than* SCQ *reformulation. As in the previous example,* q_2''' *gains over* q_2'' *by first, reducing repeated work, and second, intelligently grouping triples so that the query corresponding to each triple group can be efficiently evaluated and returns a result of manageable size. In particular, the biggest-size triples* (t_1) *and* (t_2) *had been grouped with* (t_3) *and* (t_4) *respectively, resulting in smaller intermediate results of* $2,296$ *and* $2,475$ *rows respectively, and improving the perfomance. Grouping triples* (t_3) *and* (t_4) *with the* (t_5) *and* (t_6) *respectively, yields analogous performance improvements.*

As the above examples illustrate, generalizing the state-of-the-art query reformulation language of UCQs [48, 98, 62, 99, 46, 5, 49, 32, 50] or of SCQs [96], to that of *joins* of UCQs, offers a great potential for improving the performance of reformulated queries. We introduce:

Definition 3.1.1 (JUCQ)**.** *A* Join of Unions of Conjunctive Queries (JUCQ) *is defined as follows:*

- *any conjunctive query (*CQ*) is a* JUCQ*;*
- *any union of* CQ*s (*UCQ*) is a* JUCQ*;*
- *any join of* UCQ*s is* JUCQ*.*

Next, we address the challenge of finding a best-performing JUCQ *reformulation* of a BGP query against an RDF database, among those that can be derived from a *query cover*. We define these notions as follows:

Definition 3.1.2 (JUCQ reformulation)**.** *A* JUCQ reformulation $q^{\texttt{JUCQ}}$ *of a BGP query* q *w.r.t. a database* $\texttt{db}_1$ *is a* JUCQ *such that* $q^{\texttt{JUCQ}}(\texttt{db}_2) = q(\texttt{db}_2^{\infty})$*, for any RDF database* $\texttt{db}_2$ *having the same schema as* $\texttt{db}_1$*.*

Recall that two RDF databases have the same schema iff their saturations have the same RDFS statements.

BGP query covering is a technique we introduce for exploring a space of JUCQ reformulations of a given query. The idea is to *cover* a query q with (possibly overlapping) subqueries, so as to produce a JUCQ reformulation of q by joining the (state-of-the-art) CQ to UCQ reformulations of these subqueries, obtained through any reformulation algorithm in the literature (e.g., [48]). Formally:

Definition 3.1.3 (BGP query cover)**.** *A* cover *of a BGP query* $q(\bar{x})$*:-* $t_1, \ldots, t_n$ *is a set* $C = \{f_1, \ldots, f_m\}$ *of non-empty subsets of* q*'s triples, called* fragments*, such that* $\bigcup_{i=1}^{m} f_i = \{t_1, \ldots, t_n\}$*, no fragment is included into another, i.e.,* $f_i \not\subseteq f_j$ *for* $1 \leq i, j \leq m$ *and* $i \neq j$*, and: if* C *consists of more than 1 fragment, then any fragment joins at least with another, i.e., they share a variable.*

For example, a cover of our query q_1 is $\{\{t_1, t_2\}, \{t_2, t_3\}\}$.

Definition 3.1.4 (Non-redundant BGP query cover)**.** *A* non-redundant cover *is a cover* C *such that there is no fragment* $f_k \in C$ *such that* $\bigcup_{i=1}^{m} f_i = \bigcup_{i=1, i\neq k}^{m} f_i$*.*

It can be easily shown that the reformulation based on a redundant cover C_r performs at least as much work as the reformulation based on a non-redundant cover C derived from C_r by removing one or more fragments until C is non-redundant. Therefore, in the sequel, we will only work with non-redundant covers; accordingly, we will explain how to ensure covers are and stay non-redudant, when presenting cover search algorithms.

Definition 3.1.5 (Cover queries of a BGP query)**.** *Let* $q(\bar{x})$*:-* $t_1, \ldots, t_n$ *be a BGP query and* $C = \{f_1, \ldots, f_m\}$ *one of its covers. A* cover query $q_{|f_i, 1\leq i\leq m}$ *of* q *w.r.t.* C *is the subquery whose body consists of the triples in* f_i *and whose head variables are the distinguished variables* $\bar{x}$ *of* q *appearing in the triples of* f_i*,* plus *the variables appearing in a triple of* f_i *that are shared with some triple of another fragment* $f_{j, 1\leq j\leq m, j\neq i}$*, i.e., on which the two fragments join.*

For example, the cover $\{\{t_1\}, \{t_2, t_3\}\}$ of our query q_1 leads to the cover queries:

$q_{|f_1}(x, y)$:- x rdf:type y,

and

$q_{|f_2}(x)$:- x ub:degreeFrom "$http://www.Univ532.edu$", x ub:memberOf "$http://www.Dept1.Univ7.edu$".

Query evaluation through an RDBMS is typically much more efficient when all the atoms of the query are connected through joins (in which case, properly optimized

queries oftentimes run in linear time in the size of the data), than when the query comprises a cartesian product (which leads to unavoidable quadratic or higher complexity in the size of the data). Therefore, in this work, we only consider fragments which do not feature a cartesian product.

The theorem below states that evaluating a query q as the join of the cover queries resulting from one of its covers, yields the answer set of q:

Theorem 3.1.1 (Cover-based reformulation)**.** *Let $q(\bar{x})$:- $t_1, \ldots, t_n$ be a BGP query and $C = \{f_1, \ldots, f_m\}$ be any of its covers,*

$$q^{\texttt{JUCQ}}(\bar{x})\text{:-}\ q^{\texttt{UCQ}}_{|f_1} \bowtie \cdots \bowtie q^{\texttt{UCQ}}_{|f_m}$$

is a JUCQ *reformulation of q w.r.t. any database* db*, where every $q^{\texttt{UCQ}}_{|f_i}$ is a* UCQ *reformulation of the cover query $q_{|f_i}$, for $1 \leq i \leq m$.*

Proof. The proof of Theorem 3.1.1 follows directly from the fact that any cover query $q_{|f_i}$, which is a CQ, can be equivalently reformulated w.r.t. db into a UCQ $q^{\texttt{UCQ}}_{|f_i}$, e.g., using any state-of-the-art CQ to UCQ reformulation algorithm.

For any RDF graph G, the answer set to a BGP query $q(\bar{x})$:- $t_1, \ldots, t_n$ is $q(G^\infty)$, where $q(G^\infty)$ is the relational evaluation of q against G^∞ [48].

Let $C = \{f_1, \ldots, f_m\}$ be a cover for q, $q'(\bar{x})$:- $q_{|f_1} \bowtie \cdots \bowtie q_{|f_m}$ is by definition a join decomposition of q. Therefore $q(G^\infty) = q'(G^\infty)$, hence q and q' are equivalent.

We now want to show that q' and $q^{\texttt{JUCQ}}$ are equivalent, and therefore q and $q^{\texttt{JUCQ}}$ also. Observe that for each fragment query $q_{|f_i}$ in q' we have $q_{|f_i}(G^\infty) = q^{\texttt{UCQ}}_{|f_i}(G)$, thus $q_{|f_i}$ and $q^{\texttt{UCQ}}_{|f_i}$ are equivalent, by the correctness of UCQ reformulation algorithms. Thus, replacing the fragment queries in q' by their corresponding equivalent reformulations we obtain $q^{\texttt{JUCQ}}$:

$$q^{\texttt{JUCQ}}(\bar{x})\text{:-}\ q^{\texttt{UCQ}}_{|f_1} \bowtie \cdots \bowtie q^{\texttt{UCQ}}_{|f_m}$$

Threrefore, $q^{\texttt{JUCQ}}$ is equivalent to q. □

An *upper bound* on the size of the cover-based reformulation space for a given query of n triples is given by the number of *minimal covers* of a set $\mathcal{S}$ of n elements [55], i.e., a set of non-empty subsets of $\mathcal{S}$ whose union is $\mathcal{S}$, and whose union of all these subsets but one is not $\mathcal{S}$. This bound grows rapidly as the number n of triples in a query's body increases, e.g., 1 for $n = 1$, 49 for $n = 4$, 462 for $n = 5$, 6424 for $n = 6$ (`http://oeis.org/A046165`). In practice, however, we require each fragment to share a variable with another (if any), so that cover queries, hence *cover-based reformulations do not feature cartesian products*. Therefore, the number of cover-based reformulations is smaller than the number of minimal covers.

In order to select the best performing cover-based reformulation within the above space, we assume given a *cost function* c which, for a JUCQ q, returns the cost $c(q(\texttt{db}))$ of

evaluating it through an RDBMS storing the database db. Function c may reflect any (combination of) query evaluation costs, such as I/O, CPU etc. As customary, we rely on a *cost estimation* function c^e, which statically provides an approximate value of c. For simplicity, in the sequel we will use c to denote the estimated cost.

The problem we study can now be stated as follows:

Definition 3.1.6 (Optimization problem)**.** *Let* db *be an RDF database and q be a BGP query against it. The* optimization problem *we consider is to find a* JUCQ *reformulation $q^{\texttt{JUCQ}}$ of q w.r.t.* db*, among the cover-based reformulations of q with lowest (estimated) cost.*

Optimized queries vs. optimized plans. As stated above and illustrated in Figure 3.1, we seek a best *query* that is an optimized reformulation of q against db; we do not seek to optimize its *plan*, instead, we take advantage of existing query evaluation engines for optimizing and executing it. Alternatively, one could have placed this study *within an evaluation engine* and investigate *optimized plans*. We comment more the two alternatives in Section 3.4.

3.2 Efficient query answering

We present now the ingredients for setting up our cost-based query answering technique. Section 3.2.1 introduces our cost model for JUCQ reformulation evaluation through an RDBMS. We then provide, in Section 3.2.2, an exhaustive algorithm that traverses the search space of reformulated queries, looking for a cover-based reformulation with lower cost. Finally, in Section 3.2.3, we introduce a greedy, anytime algorithm that outputs a best query cover of the input BGP query, found so far. These algorithms are then used to evaluate the query as stated by Theorem 3.1.1.

3.2.1 Cost model

In this section we detail the cost of evaluating a JUCQ (reformulation) sent to an RDBMS. Such a query is a join of UCQs subqueries of the form: $q^{\texttt{JUCQ}}(\bar{x})$:- $q_1^{\texttt{UCQ}} \bowtie \cdots \bowtie q_m^{\texttt{UCQ}}$.

The evaluation cost of $q^{\texttt{JUCQ}}$ is

$$c(q^{\texttt{JUCQ}}) = c_{\texttt{db}} + \sum_{q_i^{\texttt{UCQ}} \in q^{\texttt{JUCQ}}} (c_{eval}(q_i^{\texttt{UCQ}}) + c_{join}(q_{i,1\leq i\leq m}^{\texttt{UCQ}}) + c_{mat}(q_{i,1\leq i\leq m, i\neq k}^{\texttt{UCQ}})) + c_{unique}(q^{\texttt{JUCQ}}) \quad (3.1)$$

reflecting:

(i) the fixed overhead of connecting to the RDBMS $c_{\mathtt{db}}$;

(ii) the cost to *evaluate* each of its UCQ sub-queries $q_i^{\mathtt{UCQ}}$;

(iii) the cost of eliminating duplicate rows from each of its UCQ sub-query results;

(iv) the cost to *join* these sub-query results;

(v) the *materialization* costs: the SQL query corresponding to a JUCQ may have many sub-queries. At execution time, some of these subqueries will have their results materialized (i.e., stored in memory or on disk) while at most one sub-query will be executed in pipeline mode. We assume without loss of generality, that the largest-result sub-query, denoted $q_k^{\mathtt{UCQ}}$, is the one pipelined (this assumption has been validated by our experiments so far); and

(vi) the cost of eliminating duplicate rows from the result.

In the above, duplicates are eliminated because existing reformulation algorithms (and accordingly, our work) operate under *set semantics*.

Notations. For a given query q over a database db, we denote by $|q|_t$ the estimated number of tuples in q's answer set. Recall that $q_{|\{t_i\}}$ stands for the restriction of q to its i-th triple. Using the notations above, the number of tuples in the answer set of $q_{|\{t_i\}}$ is denoted $|q_{|\{t_i\}}|_t$.

Duplicate elimination costs. Assuming duplicate elimination is implemented by hashing, we estimate the cost of eliminating duplicate rows from $q^{\mathtt{JUCQ}}$ (and $q^{\mathtt{UCQ}}$) as:

$$c_{unique}(q^{\mathtt{JUCQ}}) = c_l \times |q^{\mathtt{JUCQ}}|_t$$

where c_l is the CPU and I/O effort involved in sorting the results.

When the results are large enough that disk merge sort is needed, we estimate the cost of eliminating duplicate rows from $q^{\mathtt{JUCQ}}$ (and $q^{\mathtt{UCQ}}$ as a particular case) result as:

$$c_{unique}(q^{\mathtt{JUCQ}}) = c_k \times |q^{\mathtt{JUCQ}}|_t \times \log |q^{\mathtt{JUCQ}}|_t$$

where c_k is the CPU and I/O effort involved in (disk-based) sorting the results.

UCQ **evaluation costs** are estimated by summing up the estimated costs of the CQs:

$$c_{eval}(q_i^{\mathtt{UCQ}}) = c_{unique}(q_i^{\mathtt{UCQ}}) + \sum_{q^{\mathtt{CQ}} \in q_i^{\mathtt{UCQ}}} c_{eval}(q^{\mathtt{CQ}})$$

The cost of evaluating *one* conjunctive query $c_{eval}(q^{\mathtt{CQ}})$, where $q^{\mathtt{CQ}}(\bar{x})$:- $t_1, \ldots, t_n$, through the RDBMS is made of the *scan* cost for retrieving the tuples for each of its triples, and the cost of *joining* these tuples:

$$c_{eval}(q^{\texttt{CQ}}) = c_{scan}(q^{\texttt{CQ}}) + c_{join}(q^{\texttt{CQ}})$$

We estimate the *scan cost* of $q^{\texttt{CQ}}$ to:

$$c_{scan}(q^{\texttt{CQ}}) = c_t \times \sum_{t_i \in q^{\texttt{CQ}}} |q^{\texttt{CQ}}_{|\{t_i\}}|_t$$

where c_t is the fixed cost of retrieving one tuple.

The *join* cost of $q^{\texttt{CQ}}$ represents the respective CPU and I/O effort; assuming efficient join algorithms such as hash- or merge-based etc. are available [88], this cost is linear in the total size of its inputs:

$$c_{join}(q^{\texttt{CQ}}) = c_j \times \sum_{t_i \in q^{\texttt{CQ}}} |q^{\texttt{CQ}}_{|\{t_i\}}|_t$$

Therefore, we have:

$$c_{eval}(q_i^{\texttt{UCQ}}) = (c_t + c_j) \times \sum_{q^{\texttt{CQ}} \in q_i^{\texttt{UCQ}}} \sum_{t_i \in q^{\texttt{CQ}}} |q^{\texttt{CQ}}_{|\{t_i\}}|_t \tag{3.2}$$

`UCQ` **join cost**. As before, we consider the join cost to be linear in the total size of its inputs:

$$c_{join}(q^{\texttt{UCQ}}_{i,1\leq i\leq m}) = c_j \times \sum_{q_i^{\texttt{UCQ}}} \sum_{q^{\texttt{CQ}} \in q_i^{\texttt{UCQ}}} \sum_{t_i \in q^{\texttt{CQ}}} |q^{\texttt{CQ}}_{|\{t_i\}}|_t \tag{3.3}$$

`UCQ` **materialization cost**. Finally, we consider the materialization cost associated to a query q is $c_m \times |q|_t$ for some constant c_m:

$$c_{mat}(q^{\texttt{UCQ}}_{i,1\leq i\leq m, i\neq k}) = c_m \times \sum_{q_i^{\texttt{UCQ}}, i\neq k} \sum_{q^{\texttt{CQ}} \in q_i^{\texttt{UCQ}}} \sum_{t_i \in q^{\texttt{CQ}}} |q^{\texttt{CQ}}_{|\{t_i\}}|_t \tag{3.4}$$

where $q_k^{\texttt{UCQ}}$ is the largest-result sub-query, and the one which is picked for pipelining (and thus not materialized).

Injecting the equations 3.2, 3.3 and 3.4 into the global cost formula 3.1 leads to the estimated cost of a given `JUCQ`. This formula relies on estimated cardinalities of various subqueries of the `JUCQ`, as well as on the system-dependent constants $c_{\texttt{db}}$, c_{scan}, c_{join} and c_{mat}, which we determine by running a set of simple calibration queries on the RDBMS being used. The details are straightforward and we omit them here.

3.2.2 Exhaustive query cover algorithm (ECov)

As a yardstick for the *quality* of the query covers we find, we developed an exhaustive query cover finding algorithm, called ECov, that traverses the search space of reformulated queries and outputs a query cover leading to a cover-based reformulation with lowest cost.

Given a BGP query q and a database db, ECov enumerates all the possible query covers, estimates the cost of the corresponding cover-based reformulations, and returns a query cover with the lowest estimated cost.

Possible moves based on the initial cover C_0 are developed and added to the list $moves$. Next (line 7), ECov starts exploring possible moves. It picks one from the $moves$ list and applies it, leading to a new query cover C', and possible moves based on C' are developed and added to the sorted $moves$ list (lines 10–12). If its estimated cost is smaller than the best (least) cost encountered so far, the best solution is updated to reflect this C' (lines 13– 14). Note that when the application of a move to a cover (line 9) leads to a redundant cover, all non-redundant covers that can be extracted from it, should be enumerated.

We use this cover as "golden standard", i.e., the best solutions based on our cost estimation function.

3.2.3 Greedy query cover algorithm (GCov)

We now present our optimized query cover finding algorithm, named GCov. Intuitively, GCov attempts to identify query covers such that the estimated evaluation cost of each cover fragment (once reformulated), together with the estimated cost of joining the results of these reformulated fragments, is minimized. Performance benefits in this context are attained from two sources: (i) avoiding the explosion in the size of the reformulated queries that results when many triples, each having many reformulations, are in the same fragment, and (ii) avoiding reformulated fragments with very large results, since materialising and joining them is costly. The key intuition for reaching these goals is to *include highly selective, few-reformulations triples in several cover fragments*. Observe that this is different from (and orthogonal to) join ordering, which the underlying query evaluation engine (RDBMS in this study) applies independently to each reformulated subquery.

GCov (Algorithm 2) starts with a simple cover C_0 consisting of *one triple fragments* (i.e., the SCQ reformulation), and explores possible *moves* starting from this state. A *move* consists of adding to one fragment, an extra triple connected to it by at least one join variable, such that the estimated cost associated to the cover-based reformulation thus obtained is smaller than that before the addition. Whenever *apply* (lines 6 and 9) leads to a redundant cover, a non-redundant cover is extracted out of the redundant one, in particular, the one with the least estimated cost. A move may reduce the cost in two

Algorithm 1: Naïve cover algorithm (**ECov**)

Input : BGP query $q(\bar{x}\text{:-}\ t_1, \ldots, t_n)$, database db
Output: Cover C for the BGP query q

1 $C_0 \leftarrow C = \{\{t_1\}, \{t_2\}, \ldots, \{t_n\}\}$;
2 $T \leftarrow C = \{t_1, t_2, \ldots, t_n\}$;
3 $C_{best} \leftarrow C_0$; $moves \leftarrow \emptyset$; $analysed \leftarrow \emptyset$;
4 **foreach** $f \in C_0, t \in T\ s.t.\ t \notin f \wedge connected(f,t) \wedge C_0.add(f,t) \notin analysed$ **do**
5 $analysed \leftarrow analysed \cup C_0.add(f,t)$;
6 $moves \leftarrow moves \cup (C_0, f, t)$;
7 **while** $moves \neq \emptyset$ **do**
8 $(C, f, t) \leftarrow moves.head()$;
9 $C' \leftarrow C.add(f,t)$;
10 **foreach** $f \in C', t \in T\ s.t.\ t \notin f \wedge connected(f,t) \wedge C'.add(f,t) \notin analysed$ **do**
11 $analysed \leftarrow analysed \cup C'.add(f,t)$;
12 $moves \leftarrow moves \cup (C', f, t)$;
13 **if** *C' estimated cost* $<$ *C_{best} estimated cost* **then**
14 $C_{best} \leftarrow C'$;
15 **return** C_{best};

ways: (i) by making a fragment more selective, and/or (ii) by leading to the removal of some fragments from the cover. For instance, let $\{\{t_1, t_2\}, \{t_1, t_3\}, \{t_3, t_4\}\}$ be a cover of a four-triples query. The move which adds t_4 to the first fragment, also renders $\{t_3, t_4\}$ redundant. Thus, the cover resulting from the move is: $\{\{t_1, t_2, t_4\}, \{t_1, t_3\}\}$. Concretely, all the fragments of a cover are kept sorted in the decreasing order of their cost. Whenever the cover is updated, we check the fragments from the first to the last; when a fragment is found redundant (with respect to the other fragments in the cover), the fragment is removed.

Possible moves based on the initial cover C_0 are developed (i.e., all possible covers that result from the possible moves are created, and the estimate cost of executing its associated cover-based reformulation on top of an RDBMS is computed) and the one leading to the cover with lower estimated cost is assigned to $move$. Next (line 8), GCov starts examining possible moves. It applies $move$, if any, leading to a new query cover C_{best}, updates the best solution to reflect this, and explores possible moves based on C_{best}; if a move resulting in a cover whose estimated cost is smaller than the best (least) cost encountered so far is found, then $move$ is set with it.

GCov explores query covers in breadth-first and *greedy* fashion, adding to the $moves$ list the possible moves starting from the current best cover, and selecting the next move

Algorithm 2: Greedy query cover algorithm (**GCov**)

Input : BGP query $q(\bar{x}\text{:-}\ t_1, \ldots, t_n)$, database db
Output: Cover C_{best} for the BGP query q

$C_0 \leftarrow \{\{t_1\}, \{t_2\}, \ldots, \{t_n\}\}$;
$T \leftarrow \{t_1, t_2, \ldots, t_n\}$;
$C_{best} \leftarrow C_0$; $move \leftarrow \emptyset$; $analysed \leftarrow \emptyset$;
foreach $f \in C_0, t \in T\ s.t.\ t \notin f \wedge connected(f, \{t\}) \wedge C_0.add(f,t) \notin analysed$ **do**
    $analysed \leftarrow analysed \cup C_0.add(f,t)$;
    if *($move$ is empty and $C_0.add(f,t)$ estimated cost $\leq C_{best}$ estimated cost) or ($C_0.add(f,t)$ estimated cost $< apply(move)$ estimated cost)* **then**
        $move \leftarrow (C_0, f, t)$;
while $move \neq \emptyset$ **do**
    $C_{best} \leftarrow apply(move)$;
    $move \leftarrow \emptyset$;
    foreach $f \in C_{best}, t \in T\ s.t.\ t \notin f \wedge connected(f, \{t\}) \wedge C_{best}.add(f,t) \notin analysed$ **do**
        $analysed \leftarrow analysed \cup C_{best}.add(f,t)$;
        if *($move$ is empty and $C_{best}.add(f,t)$ estimated cost $< C_{best}$ estimated cost) or ($C_{best}.add(f,t)$ estimated cost $< apply(move)$ estimated cost)* **then**
            $move \leftarrow (C_{best}, f, t)$;
return C_{best};

with smallest cost. In practice, one could easily change the stop condition, for instance to return the best found cover as soon as its cost has diminished by a certain ratio, or after a time-out period has elapsed etc.

3.3 Experimental evaluation

We now present an experimental assessment of our approach. Section 3.3.1 describes the experimental settings. Section 3.3.2 studies the effectiveness and efficiency of our optimized reformulation-based query answering technique. Section 3.3.3 widens our comparison to saturation-based query answering, then we conclude.

LUBM q	Q_{01}	Q_{02}	Q_{03}	Q_{04}	Q_{05}	Q_{06}	Q_{07}	Q_{08}	Q_{09}	Q_{10}	Q_{11}	Q_{12}	Q_{13}	Q_{14}	Q_{15}
$\|q^{ref}\|$	136	136	34	564	2	188	156	12	8,496	13	1	1	2	376	3,384
$\|q(\texttt{db})\|$ (1M)	123	123	41	26,048	982	5,537	0	269	0	47,268	1,530	88	4,041	20,205	0
$\|q(\texttt{db})\|$ (100M)	123	123	41	2,432,964	92,026	523,319	0	269	0	4,409,039	142,337	7,773	376,792	1,883,960	0

LUBM q	Q_{16}	Q_{17}	Q_{18}	Q_{19}	Q_{20}	Q_{21}	Q_{22}	Q_{23}	Q_{24}	Q_{25}	Q_{26}	Q_{27}	Q_{28}
$\|q^{ref}\|$	2	1	940	2,444	4	1	1	752	52	156	2,256	156	318,096
$\|q(\texttt{db})\|$ (1M)	5,364	5,388	47,348	60,342	228,086	60,342	16,134	100	12	19	5	1	0
$\|q(\texttt{db})\|$ (100M)	501,063	503,395	4,425,553	5,632,454	2,128,9440	5,632,454	1,510,695	11,820	1,508	1,463	5	1	495

DBLP q	Q_{01}	Q_{02}	Q_{03}	Q_{04}	Q_{05}	Q_{06}	Q_{07}	Q_{08}	Q_{09}	Q_{10}
$\|q^{ref}\|$	684	292	1,387	1,387	4	19	19	1,721	361	1,923,349
$\|q(\texttt{db})\|$	4,898	16,424	5,259,462	60,900	19,576	9,562	9,562	203,462	20	80

Table 3.4: Characteristics of the queries used in our study.

3.3.1 Settings

Software. We implemented our reformulation-based query answering framework in Java 7, on top of three well-known RDBMSs, namely: PostgreSQL v9.3.2, IBM DB2 Express-C v10.5, and MySQL Community Server v5.6.20. For each RDBMS, we instantiated the cost formulas introduced in Section 3.2.1 with the proper coefficients, learned by running our calibration queries on that system.

Hardware. All the RDMBSs run on 8-core Intel Xeon (E5506) 2.13 GHz machines with 16GB RAM, using Mandriva Linux release 2010.0 (Official).

Datasets. We conducted experiments using DBLP (8 million triples) [38] and LUBM [52] with 1 and 100 millions triples.

In our experiments, RDFS constraints are kept in memory, while RDF facts are stored in a $\texttt{Triples}(\texttt{s}, \texttt{p}, \texttt{o})$ table, indexed by all permutations of the $\texttt{s}, \texttt{p}, \texttt{o}$ columns, leading a total of 6 indexes. Our indexing choice is inspired by [83, 107], to give the RDBMS efficient query evaluation opportunities. Further, as in [48, 83, 107, 49], for efficiency, the $\texttt{Triples}(\texttt{s}, \texttt{p}, \texttt{o})$ table's data are dictionary-encoded, using a unique integer for each distinct value (URIs and literals). The dictionary is stored as a separate table, indexed both by the code and by the encoded value.

Queries. We used 28 and 10 BGP queries for our evaluation on LUBM and DBLP datasets, respectively. The LUBM queries appear in Section A.1, Tables A.1– A.3 and the DBLP queries in Table A.4, while their main characteristics (number of union terms in their UCQ reformulation, denoted $|q^{ref}|$, as well as the number of results when evaluated on our datasets) are shown in Table 3.4.

Some queries are modified versions of LUBM benchmark queries, in order to remove redundant triples[2]. We designed the others so that (i) they have an intuitive meaning, (ii) they exhibit a variety of result cardinalities, (iii) they exhibit a variety of reformu-

[2]A query triple is redundant when it can be inferred from the others based on the RDFS constraints. For instance, when looking for x such that x is a person and x has a social security number, if we know that only people have such numbers, the triple "x is a person" is redundant.

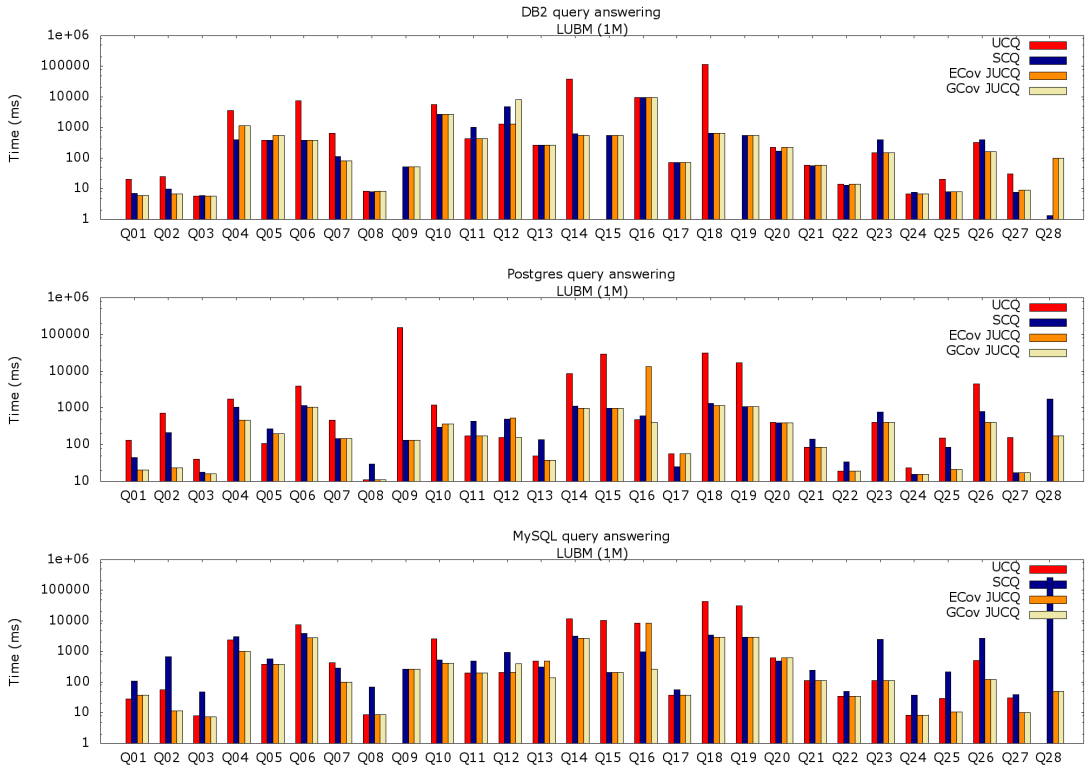

Figure 3.2: LUBM 1 million triples query answering through UCQ, SCQ, ECov and GCov JUCQ reformulations, against DB2, Postgres and MySQL.

lations, some of which are syntactically complex, to allow a study of the performance issues involved and (iv) none of their triples is redundant.

All measured times are averaged over 3 (warm) executions. Moreover, queries whose evaluation requires *more than 2 hours* were interrupted; we point them out when commenting on the experiments' results.

3.3.2 Optimized reformulation

In this section, we compare our reformulation-based query answering technique with those from the literature based on UCQs and SCQs.

Effectiveness: is an optimizer needed? The first question we ask is whether exploring the space of JUCQ alternatives is actually needed, or could one just rely on a simple (fixed) query cover?

The UCQ reformulation used in many prior works is a particular case of the JUCQ reformulations we introduced in this work; it corresponds to a cover of a single fragment

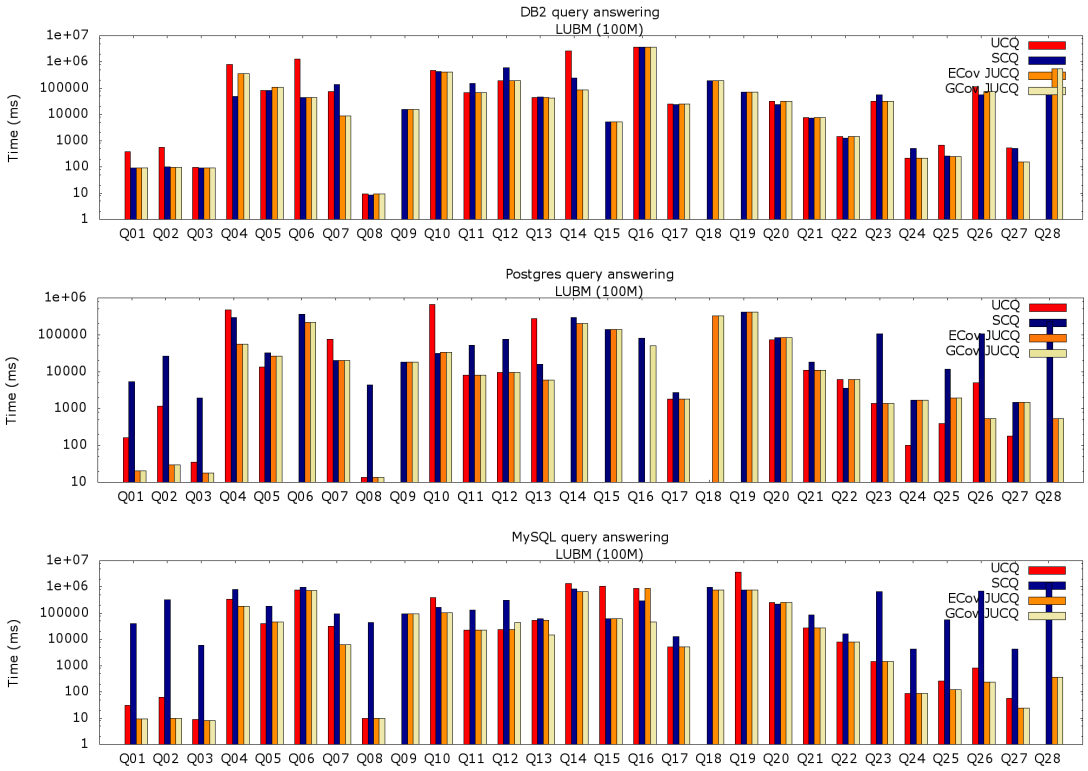

Figure 3.3: LUBM 100 million triples query answering through UCQ, SCQ, ECov and GCov JUCQ reformulations, against DB2, Postgres and MySQL.

made of all the query triples (recall q'_1 in **Motivating Example 1**, Section 3.1). From a database perspective, it corresponds to *pushing the joins below a single (potentially large) union*. At the other extreme, the SCQ reformulation proposed in [96] is a particular case of JUCQ reformulation obtained from a cover where each query triple is alone in a fragment (recall q''_1 in the same example). The SCQ reformulation can thus be thought of as *pushing all unions below a the joins*. Both the UCQ and SCQ reformulations correspond to a cover where *each triple appears in exactly one fragment*, whereas our JUCQs do not have this constraint; further, the UCQ and SCQ reformulations *do not take into account quantitative information* about the data and query.

We compared the performance of query answering through: (i) UCQ reformulation; (ii) SCQ reformulation; (iii) the JUCQ recommended by the exhaustive ECov algorithm; (iv) the JUCQ recommended by our greedy GCov algorithm.

Figures 3.2 and 3.3 shows the evaluation times for LUBM queries on the 1 million triples and 100 million triples datasets respectively, on the three RDBMSs we tested; observe the logarithmic time axis. Missing bars correspond to executions which timed out or were infeasible. Figures 3.2 and 3.3 shows that neither UCQ nor SCQ reformulation are re-

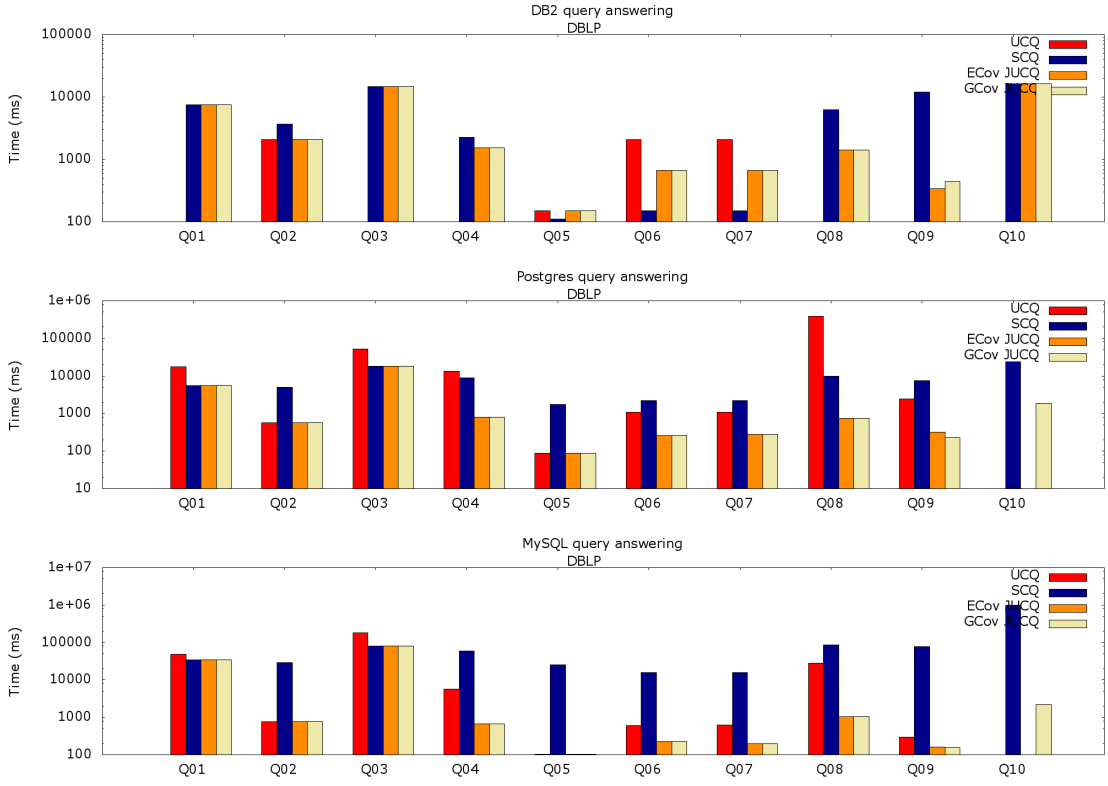

Figure 3.4: DBLP query answering through UCQ, SCQ, ECov and GCov JUCQ reformulations, against DB2, Postgres and MySQL.

liable options. Indeed, UCQ *is the slowest* for many queries on DB2 and Postgres, sometimes by more than an order of magnitude, and it *fails* for $Q_9, Q_{15}, Q_{18}(for LUBM100M), Q_{19}$ and Q_{28} on DB2, to which we add Q_6, Q_{14} and Q_{16} on Postgres (for LUBM 100M). SCQ *is very inefficient* on MySQL, and also on Postgres for Q_1, Q_2, Q_3, Q_8 etc.; it is almost always the worst choice for MySQL. In contrast, the GCov-chosen JUCQ *always completes* and is *the fastest overall* in all but Q_{24}, Q_{25} and Q_{27} on Postgres. Figures 3.2 and 3.3 also shows that the GCov JUCQ performs as well as the ECov one, thus the greedy is making smart choices. In Figure 3.3, the GCov JUCQ is *up to 4 orders of magnitude faster than the* SCQ *reformulation* and *two orders of magnitude faster than* UCQ (on LUBM 1M, it wins by 3 orders of magnitude w.r.t. UCQ). We end by noting that the Q_{16} cover chosen by ECov for Postgres has failed to execute due to insufficient memory in our runtime environment; we believe this could be avoided by further tuning the server execution parameters etc.

Figure 3.4 further highlights that no fixed reformulation technique is always the best. On DB2, SCQ performs very well for Q_5, Q_6 and Q_7, and very poorly for Q_8 and Q_9; on the latter, UCQ times out. In contrast, JUCQ performance is robust, the best in all cases but Q_6

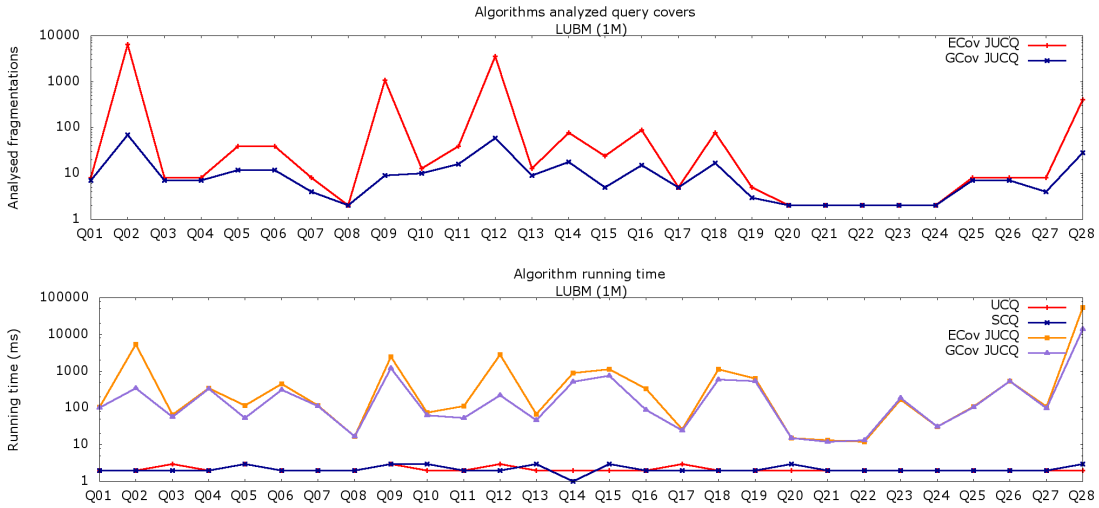

Figure 3.5: Number of query covers explored by the algorithms (top) and algorithm running times (bottom) for the LUBM queries.

and Q_7, and for those it is not very far from the optimum. On Postgres and MySQL the GCov JUCQ performance is the best in all cases, and *more than two orders of magnitude faster than the* SCQ *and* UCQ *reformulation*. These experiments highlight the interest of the JUCQ reformulation space, and the usefulness of our cost model in guiding ECov and GCov search.

GCov performance We now turn to considering *the number of covers*: overall (as explored by the exhaustive ECov), and the subset traversed by our greedy GCov; these are depicted in Figure 3.5 and 3.6, also in logarithmic scale. (Recall that UCQ and SCQ each correspond to one fixed cover.) While the search space can be very large (e.g., for LUBM Q_2, Q_9 or Q_{12}), GCov only explores a reduced subset of this space. The same figures also show *the running time* of GCov and ECov, and the time to build the UCQ, respectively the SCQ reformulations (again, observe the logarithmic time axis). The time is spent to: obtain the statistics necessary for estimating the number of results of various fragments; reformulate each fragment, estimate its cost, and all other steps shown in Algorithms 1 and 2. We see that GCov's running time may be one order of magnitude less than the one of ECov; as expected, building the (cost-ignorant) UCQ and SCQ is quite faster, at the expense of their evaluation time and their unfeasibility to evaluate some queries. The highest running time is recorded for queries having a huge UCQ reformulation (LUBM Q_{28}, respectively DBLP Q_{10}, detailed in Table 3.4); the time is taken to build and estimate statistics for such very large UCQs. In particular, for DBLP Q_{10}, the exhaustive ECov also runs out of memory building the very numerous reformulations.

Alternative: using the RDBMS cost estimation The second question we study is the quality of our cost estimation, that is crucial in guiding GCov decisions. The golden

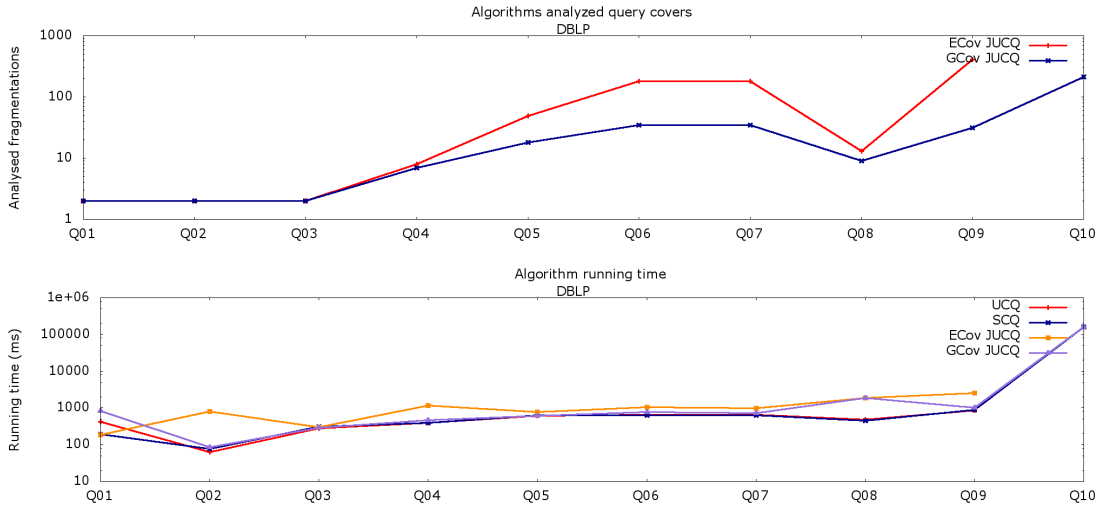

Figure 3.6: Number of query covers explored by the algorithms (top) and algorithm running time (bottom) for DBLP queries.

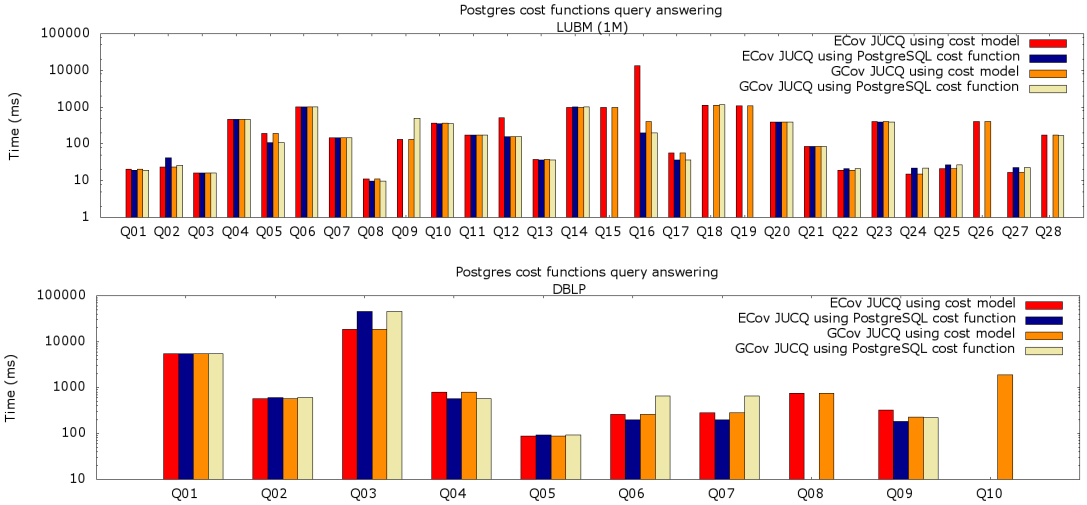

Figure 3.7: Cost model comparison.

standard one can compare against is the RDBMS's internal cost estimation function: this is because any cover we chose is evaluated by sending it (as a SQL statement) to the system which optimizes it according to its internal cost model. Thus, the cost function used by GCov should be as close as possible to the RDBMS one.

For this comparison, whenever we needed to estimate the cost of a cover, we sent to Postgres an `explain` statement for the corresponding cover-based reformulation, and

extracted from its result Postgres' cost estimation[3]

Figure 3.7 shows the evaluation time of the JUCQ reformulations chosen by ECov and GCov, based on one hand on our cost function, and on the other hand on the Postgres one. Most of the time, the results are similar, demonstrating that our cost model is indeed close to the one of Postgres. In a few cases (LUBM Q_{12} and Q_{16}), using Postgres' cost model helped avoid bad ECov decisions; however, for the LUBM queries Q_9, Q_{15}, Q_{18}, Q_{19}, Q_{26} and Q_{28}, as well as the DBLP queries Q_8 and Q_{10}, the ECov JUCQ chosen based on Postgres' cost estimation timed out before completing.

Figure 3.7 demonstrates that our cost model (Section 3.2.1) has lead our algorithm to evaluation choices very similar to the ones that Postgres made, validating its accuracy.

3.3.3 Comparison with saturation

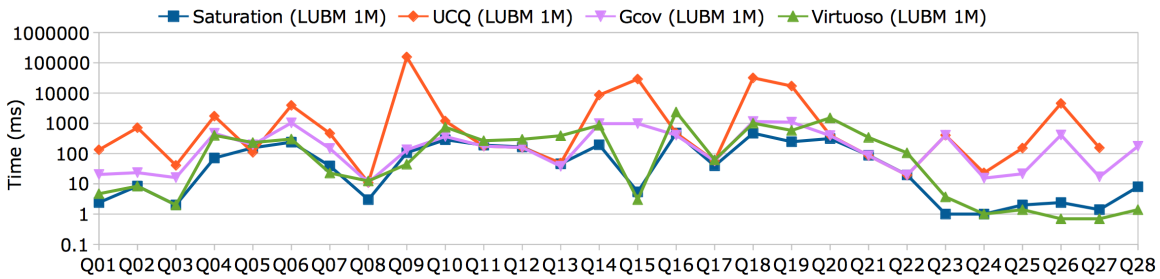

Figure 3.8: Query answering through Virtuoso and Postgres (via saturation, respectively, optimized reformulation).

As explained in the Preliminaries, graph saturation and query reformulation are the two main techniques for answering queries under constraints. Saturation-based query answering can be very efficient, once the data is saturated; however, if the RDF graph is updated, the cost of maintaining the saturation may be very high [48]. In contrast, query reformulation is performed directly at query time, and so it naturally adapts to the current state of the database. The performance trade-off between saturation- and reformulation-based query answering depends on the schema, on the nature of updates, and on the data statistics [48].

In this section, we show how our optimized JUCQ reformulation-based query answering technique impacts the performance comparison with saturation-based query answering. Figure 3.8 compares on the LUBM 1 million triples dataset: (i) UCQ reformulation; (ii) saturation-based query answering based on Postgres; (iii) saturation-based query answering based on Virtuoso v6.1.6 (open-source, multithreaded edition); and

[3]Doing this for every examined cover slowed down our search significantly, thus we do not recommend actually running GCov *out* of a RDBMS based on the RDBMS's *internal* cost model.

(iv) our GCov-chosen JUCQ. As expected, UCQ reformulation performs much worse than saturation-based query answering, and worse than the GCov JUCQ by up to three orders of magnitude. On some queries, such as Q_{15} or $Q_{23} - Q_{28}$, saturation keeps its advantage even compared to our optimized JUCQ reformulation. However, on queries such as $Q_3 - Q_{14}$ and $Q_{16} - Q_{22}$, the JUCQ reformulation *is close to (competitive with) saturation-based query answering*, which is remarkable given that reformulation reasons at query time, and considering the performance gap observed between the two in previous works, e.g., [48].

3.3.4 Experiment conclusion

Our experiments lead to the following conclusions.

(1). Confirming the intuition given by our example in Section 3.1, the space of JUCQ reformulation comprises alternative reformulations of a given BGP query w.r.t. the RDFS constraints, whose evaluation is (i) *feasible when* UCQ *reformulation fails*, and (ii) up to *4 orders of magnitude more efficient* than a fixed reformulation strategy, such as UCQ or SCQ. **(2).** While ECov is slow for large-reformulation queries, GCov identifies covers leading to efficient reformulations quite fast, confirming the feasibility of our optimized reformulation technique at query time. **(3).** The cost model on which our search is based performs globally well; in particular, when calibrated for Postgres, we have shown it leads to chosing covers very close to the ones obtained when relying on Postgres' internal cost model. **(4).** While saturation-based query answering has reasons to be much more efficient than reformulation techniques (*if one is willing to disregard the initial cost of saturating the database, as well as any cost related to saturation maintenance!*), our efficient reformulation technique is in many cases competitive with saturation-based query answering, *both through a relational server and through the native-RDF Virtuoso server*. This confirms the important performance improvement brought by our work to reformulation-based query answering in RDF; recall that any CQ to UCQ reformulation algorithm could be used with our cost-based GCov optimization technique.

3.4 Related work

The context of our work is the problem of answering conjunctive queries against RDF facts, in the presence of RDFS constraints. As mentioned in the Preliminaries, solutions from the literature rely on RDF graph saturation, on query reformulation, or by mixing both [99]; our work focused on making query answering based on reformulation performant. Below, we position our work w.r.t. these two techniques.

Saturation-based query answering. When using graph saturation, all the implicit triples are computed and explicitly added to the database; query answering then reduces to query evaluation on the saturated database. Well-known SPARQL compliant RDF

platforms such as 3store [108], Jena [110], OWLIM [112], Sesame [113], Oracle Semantic Graph [111] support saturation-based query answering, based on (a subset of) RDF entailment rules.

RDF platforms originating in the data management community, such as Hexastore [107] or RDF-3X [83], ignore entailed triples and only provide query *evaluation* on top of the RDF graph, which is assumed to be already saturated.

The drawbacks of saturation w.r.t. updates have been pointed out in [22], which proposes a *truth maintenance* technique implemented in Sesame. It relies on the storage and management of the *justifications* of entailed triples (which triples beget them). This technique incurs a high overhead of handling justifications when their number and size grow. Therefore, [20] proposes to compute only the relevant justifications w.r.t. an update, at maintenance time. This technique is implemented in OWLIM, however [112] points out that updates upon RDFS constraint deletions can lead to poor performance. More efficient saturation maintenance techniques are provided in [48, 98] based on the *number of times* triples are entailed.

Reformulation-based query answering. When using query reformulation, a given BGP query is reformulated based on the RDFS constraints into a target language, such that evaluating the reformulated query through an appropriate engine yields the query answer.

UCQ reformulation [48, 98, 62, 99, 46, 5, 49, 32, 50] applies to various fragments of RDF, ranging from the Description Logics (DL) one up to the Database one, the largest for which this technique have been considered so far. UCQ reformulation corresponds in this work to a JUCQ reformulation obtained from a single fragment query cover. SCQ reformulation [96] was defined for the DL fragment of RDF. In our setting, it corresponds to a JUCQ reformulation obtained from a query cover in which each triple is alone in a fragment. Our experiments have shown that the evaluation performance for both UCQ or SCQ reformulation can be very poor.

Among popular RDF data management systems, the only ones supporting reformulation-based query answering are Stardog, Virtuoso (which supports only the rdfs:subClassOf and rdfs:subPropertyOf RDFS rules) and AllegroGraph [109] (which supports the four RDFS rules but whose reasoning implementation is incomplete[4]). Virtuoso is based on SCQ reformulation, while Stardog uses UCQ reformulation; we found no information about AllegroGraph's query reformulation language. Nested SPARQL is the target reformulation language in [11]; in contrast, we focus on translating into a commonly supported language such as JUCQs which in turn can be efficiently evaluated by an SQL engine. In [99], the schema is maintained saturated and reformulation is applied at runtime. Our approach could apply in that setting, to improve their reformulation performance.

[4]As stated at http://franz.com/agraph/support/ documentation/v4/reasoner-tutorial.html#fnr0-2014-09-16

Datalog has also been used as a target reformulation language. For instance, Presto [46, 89] reformulates queries in a DL-Lite setting into non-recursive Datalog programs. These DL-Lite formalisms are strictly more expressive from a semantic constraint viewpoint than the RDFS constraints we consider. Thus, their method could be easily transferred (restricted) to the DL fragment of RDF which, as previously mentioned, is a subset of the database fragment of RDF that we consider. However, these works did not consider cost-driven performance optimization based on data statistics and a query evaluation cost model as in our work.

From a *database optimization* perspective, the performance advantage we gain by adding selective triples next to very large ones within query covers' fragments is akin to the semi-join reducers technique, well-known from the distributed database context [85]. It has been shown e.g., in [92] that semi-join reducers can also be beneficial in a centralized context by reducing the overall join effort. In this work, we use a technique reminiscent of semi-joins in order to pick the best *query-level* formulation of a reformulated query, to make its evaluation possible and efficient; this contrasts with the traditional usage of semi-joins *at the level of algebraic plans*.

On one hand, working at the plan level enables one to intelligently combine traditional joins and semi-joins to obtain the best performance. On the other hand, producing (as we do) an output at the *query (syntax)* level (recall Figure 3.1) enables us to take advantage of any existing system, and of its optimizer which will figure out the best way to evaluate such queries, a task at which many systems are good once the query has a "reasonable" shape and size. As shown in [71], providing the relational database systems queries of "manageable" size is key, as that RDBMSs' estimation errors grows fast with the increase of the number of joins, usually leading to bad plans and therefore poor performance. Further, expressing optimized reformulations as queries allows us *not* to (re-)explore the search space of join orders etc. together with the (already large) space of possible reformulated queries.

3.5 Conclusion

Our work is placed in the setting of query answering against RDF graphs in the presence of RDF Schema constraints. In particular, we focus on *improving the performance of reformulation-based RDF query answering*.

We have identified a space of alternative JUCQ reformulations, whose evaluation (based on a standard, semantics-unaware query processor) may be (i) feasible even when the prominent UCQ reformulation is not, and (ii) more efficient by up to three orders of magnitude. Further, we have presented a cost model for such JUCQ alternatives, and proposed an anytime greedy cost-based algorithm capable of identifying such efficient alternatives. Our technique may be used with any CQ-to-UCQ query reformulation algorithm (recall Figure 3.1) and thus we consider it a big step forward toward making

reformulation-based query answering efficient. This is particularly useful in contexts when the data and/or constraints are updated, and saturation-based techniques incur high maintenance costs as illustrated e.g., in [48]; in contrast, applying at query time, reformulation-based query answering is naturally robust to updates, and (through cost-based techniques such as the one described in our work) close to saturation-based performance but without its drawbacks.

Chapter 4

Efficient FOL reducible query answering

In this chapter we transfer the idea developed in the preceding chapter to the general setting of data model and query language pairs enjoying FOL *reducibility of query answering* (i.e., data model and query language pairs for which query answering can be reduced to evaluating a certain first-order logic formula, obtained from the query and ontology, against only the explicit facts), encompassing many knowledge base and deductive database settings, e.g., some Description Logics, Datalog$^{\pm}$ and Existential Rules fragments.

We propose a query optimization framework for *any* logical OBDA setting enjoying FOL reducibility of query answering. We extend the language of FOL reformulations beyond those considered so far in the literature, and investigate *several (equivalent)* FOL *reformulations* of a given query, out of which we pick one likely to lead to the best evaluation performance. This contrasts with existing works from the semantic query answering literature (cf. Section 4.6), which use reformulation languages allowing *single* FOL reformulation (modulo minimization). Considering a *set of reformulations* and relying on a *cost model* to pick a most efficient one has a very visible impact on the efficiency and feasibility of query answering: indeed, picking the wrong reformulation may cause the RDBMS simply to fail evaluating it (typically due to very lengthy queries), while in other cases it leads to bad performance.

We apply this general framework to the DL-Lite$_{\mathcal{R}}$ Description Logic [32] underpinning the popular W3C's OWL2 QL standard for rich Semantic Web applications, demonstrating significant performance advantages in this setting. Query answering in DL-Lite$_{\mathcal{R}}$ has received significant attention in the literature, notably techniques based on FOL reducibility, e.g., [32, 2, 86, 89, 36, 100].

Contributions. We bring the following contributions to the problem of optimizing FOL reducible query answering (see Figure 4.1):

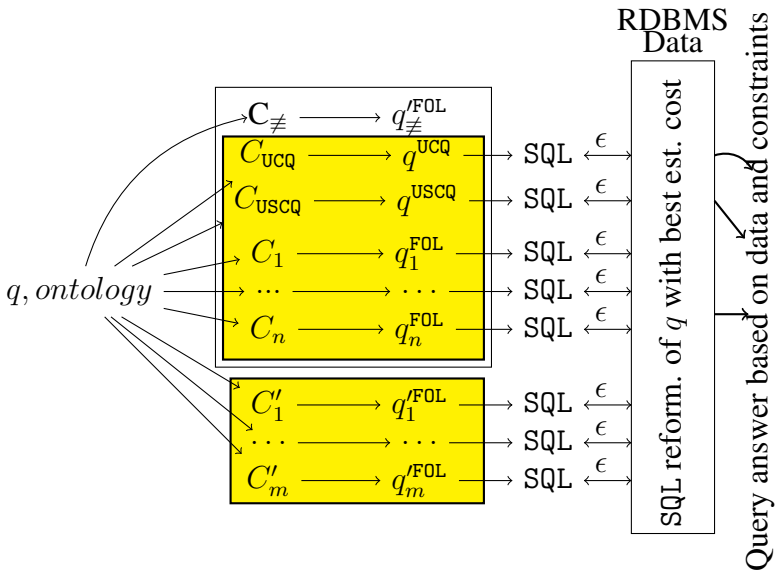

Figure 4.1: Optimized FOL reformulation approach.

1. For logical formalisms enjoying FOL reducibility of query answering, we provide a general *optimization framework* that reduces query answering to searching among a set of alternative equivalent FOL reformulations, one with minimal evaluation cost in an RDBMS (Section 4.2). In Figure 4.1, from the query q and the set of ontological constraints $\mathcal{T}$, we derive first, a space of *query covers*, shown in the top white-background box, and denoted C with some subscripts; from each such cover we show how to derive a FOL query that *may* be a FOL reformulation of q w.r.t. $\mathcal{T}$.

2. We characterize interesting spaces of such alternative equivalent FOL queries for DL-Lite$_\mathcal{R}$ (Section 4.3).
First, we identify a sufficient *safety* condition to pick covers that for sure lead to FOL reformulations of the query. This condition is met by the covers in the top yellow box in Figure 4.1, and is not met by $C_{\not\equiv}$ above them. Our safe cover space allows considering FOL reformulations encompassing those previously studied in the literature. Second, we introduce a set of *generalized covers* (bottom yellow box in Figure 4.1) and a generalized cover-based reformulation technique, which always yields FOL query reformulations, oftentimes more efficient than those based on simple covers.
Our approach can be combined with, and helps optimizing, any existing reformulation technique for DL-Lite$_\mathcal{R}$.

3. We then optimize query answering in the setting of DL-Lite$_\mathcal{R}$ by enumerating simple and generalized covers, and *picking a cover-derived* FOL *reformulation with lowest estimated evaluation cost* w.r.t. an RDBMS cost model estimation ϵ (denoted by the bidirectional ϵ-labeled arrows in the figure). We provide two algorithms, an exhaustive and a greedy, for this task (Section 4.4).

4. Evaluating any of our FOL reformulations through an RDBMS leads (thick arrows at the right of Figure 4.1) to the query answer reflecting both the data and the constraints. We demonstrate *experimentally* the effectiveness and the efficiency of our query answering technique for DL-Lite$_{\mathcal{R}}$, by deploying our query answering technique on top of Postgres and DB2, using several alternative data layouts (Section 4.5).

From a query processing and optimization perspective, our approach can be seen as belonging to the so-called *strategic optimization* stage introduced in [77] (where application semantics is injected into the query); it is also similar in spirit to the *syntax-level rewrites* performed by optimizers such as Oracle 10g's [7]. We share with [77] the idea of injecting semantics first, and like [7], we use cost estimation to guide our rewrites; a common theme is to rewrite before ordering joins, selecting physical operators etc.From this angle, our contribution can be seen as *a set of alternatives (rewritings) with correctness guarantees and algorithms to guide such rewritings*, for the special class of queries obtained from FOL reformulations of CQ against ontologies.

In the sequel, we detail the above contributions and discuss related work and conclude in Section 4.6.

The work reported here is based on the VLDB paper [28] and the ISWC demo [27].

4.1 Evaluating reformulated subqueries can be (very) hard

It is worth noting that the (naïve) exhaustive application of specialization steps leads, in general, to *highly redundant reformulations* w.r.t. the containment of their disjuncts. For instance, minimizing $q^{\texttt{UCQ}}$ introduced in Example 8 by eliminating disjuncts contained in another leads to: $q^{\texttt{UCQ}}_{min}(x)$:- $\bigvee_{i=1}^{3} q^i(x) \vee q^{10}(x)$ where the disjuncts appear in Table 2.6; they are all contained in q^{10}.

$$
\begin{aligned}
q^1(x) &\text{:- } \texttt{PhDStudent}(x) \wedge \texttt{worksWith}(y, x)\\
q^2(x) &\text{:- } \texttt{PhDStudent}(x) \wedge \texttt{worksWith}(x, y)\\
q^3(x) &\text{:- } \texttt{PhDStudent}(x) \wedge \texttt{supervisedBy}(y, x)\\
q^{10}(x) &\text{:- } \texttt{supervisedBy}(x, y)
\end{aligned}
$$

Table 4.1: Union terms in minimized CQ-to-UCQ reformulation.

Minimal UCQ reformulations can be obviously processed more efficiently. However, they still repeat some computations, e.g., in the above minimized CQ-to-UCQ reformulation example, PhDStudent is read three times, worksWith twice etc.; in general, subqueries appearing in different union terms are repeatedly evaluated.

Common subexpression elimination (CSE) techniques aim at identifying repeated subex-

pressions in queries or plans, and reformulating them so that the expression is evaluated only once and its results are shared to increase performance; CSE is often used in a Multi-Query Optimization context (MQO). However, MQO is poorly supported in today's main RDBMS engines[1] As we will see, our approach, which starts with the TBox and data statistics, and ends by handing over a chosen reformulation to the RDBMS, *never requires work to detect common (repeated) sub-expressions*.

Another source of difficulty is the *sheer size of reformulated queries*; we exhibit some whose size (i.e., length of the SQL formulation) is above 2.000.000 characters. For instance, the minimal UCQ corresponding to query Q_9 in our experiments (Section 4.5) is a union of 145 CQs, and runs in 5665 ms on DB2 and a database of 100 million facts. In contrast, the SQL translation of the best FOL reformulation identified by our approach reduces this to 156 ms (36 times faster), just by giving the engine a different (yet equivalent) SQLized FOL reformulation.

From an optimization viewpoint, the problem we are facing can be seen as follows. We aim at answering queries through RDBMSs in the presence of constraints, for FOL-reducible settings.

The standard UCQ reformulation (and other cost-ignorant ones) perform quite badly. The question is, then: is there an equivalent reformulation which would be evaluated more efficiently?

To answer this, one is faced with a set of FOL (or, alternatively, SQL) reformulations whose size is potentially very high: exponential in the query size for non-redundant queries, larger yet if one considers, for instance, queries featuring semijoins [19]; each query therein may be (very) large, have many unions etc.. From these, one would need to find the one(s) best optimized and executed by the RDBMS; their very high number makes this utterly impractical.

The following sections present our alternative approach.

4.2 Optimization framework

The performance of evaluating (the SQL translation of) a given FOL reformulation of a query through an RDBMS depends on several factors: (i) data properties (size, cardinalities, value distributions etc.); (ii) the storage model, i.e., the concrete relations storing the ABox, possible indexes etc.; (iii) the optimizer's algorithm. Among these, (i) is completely determined by the dataset (the given ABox). On the storage model (ii), for generality, we make no assumption, other than requiring that FOL query reformulations can be translated into SQL on the underlying store. (We study several such concrete models experimentally, in Section 4.5). For what concerns optimizers (iii),

[1] We checked this on Postgres, DB2, and MySQL plans; according to Paul Larson (among the authors of [114]), no major RDBMS engine as of April 2016 has a comprehensive MQO approach.

we note that off-the-shelf they perform very poorly on previously proposed FOL query reformulations, yet we would like to exploit their strengths when possible.

Approach: cover-based query answering. We identify and exploit a novel space of *alternative* FOL *reformulations of the given input* CQ. We estimate the cost of evaluating each such reformulation through the RDBMS using standard database cost formulas, and hand to the RDBMS one with the best estimation.

More specifically, a query *cover* defines a way to split the query into subqueries, that may overlap, called *fragment queries*, such that substituting each subquery with its FOL reformulation (obtained from any state-of-the-art technique) and joining the corresponding (reformulated) subqueries, *may* yield a FOL reformulation of the original query (recall also Figure 4.1).

We begin by recasting the specific query *covers* from the RDFS setting of the preceding chapter into the very general framework of FOL reducible query answering.

Definition 4.2.1 (CQ cover)**.** *A* cover *of a query q, whose atoms are $\{a_1, \ldots, a_n\}$, is a set $C = \{f_1, \ldots, f_m\}$ of non-empty subsets of atoms of q, called* fragments*, such that (i) $\bigcup_{i=1}^{m} f_i = \{a_1, \ldots, a_n\}$, (ii) no fragment is included into another, and (iii) the atoms of each fragment are connected through joins (common variables).*

Example 9 (CQ **cover**)**.** *Consider the query*

$$q(x, y) \text{:-} \quad \texttt{teachesTo}(v, x) \wedge \texttt{teachesTo}(v, y), \\ \texttt{supervisedBy}(x, w) \wedge \texttt{supervisedBy}(y, w)$$

C, below, is a query cover for q:

$$C = \{\{\texttt{teachesTo}(v, x) \wedge \texttt{supervisedBy}(x, w)\}, \\ \{\texttt{teachesTo}(v, y) \wedge \texttt{supervisedBy}(y, w)\}\}$$

Definition 4.2.2 (Fragment queries of a CQ)**.** *Let $C = \{f_1, \ldots, f_m\}$ be a cover of q. A* fragment query $q_{|f_i, 1 \leq i \leq m}$ *of q w.r.t. C is the subquery whose body consists of the atoms in f_i and whose free variables are the free variables $\bar{x}$ of q appearing in the atoms of f_i,* plus *the existential variables in f_i that are shared with another fragment $f_{j, 1 \leq j \leq m, j \neq i}$, i.e., on which the two fragments join.*

Example 10 (Fragment queries of a CQ**).** *The fragment queries of the query $q(x, y)$ w.r.t. the cover C (Example 9) are:*

$$q_{|f_1}(x, v, w) \text{:-} \ \texttt{teachesTo}(v, x) \wedge \texttt{supervisedBy}(x, w)$$

$$q_{|f_2}(y, v, w) \text{:-} \ \texttt{teachesTo}(v, y) \wedge \texttt{supervisedBy}(y, w)$$

As we shall see in the next Section, not every cover of a query leads to a FOL reformulation. Specifically, we define:

Definition 4.2.3 (Cover-based reformulation)**.** *Let $C = \{f_1, \ldots, f_m\}$ be a cover of q, and $q^{\texttt{FOL}}(\bar{x})$:- $\bigwedge_{i=1}^{m} q^{\texttt{FOL}}_{|f_i}$ a* FOL *query, where $q^{\texttt{FOL}}_{|f_i}$, for $1 \leq i \leq m$, is a* FOL *reformulation w.r.t. $\mathcal{T}$ of the fragment query $q_{|f_i}$ of q.*

$q^{\texttt{FOL}}$ is a cover-based reformulation of q w.r.t. $\mathcal{T}$ and C *if it is a* FOL *reformulation of q w.r.t. $\mathcal{T}$.*

To exemplify cover-based FOL reformulations, one needs to chose a specific KB dialect, among all those enjoying FOL reducibility; we present examples in the next Section, when instantiating our framework to the DL-Lite$_{\mathcal{R}}$ setting.

For now, it helps to see how we derive the SQL query corresponding to the cover-based reformulation. Each reformulated fragment query $q^{\texttt{FOL}}_{|f_i}$ is translated into an SQL query $\texttt{SQL}_\texttt{i}$; then, for those RDBMSs enjoying Common Table Expressions (CTEs) the overall query is of the form:

$\texttt{WITH SQL}_1\ \texttt{AS}\ \texttt{q}^{\texttt{FOL}}_{|\texttt{f}_1},\ \texttt{SQL}_2\ \texttt{AS}\ \texttt{q}^{\texttt{FOL}}_{|\texttt{f}_2}, \ldots, \texttt{SQL}_\texttt{n}\ \texttt{AS}\ \texttt{q}^{\texttt{FOL}}_{|\texttt{f}_\texttt{n}}$
$\texttt{SELECT DISTINCT}\ \bar{\texttt{x}}\ \texttt{FROM}\ \texttt{SQL}_1, \texttt{SQL}_2, \ldots, \texttt{SQL}_\texttt{n}$
$\texttt{WHERE cond}(1, 2, \ldots, \texttt{n})$

where $\texttt{cond}(1, 2, \ldots, \texttt{n})$ is the conjunction of the join predicates between all the subqueries. This leads to all the WITH-introduced subqueries being evaluated and materialized into intermediary tables[2], while the one with the largest number of results is run in pipeline fashion. The way in which each subquery is evaluated, then their results are joined, is left to the DBMS to determine. The SELECT DISTINCT ensures set semantics for the query answers.

We picked this syntax after experimenting with other variants, which in our experience lead to similar or worse performance. In particular, we tried:

- using one subquery for each fragment query $\texttt{SQL}_\texttt{i}$, and joining them. Our experiments showed no improvement in general,however it's a good variant for RDBMSs that do not support CTEs.
- defining each reformulated fragment query $\texttt{SQL}_\texttt{i}$ as a (virtual) view, and joining these views in the global reformulation. This gives the query processor more freedom as it does no longer force the materialization of $\texttt{SQL}_\texttt{i}$ but instead allows its evaluation to be blended with the evaluation of the joins across reformulated fragment queries. We noticed, however, that this did not overall improve performance.
- turning $\texttt{SQL}_\texttt{i}$ subqueries into nested ones introduced with EXISTS as soon as the subquery did not contribute variables to the head of reformulated query. We tried

[2]These SQL subqueries are of the form SELECTDISTINCT in order to reduce the size of the intermediate materialized results; this choice lead to the fastest execution in our experiments.

this both on the WITH version and on the view-based versions; we did not notice significant improvements.

Problem statement. We assume given a *query cost estimation function* ϵ which, for any FOL query q, returns the cost of evaluating it through an RDBMS storing the ABox. Thus, ϵ reflects the operations (data access, joins, unions etc.) applied on the ABox to compute the answers of a q^{FOL} reformulation. The cost estimation ϵ also accounts for the effort needed to join the reformulated fragment query answers, in the most efficient way.

Problem 1 (Optimization problem)**.** *Given a* CQ q *and a KB* $\mathcal{K}$*, the* cost-driven cover-based query answering problem *consists of finding a cover-based reformulation of* q *based on* $\mathcal{K}$ *with lowest (estimated) evaluation cost.*

A cost estimation function is provided by most RDBMSs storing the ABox for instance, in the case of Postgres, through the SQL `explain` directive. One can also estimate costs outside the engine using well-known textbook formulas, as in e.g., Chapter 3 (Section 3.2.1) and [71]. We use both options in our experiments.

4.3 Cover-based query answering in DL-Lite$_\mathcal{R}$

We now instantiate our cover-based query answering technique to the popular setting of DL-Lite$_\mathcal{R}$. For establishing our results as well as for our examples we rely on the simple CQ-to-UCQ reformulation technique of [32]. However, our approach applies to *any* other FOL reformulation techniques for DL-Lite$_\mathcal{R}$, e.g., optimized CQ-to-UCQ or CQ-to-USCQ reformulation techniques, since these produce *equivalent* (though possibly syntactically different) FOL queries.

Example 11 (**Running example**)**.** *Let* $\mathcal{K}$ *be the KB with TBox* $\mathcal{T} = \{\texttt{Graduate} \sqsubseteq \exists\texttt{supervisedBy}, \texttt{supervisedBy} \sqsubseteq \texttt{worksWith}\}$ *and ABox*

$$\mathcal{A} = \{\texttt{PhDStudent}(Damian), \texttt{Graduate}(Damian)\}$$

Consider the query $q(x)$:- $\texttt{PhDStudent}(x) \wedge \texttt{worksWith}(x, y) \wedge \texttt{supervisedBy}(z, y)$*, whose answer against* $\mathcal{K}$ *is* $\{Damian\}$.

The UCQ *reformulation of* q *is* $q^{\text{UCQ}}(x)$:- $\bigvee_{i=1}^{4} q^i(x)$ *with:*

$$\begin{aligned}
q^1(x)\text{:-}\ & \texttt{PhDStudent}(x) \wedge \texttt{worksWith}(x, y) \\
& \wedge \texttt{supervisedBy}(z, y) \\
q^2(x)\text{:-}\ & \texttt{PhDStudent}(x) \wedge \texttt{supervisedBy}(x, y) \\
& \wedge \texttt{supervisedBy}(z, y) \\
q^3(x)\text{:-}\ & \texttt{PhDStudent}(x) \wedge \texttt{supervisedBy}(x, y) \\
q^4(x)\text{:-}\ & \texttt{PhDStudent}(x) \wedge \texttt{Graduate}(x)
\end{aligned}$$

Above, q^1 has the body of q; q^2 is obtained from q^1 by specializing the atom $\texttt{worksWith}(x, y)$ *through a backward application of* $\texttt{supervisedBy} \sqsubseteq \texttt{worksWith}$. *$q^3$ (highlighted in blue) results from q^2 by replacing* $\texttt{supervisedBy}(x, y)$ *and* $\texttt{supervisedBy}(z, y)$ *with their most general unifier*[3]. *Finally, q^4 is obtained from q^3, by specializing* $\texttt{supervisedBy}(x, y)$ *through the backward application of* $\texttt{Graduate} \sqsubseteq \exists\texttt{supervisedBy}$*; we also show q^4 in blue to highlight its connection with q^3.*

Now let $C_1 = \{\{\texttt{PhDStudent}(x), \texttt{worksWith}(x, y)\}, \{\texttt{supervisedBy}(z, y)\}\}$ *be a cover of q. From Definition 4.2.2, the corresponding fragment queries are:*

$$q_1(x, y) \text{:-}\ \texttt{PhDStudent}(x) \wedge \texttt{worksWith}(x, y)$$
$$q_2(y) \text{:-}\ \texttt{supervisedBy}(z, y)$$

The reformulation of q_1 using $\mathcal{T}$ is $q_1^{\texttt{UCQ}}(x, y) \text{:-}\ \bigvee_{i=1}^{2} q_1^i(x, y)$, *where*

$$q_1^1(x, y) \text{:-}\ \texttt{PhDStudent}(x) \wedge \texttt{worksWith}(x, y)$$
$$q_1^2(x, y) \text{:-}\ \texttt{PhDStudent}(x) \wedge \texttt{supervisedBy}(x, y)$$

q_1^2 is obtained from q_1^1 by the backward application of the constraint $\texttt{supervisedBy} \sqsubseteq \texttt{worksWith}$.

The reformulation of q_2 using $\mathcal{T}$ is simply:

$$q_2^{\texttt{UCQ}}(y) \text{:-}\ \texttt{supervisedBy}(z, y)$$

By Definition 4.2.3, the reformulation of q using C_1 is the conjunction $q_{C_1}^{\texttt{JUCQ}}(x) \text{:-}\ q_1^{\texttt{UCQ}}(x, y) \wedge q_2^{\texttt{UCQ}}(y)$, *which is clearly equivalent to the following* UCQ *obtained by distributing $\wedge$ over $\vee$:*

$$q_{C_1}^{\texttt{UCQ}}(x) \text{:-}\ (q_1^1(x, y) \wedge q_2^{\texttt{UCQ}}(y)) \vee (q_1^2(x, y) \wedge q_2^{\texttt{UCQ}}(y))$$

where the first and second disjuncts correspond to the CQ*s:*

$$q_{C_1}^1(x) \text{:-}\ \texttt{PhDStudent}(x) \wedge \texttt{worksWith}(x, y) \wedge \texttt{supervisedBy}(z, y)$$
$$q_{C_1}^2(x) \text{:-}\ \texttt{PhDStudent}(x) \wedge \texttt{supervisedBy}(x, y) \wedge \texttt{supervisedBy}(z, y)$$

Above, $q_{C_1}^1(x)$ and $q_{C_1}^2(x)$ are exactly q^1 and q^2 from the UCQ *reformulation of q; however, q^3 and q^4 are missing from $q_{C_1}^{\texttt{JUCQ}}(x)$. Since q^4 derives from q^3, the absence of both can be traced to the absence of q^3. The reason C_1 does not lead to q_3 is that* $\texttt{supervisedBy}(x, y) \wedge \texttt{supervisedBy}(z, y)$ *is not obtained while reformulating $q_1(x, y)$, thus the unification of these two atoms (which could have lead to q^3) is missed. In the* CQ*-to-*UCQ *reformulation of q,* $\texttt{supervisedBy}(x, y) \wedge \texttt{supervisedBy}(z, y)$ *appears in q^2*

[3] In this case, the *mgu* is $\texttt{supervisedBy}(x, y)$ because x is the head variable. Also, q^3 is equivalent to (and a minimal form of) q^2, but in general, q^3 is only guaranteed to be contained in (or equivalent to) q^2.

because $\texttt{worksWith}(x, y) \wedge \texttt{supervisedBy}(z, y)$ *appears in* q^1. *However,* C_1 separates the worksWith and supervisedBy atoms in different fragments. *Reformulating them independently misses exactly the opportunity to produce* q^3 *and* q^4.

Due to these absent subqueries, $q^{\texttt{JUCQ}}_{C_1}$ *is not a* FOL *reformulation of* q *w.r.t.* $\mathcal{T}$*, i.e., it fails to compute* q*'s answer:* $ans(q^{\texttt{JUCQ}}_{C_1}, \langle \emptyset, \mathcal{A} \rangle) = \emptyset$ *while the answer of* q *against* $\mathcal{K}$ *is* $\{Damian\}$.

More generally, given an input CQ and a TBox, each pair of query atoms begetting unifications during the CQ-to-UCQ reformulation of the whole query must not be separated by (must not be in different fragments of) a cover, in order for the corresponding cover-based reformulation to be a FOL reformulation. When this is the case, we say the cover is *safe* for query answering.

Thus, we are interested in a sufficient condition for a cover to be safe; intuitively, we must approximate (by some supersets) those sets of atoms which (directly or after some specializations) are pairwise unified by the CQ-to-UCQ algorithm, and ensure that each such atom set is in the same cover fragment.

Only atoms with the same predicate may unify. Thus, we identify for each *predicate* (i.e., concept or role name) occurring in a query, *the set of all TBox predicates in which this predicate may turn* through some sequence of atom specializations, i.e., backward constraint application and/or unification (the two operations applied by the technique of [32] which we consider here). This is captured by the classical notion of dependencies between predicates within knowledge bases, Datalog programs, etc. In DL-Lite$_\mathcal{R}$, this notion translates into the following recursive definition.

Definition 4.3.1 (Concept and role dependencies w.r.t. a TBox)**.** *Given a TBox* $\mathcal{T}$*, a concept or role name* N depends w.r.t. $\mathcal{T}$ on *the set of concept and role names denoted* $dep(N)$ *and defined as the fixpoint of:*

$$\begin{aligned} dep^0(N) &= \{N\} \\ dep^n(N) &= dep^{n-1}(N) \\ &\quad \cup \{cr(Y) \mid Y \sqsubseteq X \in \mathcal{T} \text{ and } cr(X) \in dep^{n-1}(N)\} \end{aligned}$$

where $cr(Y)$ *returns, for any input* Y *of the form* Z, Z^- *or* $\exists Z$ *(for some concept or role* Z*), the concept or role name* Z *itself.*

Example 12 (Predicate dependencies). *In the TBox of Example 11:*

$$\begin{aligned} dep(\texttt{PhDStudent}) &= \{\texttt{PhDStudent}\} \\ dep(\texttt{Graduate}) &= \{\texttt{Graduate}\} \\ dep(\texttt{worksWith}) &= \{\texttt{worksWith}, \texttt{supervisedBy}, \texttt{Graduate}\} \\ dep(\texttt{supervisedBy}) &= \{\texttt{supervisedBy}, \texttt{Graduate}\} \end{aligned}$$

Above, worksWith *depends on* supervisedBy *because of the constraint* $\texttt{supervisedBy} \sqsubseteq \texttt{worksWith}$*; similarly,* supervisedBy *depends on* Graduate *due*

to the constraint Graduate $\sqsubseteq \exists$supervisedBy*, thus* worksWith *in turn depends on* Graduate*, too.*

Definition 4.3.2 (Safe cover for query answering)**.** *A cover C of q is* safe for query answering w.r.t. $\mathcal{T}$ *(or* safe *in short) iff it is a partition of q's atoms such that two atoms whose predicates depend on a common concept or role name w.r.t. $\mathcal{T}$ are in a same fragment.*

Note that while Definition 4.3.2 requires covers to be partitions, we will relax this restriction in Section 4.4.2.

Theorem 4.3.1 (Cover-based query answering)**.** *Let C be a safe cover for q w.r.t. $\mathcal{T}$. The cover-based reformulation (Definition 4.2.3) of q based on C, using any* CQ*-to-*UCQ *(resp.* CQ*-to-*USCQ*) reformulation technique, yields a cover-based reformulation $q^{\texttt{FOL}}$ of q w.r.t. $\mathcal{T}$.*

Proof. The proof follows from that of correctness of the CQ-to-UCQ reformulation technique in [32] for query answering. It directly *extends* to the use of any CQ-to-UCQ or CQ-to-USCQ reformulation technique for DL-Lite$_\mathcal{R}$, as, for any CQ and TBox, the FOL queries they compute are equivalent to the query produced by the technique described in [32].

Soundness: for any $\mathcal{T}$-consistent Abox $\mathcal{A}$, $ans(q^{\texttt{FOL}}, \langle\emptyset, \mathcal{A}\rangle) \subseteq ans(q, \langle\mathcal{T}, \mathcal{A}\rangle)$ holds.

Let t be a tuple in $ans(q^{\texttt{FOL}}, \langle\emptyset, \mathcal{A}\rangle)$. From Definition 4.2.3, $q^{\texttt{FOL}}$ is $q^{\texttt{FOL}}(\bar{x})$:- $\bigwedge_{i=1}^{m} q_{|f_i}^{\texttt{FOL}}$, thus t results from $t_i \in ans(q_{|f_i}^{\texttt{FOL}}, \langle\emptyset, \mathcal{A}\rangle)$, for $1 \leq i \leq m$. Therefore, for $1 \leq i \leq m$, $t_i \in ans(q_{|f_i}, \langle\mathcal{T}, \mathcal{A}\rangle)$ holds, because of the soundness of the CQ-to-UCQ reformulation technique. Hence, from Definition 4.2.3, $t \in ans(q, \langle\mathcal{T}, \mathcal{A}\rangle)$ holds.

Completeness: for any $\mathcal{T}$-consistent Abox $\mathcal{A}$, $ans(q, \langle\mathcal{T}, \mathcal{A}\rangle) \subseteq ans(q^{\texttt{FOL}}, \langle\emptyset, \mathcal{A}\rangle)$ holds.

Let t be a tuple in $ans(q, \langle\mathcal{T}, \mathcal{A}\rangle)$. Let $q^{\texttt{UCQ}}$ be its reformulation using the CQ-to-UCQ technique. From the completeness of this technique, $t \in ans(q^{\texttt{UCQ}}, \langle\emptyset, \mathcal{A}\rangle)$ holds. Let $q^{\texttt{UCQ}}$ be $\bigvee_{l=1}^{\alpha} cq_l$, then necessarily for some l: $t \in ans(cq_l, \langle\emptyset, \mathcal{A}\rangle)$ holds [32].

Let $q^{\texttt{FOL}}$ be $\bigwedge_{i=1}^{m} q_{|f_i}^{\texttt{FOL}} = \bigwedge_{i=1}^{m} \bigvee_{j=1}^{\beta_i} cq_{i,j}$. Since Definition 4.3.2 makes the reformulation of each fragment independent from another w.r.t. the CQ-to-UCQ technique, for any cq_l in $q^{\texttt{UCQ}}$: $cq_l = \bigwedge_{i=1}^{m} cq_{i,k\in[1,\beta_i]}$ holds. Hence, $t \in ans(q^{\texttt{FOL}}, \langle\emptyset, \mathcal{A}\rangle)$ holds. □

If a CQ-to-UCQ reformulation algorithm is used on fragment queries, the cover-based reformulation will be a JUCQ; otherwise, a CQ-to-USCQ reformulation of the fragment queries lead to a JUSCQ reformulation.

Note that the trivial one-fragment cover (comprising all query atoms) is always safe; in this case, our query answering technique reduces to just one reformulation, the CQ-to-UCQ one identified by previous reformulation algorithms from the literature.

Example 13 (JUCQ **reformulation with a safe cover**). *We now consider the* safe *cover* $C_2 = \{\{\texttt{PhDStudent}(x)\}, \{\texttt{worksWith}(x, y), \texttt{supervisedBy}(z, y)\}\}$. *The cover-based reformulation based on* C_2 *is the* JUCQ *query* $q^{\texttt{JUCQ}}(x)$*:-* $q_1^{\texttt{UCQ}}(x) \wedge q_2^{\texttt{UCQ}}(x)$*, where:*

$$
\begin{aligned}
q_1^{\texttt{UCQ}}(x)\text{:-}\ & \texttt{PhDStudent}(x)\\
q_2^{\texttt{UCQ}}(x)\text{:-}\ & (\texttt{worksWith}(x, y) \wedge \texttt{supervisedBy}(z, y))\\
& \vee (\texttt{supervisedBy}(x, y) \wedge \texttt{supervisedBy}(z, y))\\
& \vee \texttt{supervisedBy}(x, y) \vee \texttt{Graduate}(x)
\end{aligned}
$$

Observe that $ans(q^{\texttt{JUCQ}}, \langle \emptyset, \mathcal{A}\rangle) = \{Damian\} = ans(q, \langle \mathcal{T}, \mathcal{A}\rangle)$.

4.4 Cover-based query optimization in DL-Lite$_\mathcal{R}$

We study now the query answering optimization problem of Section 4.2 for DL-Lite$_\mathcal{R}$. We analyze a first optimization space in Section 4.4.1, before extending our discussion to a larger space in Section 4.4.2. Finally, we describe our search algorithms in Section 4.4.3.

4.4.1 Safe covers optimization space

Below, we study the space of safe covers for a given query and TBox. We start by identifying a particularly interesting one:

Definition 4.4.1 (Root cover). *We term* root cover *for a query* q *and TBox* $\mathcal{T}$ *the cover* C_{root} *obtained as follows. Start with a cover* C_1 *where each atom is alone in a fragment. Then, for any pair of fragments* $f_1, f_2 \in C_1$ *and atoms* $a_1 \in f_1, a_2 \in f_2$ *such that there exists a predicate on which those of* a_1 *and* a_2 *depend w.r.t.* $\mathcal{T}$*, create a fragment* $f' = f_1 \cup f_2$ *and a new cover* $C_2 = (C_1 \setminus \{f_1, f_2\}) \cup \{f'\}$. *Repeat the above until the cover is stationary; this is the root cover, denoted* C_{root}.

It is easy to see that C_{root} does not depend on the order in which the fragments are considered (due to the inflationary method building it). Further, C_{root} is safe, given that it keeps in a single fragment any two atoms whose predicates may be unified.

The following important lemma characterizes the structure of C_{root} fragments:

Lemma 1 (C_{root} fragment structure). *A fragment* f *in the root cover* C_{root} *is of one of the following two forms:*

1. *a singleton, i.e.,* $f = \{a_i\}$ *for some query atom* a_i*;*
2. $f = \{a_{i_1}, \ldots, a_{i_n}\}$*, for* $n \geq 2$*, and for every atom* $a_{i_1} \in f$*, there exists one atom* $a_{i_2} \in f$*, and a predicate* b_j *in the TBox, such that both the predicates of* a_{i_1} *and of* a_{i_2} *depend on* b_j.

Proof. The lemma follows directly from the definition of C_{root}. Those atoms that do not share a dependency with any other atom appear in singleton fragments (case 1 above, as the construction of the root cover never groups them together). Atoms which share some dependencies (i.e., atoms whose predicates depend on one another) get unioned in fragments of the form 2 above. □

Example 14 (**Root cover**). *On the query and TBox from Example 11, the root cover is C_2 from Example 13;* `worksWith`(x, y) *and* `supervisedBy`(z, y) *are in the same C_2 fragment because* `worksWith` *depends on* `supervisedBy` *(cf. Example 12).*

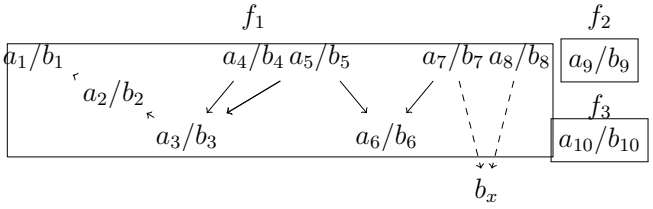

Figure 4.2: Sample C_{root} cover.

Example 15 (**Complex root cover**). *Figure 4.2 depicts a possible C_{root} cover of a 10-atoms query; the cover has 3 fragments, each shown in a rectangle. Every a_i/b_i denotes a query atom a_i whose predicate is b_i; a plain arrow from a node to another denotes that the predicate of the first depends on the predicate of the second. The predicate b_x appears in the TBox but does not appear in the query. In this example, b_1, b_2, b_4 and b_5 depend on b_3; b_5 and b_7 depend on b_6; b_7 and b_8 on b_x etc.. Fragment f_1 corresponds to case 2 of Lemma 1, while fragments f_2 and f_3 correspond to its first case.*

Proposition 4.4.1 states that C_{root} has the maximal number of fragments (equivalently, it has the smallest fragments) among all the safe covers for q and $\mathcal{T}$; its proof is based on Lemma 1.

Proposition 4.4.1 (Minimality of C_{root} fragments). *Let C_{root} be the root cover for q and $\mathcal{T}$, and C be another safe cover. For any fragment $f \in C_{\text{root}}$, and atoms $a_i, a_j \in f$, there exists a fragment $f' \in C$ such that $a_i, a_j \in f'$, in other words: any pair of atoms together in C_{root} are also together in C.*

Proof. For ease of explanation, in the proof, we rely on the graphical directed graph representation used in Figure 4.2 for dependencies between the predicates appearing in the atoms of a cover and/or other predicates from the KB.

Because f holds at least a_i and a_j, it must be a fragment of form 2, as stated in Lemma 1. It follows, thus, that in f there exists what we call an *extended path* e, going from a_i to a_j following the dependency edges either from source to target, or in the opposite direction; in other words, e alternately moves "up (or down) then down (or up)" a certain number of times in the fragment.

If e only contains edges in the same direction (either all are $\rightarrow$ or all are $\leftarrow$), it follows immediately that a_i and a_j are in the same fragment of C.

In the contrary case, there must exist some predicates in the TBox $b_1, \ldots, b_m$, $m \geq 1$, and some f atoms $a_1, \ldots, a_{m-1}$ defining an extended path e from a_i to a_j in f, as follows:

1. $a_i \rightarrow \ldots \rightarrow b_1$ (a $\rightarrow$ path segment), or b_1 is the predicate used in a_i;
2. b_k ($1 \leq k < m-1$), is the predicate used in a_l ($k \leq l < m-1$), or $b_k \leftarrow \ldots \leftarrow a_l$ (a $\leftarrow$ path segment);
3. $b_k \leftarrow \ldots \leftarrow a_l$ (a $\leftarrow$ path segment), with $1 \leq k < m-1$ and $k \leq l < m-1$, and $a_l \rightarrow \ldots \rightarrow b_{k+1}$. :
4. b_{m-1} is the predicate used in a_{m-1} or $b_{m-1} \leftarrow \ldots \leftarrow a_{m-1}$, then $a_{m-1} \rightarrow \ldots \rightarrow b_m$;
5. b_m is the predicate used in a_j or $b_m \leftarrow \ldots \leftarrow a_j$

Observe that items (2) and (3) can repeat (alternately) until b_{m-1} is reached.

Since C is safe, a_i and a_1 must appear in the same fragment in C (and only in that fragment), because they both depend on b_1.

For $1 \leq i \leq m-2$, a_i must appear in the same fragment as a_{i+1} (and only there), given that they both depend on b_i.

Since C is safe, a_j and a_{m-1} must appear in the same fragment of C (and only there).

From the above follows that $\{a_i, a_1, \ldots, a_{m-1}, a_j\}$ are all in the same fragment of C, which contradicts our hypothesis. □

From Proposition 4.4.1, we obtain:

Theorem 4.4.2 (Safe cover space)**.** *Let C be a safe cover and f one of its fragments. Then, f is the union of some fragments from C_{root}.*

Proof. Suppose that f is not a union of some fragments from C_{root}, and let us show a contradiction. In this case, f necessarily contains a strict, non-empty subset of a fragment of C_{root}. It follows that there are two atoms whose predicates depend on a common concept or role name w.r.t. $\mathcal{T}$ (as they were together in the fragment of C_{root}) that are not in a same fragment of C. Therefore C is not a safe cover, a contradiction. □

Safe cover lattice. Theorem 4.4.2 entails that the safe covers of a query q form a *lattice*, denoted $\mathcal{L}_q$, whose precedence relationship is denoted $\prec$, where $C_1 \prec C_2$ iff each fragment of C_2 is a union of some fragments of C_1. The lattice has as lower bound

the single-fragment cover, and as upper bound the *root* cover. For convenience, we also use $\mathcal{L}_q$ to denote the set of all safe covers.

The size of the safe cover lattice is bounded by the number of partitions of the fragments in C_{root}, i.e., by the number of partitions of the query atoms[4], a.k.a. the Bell number B_n for a query of n atoms; the bound occurs when there is no dependency between the atom predicates.

4.4.2 Generalized covers optimization space

A dependency-rich TBox leads to few, large fragments in C_{root}, thus to a relatively small number of alternative cover-based reformulations. In this section, we explore a notion of *generalized covers*, and propose a method for deriving FOL query reformulations from such covers. This enlarges our space of alternatives and thus potentially leads to a better cost-based choice of reformulation.

We call *generalized fragment* of a query q and denote $f\|g$ a pair of q atom sets such that $g \subseteq f$. A *generalized cover* is a set of generalized fragments $C = \{f_1\|g_1, \ldots, f_m\|g_m\}$ of a query q such that $\cup_{1\leq i\leq m} f_i$ is the set of atoms of the query, and no f_i is included in f_j for $1 \leq i \neq j \leq m$.

To a generalized fragment $f\|g$ of a generalized cover C, we associate:

Definition 4.4.2 (Generalized fragment query of a CQ)**.** *The* generalized fragment query $q_{|f\|g}$ *of* q *w.r.t.* C *is the subquery whose body consists of the atoms in* f*, and whose free variables are the free variables of* q *appearing in the atoms of* g*, plus the variables appearing in an atom of* g *that are shared with some atom in* g'*, for some other generalized fragment* $f'\|g'$ *of* C*.*

In a generalized fragment query, atoms from $f \setminus g$ only *reduce (filter)* the answers, without adding variables to the head. In particular, if $f = g$, $q_{|f\|g}$ coincides with the regular fragment query (Definition 4.2.2).

Given a generalized cover, the *generalized cover-based reformulation of a query* q is the FOL query

$$q^g(\bar{x}) \text{:-} \bigwedge_{i=1}^{m} q^{\texttt{FOL}}_{|f_i\|g_i}$$

if q^g is a FOL reformulation.

If $f_i = g_i$ for all the fragments $f_i\|g_i$, the generalized cover-based reformulation coincides with the regular cover-based one (Definition 4.2.3). As for simple cover-based reformulations, if fragments are reformulated into UCQs, the reformulated query is a JUCQ, whereas if they are reformulated into USCQs, the reformulated query is a JUSCQ.

[4]See https://oeis.org/A000110.

The introduction of extra atoms in generalized fragments is reminiscent of the classical semijoin reducers [19], whereas one computes $R(x, y) \bowtie_y S(y, z)$ by

$$(R(x, y) \ltimes_y \pi_y(S(y, z))) \bowtie_y S(y, z)$$

where $\ltimes_y$ denotes the left semijoin, returning every tuple from the left-hand side input that joins with the right-hand input. The semijoin filters ("reduces") the R relation to only those tuples having a match in S. If there are few distinct values of y in S, $\pi_y(S(y, z))$ is small, thus the $\ltimes_y$ operator can be evaluated very efficiently. Further, if only few R tuples survive the $\ltimes_y$, the cost of the $\bowtie_y$ operator likely decreases with the size of its input.

While the benefits of semijoins are well-known, there are many ways to introduce them in a given query, increasing the space of alternative plans to be considered by an optimizer. While some heuristics have been proposed to explore only some carefully chosen semijoin plans [92], we noted that RDBMS optimizers do not explore semijoin options, in particular for the very large queries resulting from the FOL reformulations of CQs. *Generalized fragments mitigate this problem by intelligently using semijoin reducers to fasten the evaluation of the* FOL *reformulation by the RDBMS.*

Generalized search space. We now define the space $\mathcal{G}_q$ of generalized covers for a given query q, based on the safe cover set $\mathcal{L}_q$. A generalized cover $C = \{f_1 \| g_1, \ldots, f_m \| g_m\}$ is part of $\mathcal{G}_q$ iff:

- The cover $C_s = \{g_1, \ldots, g_m\}$ is safe, i.e., $C_s \in \mathcal{L}_q$;
- For each $1 \leq i \leq m$, the atoms in f_i form a connected graph.

Note that an atom $a \in f$, for $f \| g \in C$, *has no impact on the head of the corresponding generalized fragment query*; only the body of this query changes.

The size of $\mathcal{G}_q$ obviously admits that of $\mathcal{L}_q$ as a lower bound. For a query q of n atoms, a upper bound is $B_n * n * 2^{n-1}$, where B_n is the n-th Bell number: for each safe cover C (of which there are at most B_n, see the previous section), each of the n atoms may, in the worst case, be added or not to all the fragments to which it does not belong. In the worst case, there are $n - 1$ such fragments.

The core result allowing us to benefit of the performance savings of generalized covers in order to efficiently answer queries is:

Theorem 4.4.3 ($\mathcal{G}_q$ cover-based query answering)**.** *The reformulation of a query q based on $\mathcal{T}$ and a generalized cover $C \in \mathcal{G}_q$ is a* FOL *reformulation of q w.r.t. $\mathcal{T}$.*

Proof. The proof follows from that of Theorem 4.3.1. It relies on the fact that, given a safe cover $C = \{g_1, \ldots, g_m\}$ of q and a generalized cover $C' = \{f_1 || g_1, \ldots, f_m || g_m\}$ of q, the queries $q(\bar{x})$:- $\bigwedge_{i=1}^{m} q_{|g_i}$ and $q'(\bar{x})$:- $\bigwedge_{i=1}^{m} q_{|f_i||g_i}$ are equivalent, though each $q_{|g_i}$ subsumes $q_{|f_i||g_i}$. Indeed, q' is obtained from q by duplicating atoms already present in q, thus q^e only adds redundancy w.r.t. q, hence remains equivalent to it. □

Example 16 (Generalized cover-based reformulation). *Recall the query and KB from Example 11. Let* $f_0 = \{\texttt{PhDStudent}(x)\}$ *and* $f_1 = \{\texttt{worksWith}(x, y), \texttt{supervisedBy}(z, y)\}$ *be the two fragments of the root cover* C_{root}*. Consider also the generalized cover* $C_3 = \{f_1 \| f_1, f_2 \| f_0\}$*, where* $f_2 = \{\texttt{PhDStudent}(x), \texttt{worksWith}(x, y)\}$*.*

The generalized fragment query $q_{|f_1\|f_1}$ *of* q *w.r.t.* C_3 *is the subquery* $q_{|f_1\|f_1}(x)$*:-* $\texttt{worksWith}(x, y) \wedge \texttt{supervisedBy}(z, y)$*. Observe that* y *is not a free variable of* $q_{|f_1\|f_1}$*, as it is neither a free variable of* q *nor a variable in* f_0*, whereas* $f_2 \| f_0$ *is the only other fragment in the cover* C_3*.*

The generalized fragment query $q_{|f_2\|f_0}$ *of* q *w.r.t.* C_3 *is the subquery* $q_{|f_2\|f_0}(x)$*:-* $\texttt{PhDStudent}(x) \wedge \texttt{worksWith}(x, y)$*. Again, note that* y *is not a (free) variable of* f_0*, and therefore it is not a free variable of* $q_{|f_2\|f_0}$*.*

Then, the generalized cover-based reformulation corresponding to C_3 *is the* FOL *query:*

$$q^g(x) \text{:-}\ q^{\texttt{FOL}}_{|f_1\|f_1}(x) \wedge q^{\texttt{FOL}}_{|f_2\|f_0}(x)$$

where:

$$\begin{aligned} q^{\texttt{FOL}}_{|f_1\|f_1}(x) \text{:-}\quad & (\texttt{worksWith}(x, y) \wedge \texttt{supervisedBy}(z, y)) \\ & \vee \texttt{supervisedBy}(x, y) \vee \texttt{Graduate}(x) \end{aligned}$$

$$\begin{aligned} q^{\texttt{FOL}}_{|f_2\|f_0}(x) \text{:-}\quad & (\texttt{PhDStudent}(x) \wedge \texttt{worksWith}(x, y)) \\ & \vee (\texttt{PhDStudent}(x) \wedge \texttt{supervisedBy}(x, y)) \\ & \vee (\texttt{PhDStudent}(x) \wedge \texttt{Graduate}(x)) \end{aligned}$$

Applying $\texttt{supervisedBy} \sqsubseteq \texttt{worksWith}$ *to* $q_{|f_1\|f_1}$ *leads to:*

$$\begin{aligned} & (\texttt{worksWith}(x, y) \wedge \texttt{supervisedBy}(z, y)) \\ & \vee (\texttt{supervisedBy}(x, y) \wedge \texttt{supervisedBy}(x, y)) \\ \equiv\ & (\texttt{worksWith}(x, y) \wedge \texttt{supervisedBy}(z, y)) \\ & \vee \texttt{supervisedBy}(x, y) \end{aligned}$$

Then, applying $\texttt{Graduate} \sqsubseteq \exists\texttt{supervisedBy}$*, we obtain the reformulation of* $q_{|f_1\|f_1}$ *w.r.t. TBox* $\mathcal{T}$*, i.e.,* $q^{\texttt{FOL}}_{|f_1\|f_1}$*. Similarly, applying to* $q_{|f_2\|f_0}$ *the constraint* $\texttt{supervisedBy} \sqsubseteq \texttt{worksWith}$ *and subsequently* $\texttt{Graduate} \sqsubseteq \exists\texttt{supervisedBy}$ *leads to* $q^{\texttt{FOL}}_{|f_2\|f_0}$*.*

Note that $ans(q^g, \langle \emptyset, \mathcal{A} \rangle) = \{Damian\} = ans(q, \langle \mathcal{T}, \mathcal{A} \rangle)$*.*

4.4.3 Cost-based cover search algorithms

Our first algorithm, **EDL (Exhaustive Covers for DL)**, starts from C_{root} and builds all $\mathcal{L}_q$ covers by unioning fragments, and all $\mathcal{G}_q$ covers by adding atoms (Algorithm 3).

Algorithm 3: Exhaustive Cover Search for DL-Lite$_\mathcal{R}$ (**EDL**)

Input : CQ $q(\bar{x})$:- $a_1 \wedge \cdots \wedge a_n$, KB $\mathcal{K}$
Output: Best cover for reformulating for q

1 $\mathcal{L}_q \leftarrow \emptyset; \mathcal{G}_q \leftarrow \emptyset$;
2 $F \leftarrow C_{\text{root}}$; $C_{best} \leftarrow C_{\text{root}}$;
3 **foreach** $P = \{s_1, \ldots, s_{|P|}\}$ *distinct partition of* F *s.t. the atoms of all the fragments in each set* s_i*, for* $1 \leq i \leq |P|$*, are connected* **do**
4 $C_P \leftarrow \emptyset$;
5 **foreach** *fragment set* $s_i = \{f_i^1, \ldots, f_i^{n_i}\} \in P$ **do**
6 $C_P \leftarrow C_P \cup \{f_i^1 \cup \cdots \cup f_i^{n_i}\}$;
7 $\mathcal{L}_q \leftarrow \mathcal{L}_q \cup \{C_P\}$;
8 **if** C_P *estimated cost* $< C_{best}$ *estimated cost* **then**
9 $C_{best} \leftarrow C_P$;
10 **foreach** *safe cover* $C = \{f_1, f_2, \ldots, f_n\} \in \mathcal{L}_q$ **do**
11 $C' \leftarrow \{f_1||f_1, f_2||f_2, \ldots, f_n||f_n\}$;
12 **foreach** $f||g \in C', a_i \notin f$ *such that* a_i *shares a variable with an* f *atom* **do**
13 $C'' \leftarrow C' \setminus \{f||g\} \cup \{f \cup \{a_i\}||g\}$;
14 $\mathcal{G}_q \leftarrow \mathcal{G}_q \cup \{C"\}$;
15 **if** C'' *estimated cost* $< C_{best}$ *estimated cost* **then**
16 $C_{best} \leftarrow C''$;
17 **return** C_{best};

The second one, **GDL (Greedy Covers for DL)** (Algorithm 4) works in a greedy fashion. It is based on exploring, from a given cover C, the set of possible next moves (lines 2-4 and 5-7); these are all the covers that may be created *from* C by unioning two of its fragments or by enlarging one of its fragments, i.e., turning a fragment $f\|g$ into $f \cup \{a\}\|g$ for some query atom a sharing a variable with f. The best one seen at a given point (w.r.t. the estimated evaluation cost) is kept as the selected next move in the $move$ variable. Lines 2 and 5, respectively, assign to $move$ the move where the union of fragments f_1 and f_2 is performed on cover C, and the move enlarging fragment f with atom a on cover C.

At the end of this exploration step (line 9), the best move is applied, leading to the new best cover C from which the next exploration step starts. The exploration stops when no possible next move improves the cost of the currently selected best cover C.

When unioning two fragments, ϵ decreases if the resulting fragment is more selective

Algorithm 4: Greedy Cover Search for DL-Lite$_\mathcal{R}$ (**GDL**)

Input : CQ $q(\bar{x})$:- $a_1 \wedge \cdots \wedge a_n$, KB $\mathcal{K}$
Output: Best cover for reformulating q

1 $C \leftarrow C_{\text{root}}$; $move \leftarrow \emptyset$;
2 **foreach** $f_1, f_2 \in C$ **do**
3 **if** *($move$ is empty and $C.union(f_1, f_2)$ est. cost $\leq C$ est. cost) or ($C.union(f_1, f_2)$ est. cost $< apply(move)$ est. cost)* **then**
4 $move \leftarrow (C, f_1, f_2)$;
5 **foreach** $f \in C, a \in q$ *s.t. a is connected to f* **do**
6 **if** *($move$ is empty and $C.enlarge(f, a)$ est. cost $\leq C$ est. cost) or ($C.enlarge(f, a)$ est. cost $< apply(move)$ est. cost)* **then**
7 $move \leftarrow (C, f, \{a\})$;
8 **while** $move \neq \emptyset$ **do**
9 $C \leftarrow apply(move)$; // the cover obtained from that move
10 $move \leftarrow \emptyset$;
11 // Gather move starting from C as was done at lines 2–7 above
12 **return** C;

than the two fragments it replaces. Therefore, the RDBMS may find a more efficient way to evaluate the query reformulation of this fragment, and/or its result may be smaller, making the evaluation of q^{FOL} based on the new cover C faster. When adding an atom to an extended fragment, ϵ decreases if the conditions are met for the semijoin reducer to be effective (Section 4.4.2). In our context, many such opportunities exist, as our experiments show.

4.5 Experimental evaluation

We implemented our cover-based query answering approach in Java 8; the source code has about 10.000 lines, including the statistics and cost estimation (see below).

4.5.1 Experimental settings

RDBMSs and data layout. First, we used **PostgreSQL v9.3.2** to store the data and evaluate FOL query reformulations. Our *first data layout* within Postgres stored all the assertions into a single triple table (where each $C(x) \in \mathcal{A}$ leads to a triple x *type* C and each $R(a, b) \in \mathcal{A}$ leads to a triple $a\ R\ b$), and built all six three-attribute indexes on this triple table [83]. Our *second data layout* stored a unary table for each concept and a binary table for each role, and built all one- and two-attribute indexes, respectively,

on those tables. Our tests showed that the second layout significantly outperformed the first; this is not surprising, as smaller tables lead to better performance, at the same time it reduces the number of query conditions (as some of them are encoded by accessing a certain table). Thus, *for Postgres, we only report results based on the layout featuring role and concept tables*.

Second, we used the **IBM DB2 Express-C 10.5**. We chose it because (i) we previously (Chapter 3) found out (and confirm below) that it evaluates large FOL reformulations better than Postgres, and (ii) it provides a relatively recent, smart storage layout for RDF graphs [21], intelligently bundling assertions into a small set of tables with potentially many attributes, so that the roles to which an individual participates are stored, to the extent possible, in the same tuple. This reduces the number of joins needed for query evaluation, and has been shown [21] to improve query performance. However, DB2 does not support reasoning, i.e., it only provides query evaluation. For DB2, *we report results based on the concept and role tables (denoted simple layout) and on the RDF layout* of [21].

In the simple layout, as customary in efficient Semantic Web data management systems, e.g., [83], facts are *dictionary-encoded* into integers, prior to storing them in the RDBMS. The TBox and predicates dependencies are stored in memory.

Hardware. The database servers ran on an 8-core Intel Xeon E5506 2.13 GHz machine with 16GB RAM, using Mandriva Linux r2010.0.

Datasets, queries, and reformulation engine. We used two LUBM$^{\exists}_{20}$ benchmark KBs, comprising a DL-Lite$_{\mathcal{R}}$ TBox and two ABoxes of 15 million, respectively, 100 million facts, obtained using the EUDG data generator [76]. The TBox consists of 34 roles, 128 concepts and 212 constraints.

We devised a set of 13 CQs against this knowledge base, shown in the Section A.2, Tables A.5 and A.6. The queries have between 2 and 10 atoms, with an average of 5.77 atoms. Their UCQ reformulations are unions of 35 to 667 CQs, 290.2 on average. This parameter characterizing the query can be seen as a (rough) measure of the complexity of its reformulation; it is shown in Table A.5 in the column $|q^{\text{UCQ}}|$.

We relied on the RAPID [36] CQ-to-UCQ reformulation tool to reformulate (simple or generalized) fragment queries (Definitions 4.2.2 and 4.4.2); any other CQ-to-UCQ or CQ-to-USCQ reformulation technique could have been used instead.

Cost estimation function. For the cost function estimation ϵ, we first used the RDBMS cost estimation for the SQL translation of each candidate FOL reformulation produced by our algorithms. For Postgres, we obtained this using `explain`[5], while for DB2 we used `db2expln`[6].

[5] See http://www.postgresql.org/docs/9.1/static/sql-explain.html.

[6] See http://www-01.ibm.com/support/knowledgecenter/SSEPGG_10.5.0/com.ibm.db2.luw.admin.cmd.doc/doc/r0005736.html.

Further, for the simple layout, we implemented our own Java-based cost estimation, based on statistics on the stored data (cardinality and number of distinct values in each stored table attribute), and on the uniform distribution and independent distributions assumptions. Better RDF cardinality estimation techniques such as [81] may be used to improve the accuracy of our cost model.

For the sake of completeness, the next Section details how we compute the cost of evaluating a JUCQ (reformulation) sent to an RDBMS; this presentation is borrowed and adapted from Section 3.2.1.

4.5.2 Our cost estimation function

A JUCQ is a join of UCQs subqueries of the form: $q^{\texttt{JUCQ}}(\bar{x})$:- $q_1^{\texttt{UCQ}} \bowtie \cdots \bowtie q_m^{\texttt{UCQ}}$.
The evaluation cost of $q^{\texttt{JUCQ}}$ is

$$\begin{aligned} c(q^{\texttt{JUCQ}}) = \; & c_{\texttt{db}} + \sum_{1 \leq i \leq m} \left(c_{eval}(q_i^{\texttt{UCQ}}) \right) \\ & + \sum_{i, 1 \leq i \leq m, i \neq k} \left(c_{mat}(q_i^{\texttt{UCQ}}) \right) + c_{join}(q_{i, 1 \leq i \leq m}^{\texttt{UCQ}}) \\ & + c_{unique}(q^{\texttt{JUCQ}}) \end{aligned} \tag{4.1}$$

reflecting:

- (i) the fixed overhead of connecting to the RDBMS $c_{\texttt{db}}$;
- (ii) the cost to *evaluate* each of its UCQ sub-queries $q_i^{\texttt{UCQ}}$;
- (iii) the *materialization* costs: the SQL query corresponding to a JUCQ may have many sub-queries. At execution time, some of these subqueries will have their results materialized (i.e., stored in memory or on disk) while at most one sub-query will be executed in pipeline mode. We assume without loss of generality, that the largest-result sub-query, denoted $q_k^{\texttt{UCQ}}$, is the one pipelined (this assumption has been validated by our experiments so far);
- (iv) the cost to *join* these sub-query results; and
- (v) the cost of *eliminating* duplicates, in order to enforce our desired set semantics: from the results of each $q_i^{\texttt{UCQ}}$, and from the final results, by means of DISTINCT clauses. We found that this two-level elimination of duplicates lead to the best performance overall. Note that removing duplicates in the results of $q_i^{\texttt{UCQ}}$ does not break an evaluation pipeline, as those results were materialized anyway.

In the above, duplicates are eliminated because existing reformulation algorithms (and accordingly, our work) operate under *set semantics*.

Notations. For a given query q over a database db, we denote by $|q|_t$ the estimated number of tuples in q's answer set. Also, $q_{|\{a_i\}}$ stands for the restriction of q to its i-th atom. Using the notations above, the number of tuples in the answer set of $q_{|\{a_i\}}$ is denoted $|q_{|\{a_i\}}|_t$.

Duplicate elimination costs. Assuming duplicate elimination is implemented by hashing, we estimate the cost of eliminating duplicate rows from an SQL query $q^{\texttt{JUCQ}}$ and/or $q_i^{\texttt{UCQ}}$) as:

$$c_{unique}(q^{\texttt{JUCQ}}) = c_l \times |q^{\texttt{JUCQ}}|_t$$

where c_l is the CPU and I/O effort involved in sorting the results.

When the results are large enough to require disk merge sort, we estimate the cost of eliminating duplicate rows from $q^{\texttt{JUCQ}}$ (and $q_i^{\texttt{UCQ}}$ as a particular case) result as:

$$c_{unique}(q^{\texttt{JUCQ}}) = c_k \times |q^{\texttt{JUCQ}}|_t \times \log |q^{\texttt{JUCQ}}|_t$$

where c_k is the CPU and I/O effort involved in (disk-based) sorting the results.

UCQ **evaluation costs** are estimated by summing up the estimated costs of the CQs:

$$c_{eval}(q_i^{\texttt{UCQ}}) = \sum_{q^{\texttt{CQ}} \in q_i^{\texttt{UCQ}}} c_{eval}(q^{\texttt{CQ}})$$

The cost of evaluating *one* conjunctive query $c_{eval}(q^{\texttt{CQ}})$, where $q^{\texttt{CQ}}(\bar{x})$:- $a_1 \wedge \cdots \wedge a_n$, through the RDBMS is estimated by analyzing the selections (known attribute values) in each atom of $q^{\texttt{CQ}}$, estimating (exactly – see below) how many triples match these atoms, and *estimating the data access costs and the join costs together*. The data layouts we consider feature indexes on the relations storing class and role instances; as soon as the query selections and joins allow it, the RDBMS heavily relies on the indexes to simultaneously join and access the data i.e., the plan chains index-based accesses and index-based joins. Assuming efficient join algorithms such as hash- or merge-based etc. are available [88], the join-only cost of $q^{\texttt{CQ}}$ is linear in the total size of its inputs:

$$c_{join}(q^{\texttt{CQ}}) = c_j \times \sum_{a_i \in q^{\texttt{CQ}}} |q^{\texttt{CQ}}_{|\{a_i\}}|_t$$

where c_j is a constant factor representing per-tuple join effort. Therefore, we have:

$$c_{eval}(q_i^{\texttt{UCQ}}) = (c_t + c_j) \times \sum_{q^{\texttt{CQ}} \in q_i^{\texttt{UCQ}}} \sum_{a_i \in q^{\texttt{CQ}}} |q^{\texttt{CQ}}_{|\{a_i\}}|_t \tag{4.2}$$

where c_t is a constant representing the per-tuple I/O (access) effort.

UCQ **join cost**. As before, we consider the join cost to be linear in the total size of its inputs:

$$c_{join}(q^{\texttt{UCQ}}_{i,1\leq i\leq m}) = c_j \times \sum_{i=1}^{m} \sum_{q^{\texttt{CQ}}\in q_i^{\texttt{UCQ}}} \sum_{a_i\in q^{\texttt{CQ}}} |q^{\texttt{CQ}}_{|\{a_i\}}|_t \tag{4.3}$$

UCQ **materialization cost**. Finally, we consider the materialization cost associated to a query q is $c_m \times |q|_t$ for some constant c_m:

$$c_{mat}(q^{\texttt{UCQ}}_{i,1\leq i\leq m, i\neq k}) = c_m \times \sum_{i=1, i\neq k}^{m} \sum_{q^{\texttt{CQ}}\in q_i^{\texttt{UCQ}}} \sum_{a_i\in q^{\texttt{CQ}}} |q^{\texttt{CQ}}_{|\{a_i\}}|_t \tag{4.4}$$

where $q_k^{\texttt{UCQ}}$ is the largest-result sub-query, and the one which is picked for pipelining (and thus not materialized).

Injecting the equations 4.2, 4.3 and 4.4 into the global cost formula 4.1 leads to the estimated cost of a given JUCQ. This formula relies on estimated cardinalities of various subqueries of the JUCQ, as well as on the system-dependent constants $c_{\texttt{db}}$, c_l, c_k, c_j, c_t and c_m, which we determine by running a set of simple *calibration queries* (inspired by the approach of [45]) on the RDBMS being used; calibration details are straightforward and we omit them here.

For what concerns cardinality estimations, as in [83], RDBMS statistics provide, for each query atom, the exact number of triples matching it. Subsequently, textbook formulas are used to estimate the cardinality of more complex subqueries, based on statistics on the minimum and maximum value, and the number of distinct values in each attribute. We make the simple assumptions of uniform distribution of each attribute, and independent distributions among attributes. Any more refined RDF cardinality estimation technique, e.g., [81], could be used to improve the estimation accuracy.

4.5.3 Search space and EDL running time

We first studied the number of covers in $\mathcal{L}_q$ and $\mathcal{G}_q$ (recall Section 4.4). Our workload features some queries of 2 atoms, and the immediately larger ones have 6; we quickly realized that the number of generalized covers is prohibitively high for 6 or more atoms. To study this more closely, we derived from Q_1 a set of queries A_i, $3 \leq i \leq 6$, each of which is a star-join of i atoms on a common subject; in particular, A_6 is Q_1. Star queries are frequent over Semantic Web Data, as noted e.g., in [107, 21]. The sizes of the resulting search spaces are reported in Table 4.2; for A_6 we stopped the search at 20.003 generalized covers (there were more). This demonstrates that exploring the full $\mathcal{G}_q$ space is in general not feasible, as the overhead of examining so many options is

prohibitive. Thus, in the sequel, we do not use EDL for our tests, as it is impractical beyond (very) small queries. Table 4.2 also shows the number of covers explored by the greedy GDL: these grow very moderately with the query size.

Finally, for A_3-A_6, *the running times of the best reformulation found by EDL and GDL (limited at* 20.000 *covers for* A_6*) coincided*. In general this is not guaranteed, but it is still an encouraging indicator of the good options found by GDL.

4.5.4 Evaluation time of reformulated queries

Figure 4.3 depicts the evaluation time, using Postgres with the simple layout, of four FOL reformulations:

1. the UCQ produced by the RAPID [36] reformulation engine;
2. the JUCQ reformulation based on C_{root};
3. a JUCQ reformulation corresponding to a best-performing (safe or generalized) cover, found by our algorithm GDL, using Postgres' cost estimation (*RDBMS*);
4. a JUCQ reformulation corresponding to a best-performing (safe or generalized) cover, found by our algorithm GDL, using our cost estimation (*ext*).

GDL running time is not reported in these graphs (see Section 4.5.5). We first analyze the top graph corresponding to LUBM$^{\exists}_{20}$ 15 million triples. It shows, first, that the UCQ reformulation is inefficient (one order of magnitude slower than the best reformulation found, e.g., for Q_5 and Q_9). Second, the cover derived from C_{root} may also be very inefficient, in some cases (Q_6-Q_8, Q_{13}) much worse than the UCQ. These are both very large and complex queries; Figure 4.3 demonstrates that Postgres' optimizer called directly on the fixed-form reformulation may performed quite poorly. The GDL-selected covers, in contrast, *lead to the best-performing reformulations for all queries (often by an order of magnitude)*. Thus, our cost-based approach helps ask *the RDBMS the optimization question it can best answer*, among its equivalent formulations from the search space $\mathcal{G}_q$. Striking exceptions are Q_9, Q_{10} which have *both* many atoms and complex reformulations, and Q_{11} which has 2 atoms but the maximum number (667) of reformulations. Here, the GDL reformulations selected using the RDBMS cost model perform very poorly, whereas the ones based on our own cost estimation are much faster. This may be because Postgres takes drastic shortcuts when estimating the cost of an extremely large query; in contrast, our cost estimation treats uniformly queries of all sizes. Recall that *Postgres' optimizer always has the last word in chosing how to evaluate the reformulation* we select, using its own cost model. Thus, the difference can only be attributed to the cost estimation.

The bottom graph in Figure 4.3 corresponds to LUBM$^{\exists}_{20}$ 100 million triples; note the logarithmic y axis. Overall, the findings are the same: the UCQ and (especially in this

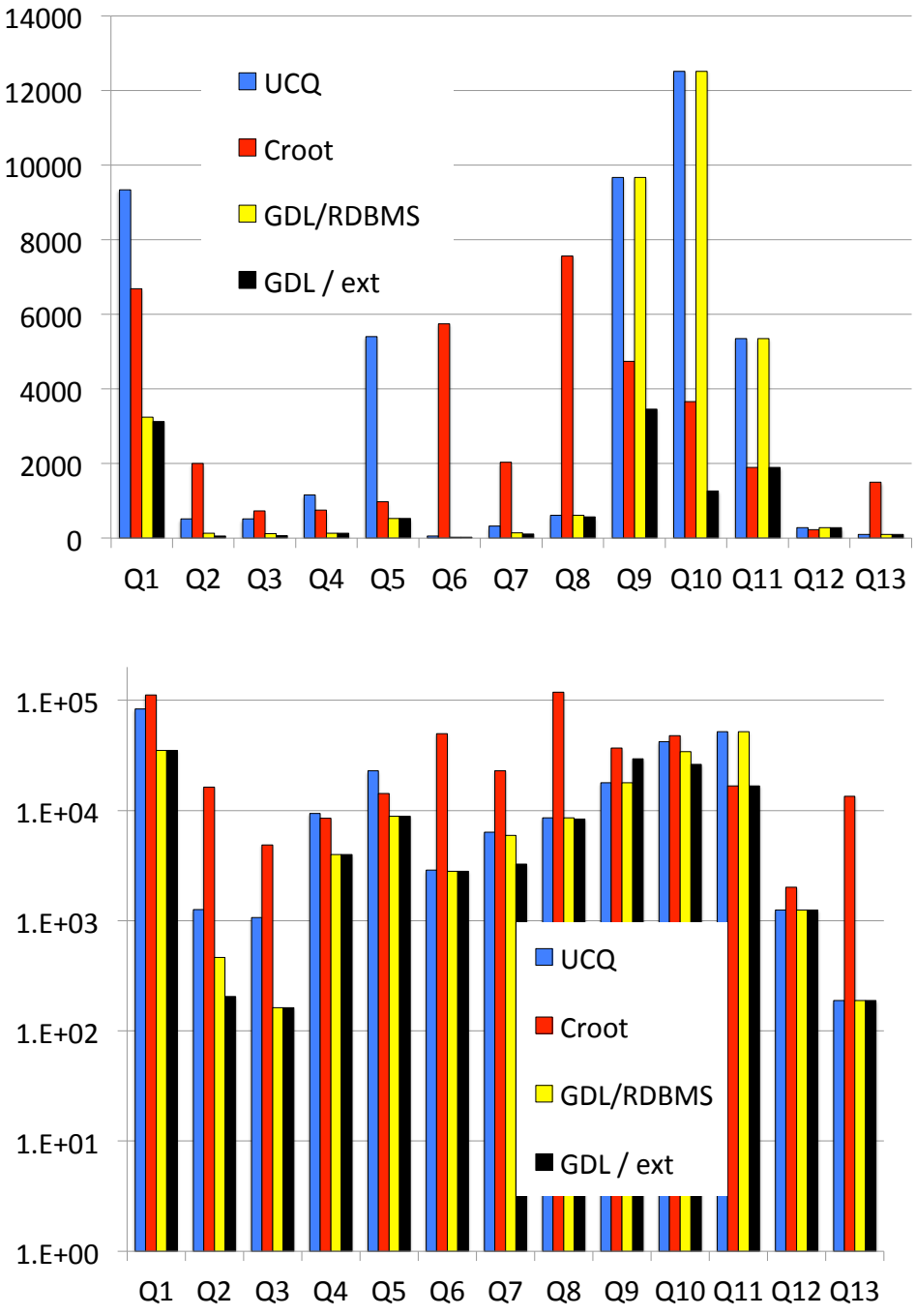

Figure 4.3: Evaluation time (ms) on Postgres on LUBM$^{\exists}_{20}$ 15 million (top) and 100 million (bottom) triples.

case) the C_{root} reformulation perform poorly, while those picked by GDL are faster than the standard UCQ by a factor of up to 6.6 (Q_3).

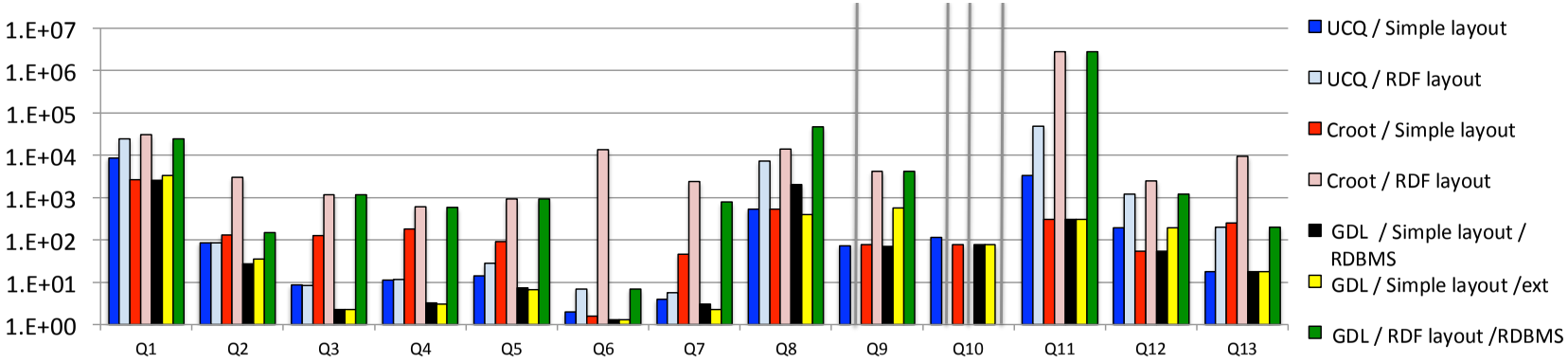

Figure 4.4: First caption

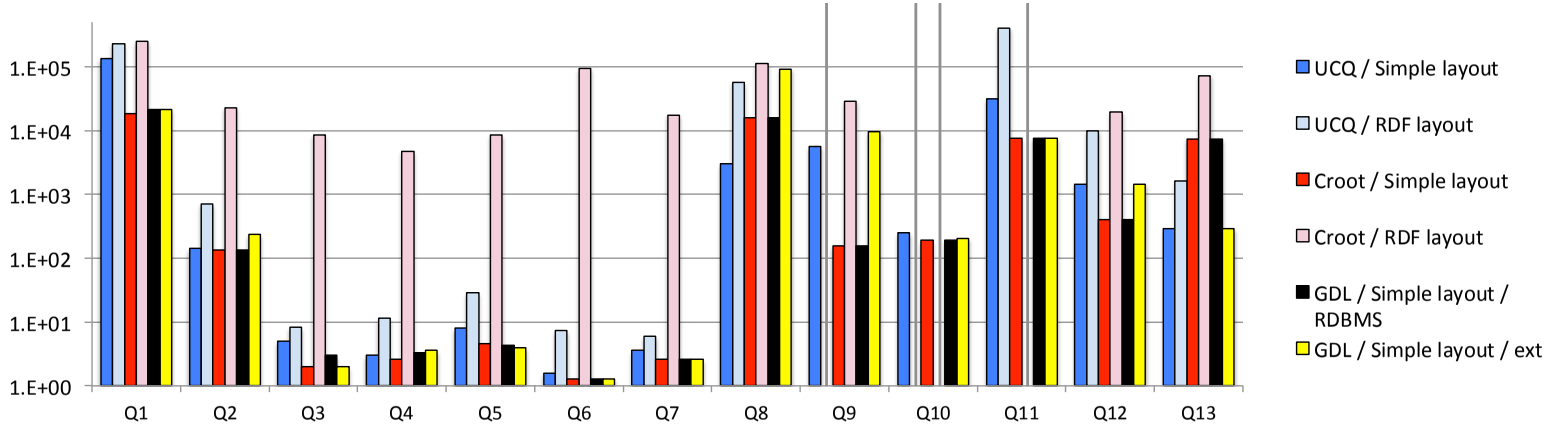

Figure 4.5: Evaluation time (ms) on DB2 and LUBM$^{\exists}_{20}$ 15 million (top) and 100 million (bottom) triples.

Query	A_3	A_4	A_5	A_6
$\|\mathcal{L}_q\|$	2	7	71	93
$\|\mathcal{G}_q\|$	4	67	5674	> 20000
$\mathcal{L}_q$ covers explored by GDL	2	5	11	18
$\mathcal{G}_q$ covers explored by GDL	4	12	27	59

Table 4.2: Search space sizes for queries A_3 to A_6.

Evaluation on DB2. The graph at the top of Figure 4.5 shows the evaluation time for DB2, on LUBM$^{\exists}_{20}$ 15 million triples, of seven reformulations: the same four which we ran on Postgres, to which we add, on the RDF layout [21]: the UCQ reformulation, the one based on C_{root}, and the ones selected by GDL with the help of the RDBMS cost model. We did not code a cost estimation corresponding to this RDF-specific store, since (i) an accurate model of data access costs under such a complex layout (determined by running a linear programming solver etc..) seemed very hard to attain outside the server and (ii) DB2's cost model performed similarly to (or better than) ours for all the GDL-selected covers, on the simple layout. Thus, replacing it with our own seemed unlikely to improve the performance. Note the logarithmic y axis of the graph.

First, note that five bars are missing (replaced by the vertical lines), one for Q_9 and four for Q_{10}. They all correspond to reformulations against the RDF layout. The server error was "The statement is too long or too complex. Current SQL statement size is 2,247,118" for the UCQ of Q_9, and the same error (with similar query sizes) in the other cases. This shows that *the cummulated impact of, first, the DB2RDF storage layout (which leads to* `IF... THEN... ELSE` *and nesting in the SQL query corresponding to a simple* CQ*), and second, of ontology-based reformulation, yields queries too large for evaluation*. For illustration, the SQL versions of Q_1 before and after UCQ reformulation on DB2's RDF store appear at http://bit.ly/1TqeVMA. In cases where DB2 handled them, the reformulations corresponding to the UCQ, C_{root} and GDL on the RDF layout performed very poorly, up to 1 (UCQ) or even 4 (C_{root}) orders of magnitude worse than the best reformulations identified. Thus, our (somehow unexpected) conclusion is that the RDF-specific layout, while interesting for CQ evaluation, is not the best alternative when evaluating queries issued from reformulation against an ontology.

Focusing only on the simple layout, we see that the cost-unaware UCQ and C_{root}-derived reformulations perform again poorly, while the GDL ones perform best and in many cases coincide. The two cost estimations behaved mostly the same, except that our estimation worked better for Q_8 and worse for Q_9. Overall, our chosen reformulations lead to performance gains of up to a factor of 9 w.r.t. the UCQ and/or C_{root} on the simple layout, for which we found DB2's cost estimation quite reliable.

At the bottom of Figure 4.5 we show the evaluation times on LUBM$^{\exists}_{20}$ 100 million triples for the first eight among the ten reformulations shown in the top graph (we gave up GDL on the RDF layout, given our experience on the smaller

dataset). The four execution errors (grey vertical lines) on the UCQ and C_{root} reformulations on the RDF layout are again due to overly large SQL queries. The first four alternatives are overall the worse, with C_{root} and at a lesser extent UCQ on the RDF layout performing very poorly. When focusing on the simple layout only, we notice that the cost-based reformulations improve over the simple UCQ performance by a factor of up to 36 (4.85 on average). There is an exception for Q_8, where the UCQ was best; in this case, both DB2's and our cost estimations were inaccurate, which we believe cannot be avoided in all cases. DB2's estimation lead to significantly better reformulations than ours for the queries Q_2, Q_8, Q_9 and Q_{12}, while our cost model was clearly better for Q_{13}. Overall, we found DB2's cost estimation more accurate than our own (while the opposite holds for Postgres). By inspecting query plans, we confirmed that DB2 and Postgres do not apply any CSE across union terms. The better performance of DB2 is likely due to efficient runtime support for repeated scans [69].

In all experiments presented in this section, GDL ran between 1 ms (for 2-atom queries) to 207 ms (for the larger Q_1); we discuss the running time of our optimization approach in more detail in Section 4.5.5.

Finally, always (when using our cost model) and about half of the time (with the RDBMS cost model), GDL picked a generalized cover. This confirms the interest of searching in the $\mathcal{G}_q$ space.

4.5.5 Time-limited GDL

Figure 4.6 shows the running time of algorithm GDL on the LUBM datasets of 15 million and 100 million triples, in four configurations: using no cost estimation (this artificial case where all costs are estimated to 0 was built to measure our algorithm's running time independently of the cost estimation time); using our own cost estimation (described in Section 4.5.2); using the cost estimation of Postgres; and finally, using the estimation of DB2. Note that the vertical axis is in logarithmic scale. The two graphs are similar, which is to be expected given that we measure optimization time, which is not (strongly) impacted by the data sizes. The times using Postgres' and DB2's cost estimations are not identical: internal heuristics in the Postgres and DB2 systems may have led to different plans shapes being explored for the different database sizes.

We make the following observations.

1. The running time of GDL without any cost estimation is very small, bounded by 23 ms.

2. Using our cost model has a discernible yet still small overhead, bringing the total running time of our optimization technique to about 207 ms.

3. Using Postgres' cost estimation time incurs a significant overhead, going up to 10, 100 and even (in the pathological case of Q_5) 1000 seconds, which is prohibitive

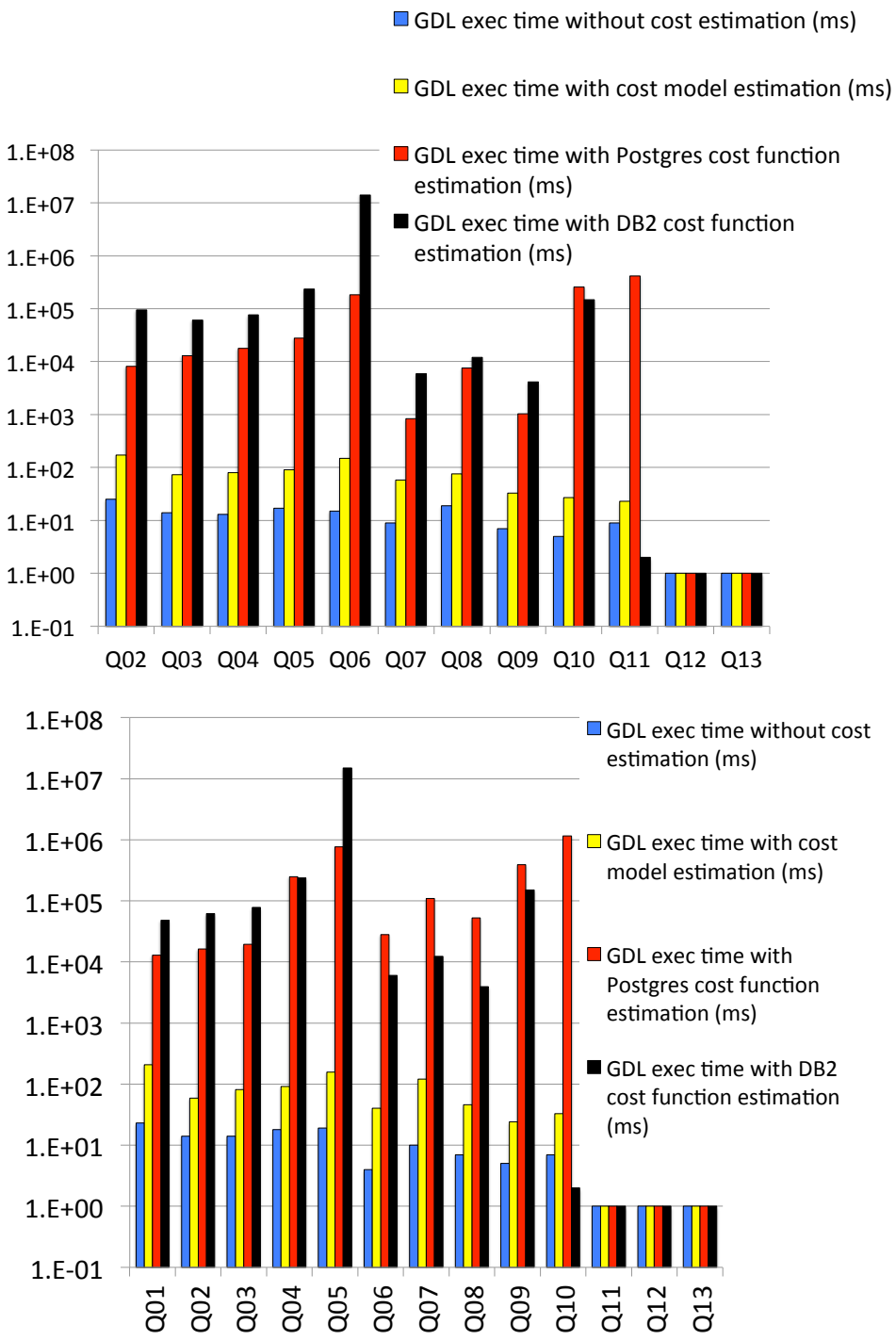

Figure 4.6: GDL running time (ms) on LUBM$^{\exists}$ 15 million (top) and 100 million (bottom) triples.

for an optimization step.

4. Using DB2's cost estimation is for many queries even more expensive. This is because the db2expln utility requires large queries to be written into an OS file given as parameter to the cost estimation, whose output must then be extracted from the detailed information db2expln returns. This is more expensive than Postgres' provided explainer functionality, which is accessible through the JDBC driver, without the need to make a runtime call from our Java optimizer code etc..

5. GDL including DBMS cost estimation is visibly correlated with the size of the query; note the peaks for Q_5, Q_9 and Q_{10}, which are the most complex (as shown in Table A.5).

The graphs show that it is clearly preferrable to run GDL in a context where the cost estimation function is accessible without a high overhead. This was the case when using our own estimation, while the estimations of Postgres and especially DB2 were harder for us to access.

Time-limited GDL. Therefore, we investigated a *time-limited* version of GDL, which was allowed to explore only during 20 ms. Figure 4.7 compares the running time of the cover found by GDL after only 20 ms, with that of the cover found by GDL allowed to run to completion. We see that the running times are very close for Postgres, and also generally close for DB2, demonstrating that interesting covers are quickly found. This is because on our queries, the strongest reduction (mostly through reducing intermediary result sizes) are identified early during the greedy search, thus most of the performance benefits can be reaped early on. More generally, this corresponds to the good behavior of greedy algorithms when there are very advantageous moves to be made. Thus, we find time-limited GDL performs well in practice, for a modest overhead.

Against the expectation, in some cases, the limited GDL performed better than the unlimited one (for instance, on Q_5 and Q_7 on DB2 in Figure 4.7). This is an accident due to our cost model; it turns out that in these cases, the longer search ended up recommending a state whose cost was slightly worse.

4.5.6 Experiment conclusions

Our experiments show that plain `UCQ` reformulation is evaluated poorly by both Postgres and DB2, even more so (or even fails) on DB2's RDF-specific data layout. On the simple layout, the fixed cover-based reformulation corresponding to the root cover C_{root} also performs very poorly. In contrast, GDL-selected reformulation improve over the `UCQ` in all 13 queries $\times 2$ systems $\times 2$ datasets but one, and they do so by up to a factor of 36. Our cost estimation helped w.r.t. Postgres' `explain`, but when using DB2, we find `db2explain`'s estimation more accurate overall.

The generalized cover space has prohibitive size, thus EDL is impractical. In contrast, our greedy GDL is efficient when used with a low-overhead cost estimation (such as

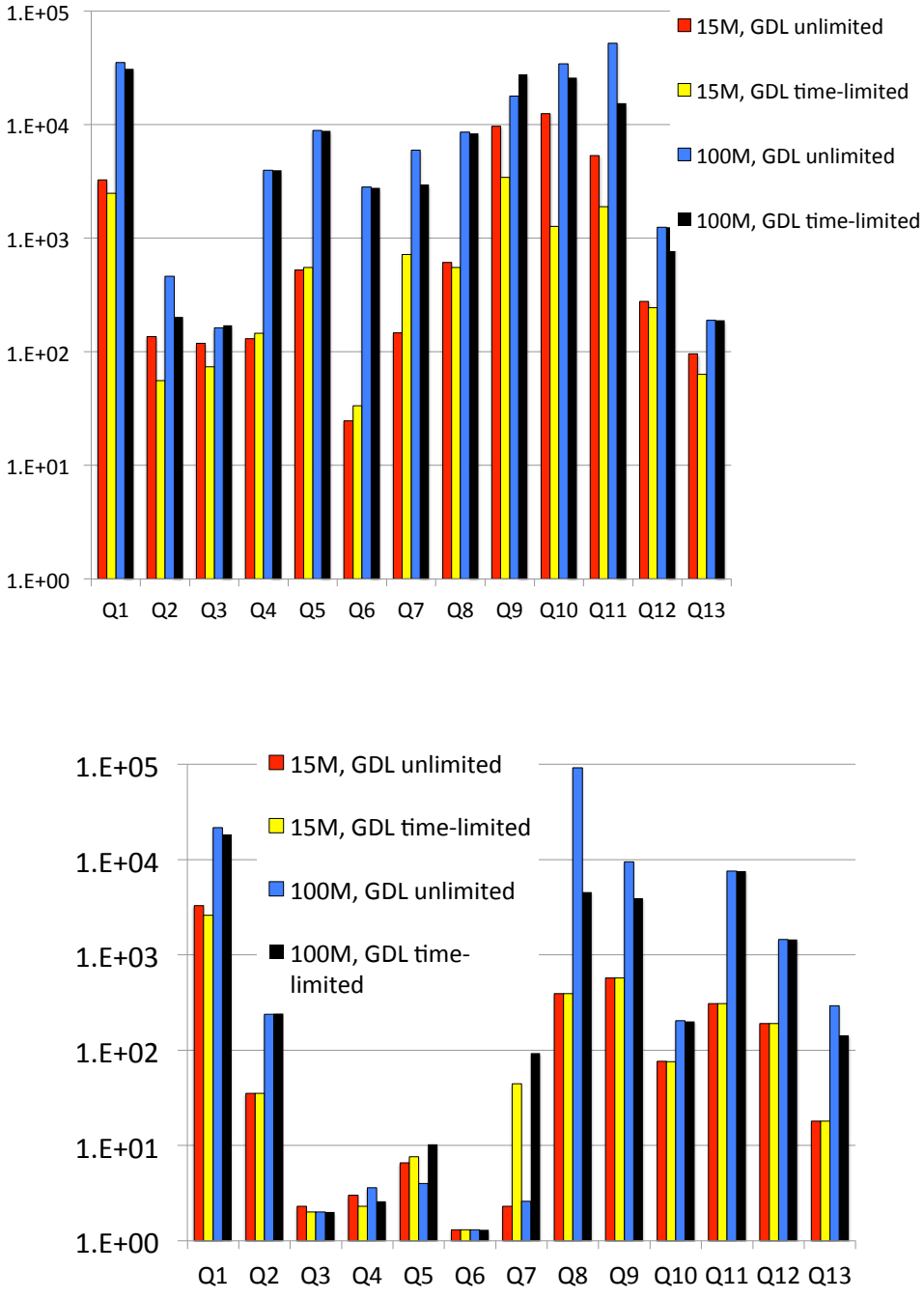

Figure 4.7: Query evaluation time of GDL-selected covers, without time limits, and limited to 20 ms for Postgres (top) and DB2 (bottom).

the one we implemented), and effective in optimizing reformulated queries. GDL attains most of its cost reductions early on during the search, making it a robust tool for improving reformulated query answering performance.

4.6 Related work and conclusion

We proposed a novel framework for any OBDA setting enjoying FOL reducibility of query answering, for which we studied a *space of alternative* FOL *reformulations to evaluate through an RDBMS*. We applied this framework to the DL-Lite$_\mathcal{R}$ description logic, and experimentally demonstrated its performance benefits.

4.6.1 Relationship with prior work on reformulation-based query answering

Our approach departs from the literature focused on a *single* FOL query reformulation, where optimization mainly reduces to *producing fast a* UCQ *reformulation as minimized as possible*: [36, 46, 100, 67, 101, 84, 51] consider DL-Lite$_\mathcal{R}$, existential rules and Datalog$^\pm$. [96] studies CQ-to-USCQ reformulation for existential rules encompassing DL-Lite$_\mathcal{R}$; USCQ reformulations are shown to perform overall better than UCQ ones in an RDBMS. We build on these works to devise CQ-to-JUCQ and CQ-to-JUSCQ reformulation techniques, and used cost estimations to speed up reformulated query evaluation. In particular, our generalized covers can be seen as adapting semijoin-based reducers to the query answering setting. [89] proposes a cost-unaware CQ-to-Datalog reformulation technique; it produces a *non-recursive* Datalog program, which amounts to a JUCQ.

One contribution of this work is an optimization framework (Section 4.2) for any formalism for which query answering is FOL-reducible, e.g., some Description Logics, Datalog$^\pm$ and Existential Rules fragments. The work presented in Chapter 3 is a particular case of this framework for the RDFS ontology language, which corresponds only to the constraints 1, 4, 5 and 11 from Table 2.3, while the DL-Lite$_\mathcal{R}$ language we use comprises 22 such constraints. When reformulating under this rich language, some covers are unsafe (recall Example 11), while in the work presented in the previous chapter any cover leads to a correct query reformulation for the 4 constraints considered there. DL-Lite$_\mathcal{R}$ is important as it is provides the foundations for W3C's standard for *very large* Semantic Web data management OWL2 QL. Thus, the other contributions of our work are: to identify and characterize *safe covers*, guaranteed to lead to reformulations, and a carefully chosen extra space of *generalized covers* which lead to equivalent FOL reformulations and often improve query performance. Our EDL and GDL optimization algoritms (Section 4.4.3) respectively explore exhaustively and greedily this DL-Lite$_\mathcal{R}$-specific space to speed up reformulation-based query answering under DL-Lite$_\mathcal{R}$ constraints. Another difference w.r.t. the work introduced in the preceding chapter is that

this work explores the usage of DB2's RDF store, and find it unsuitable to the complex queries resulting from reformulation.

In the database and Semantic Web communities, there have been intense efforts invested in developing scalable RDF data management platforms, including distributed ones; see e.g., the survey [61]. However, these platforms do not take constraints into account, and thus only support query evaluation, not query answering. Our work is the first to consider optimized algorithms for answering queries under DL-Lite$_{\mathcal{R}}$ constraints through relational databases.

4.6.2 Relationship with prior work on multi-query optimization

Generally speaking, the relationship can be stated as follows. Multi-query optimization (MQO) is interesting as soon as the query to be evaluated has redundant (repeated) subexpressions. In our setting, we distinguish:

Reformulation-induced redundancy refers to the repeated subexpressions appearing in a reformulated query due to the reformulation itself. The `UCQ` has most such redundancies, as illustrated e.g., in Table 3 in the paper.

Reformulation-independent redundancy designates the redundancy that a query may have regardless of the impact of reformulation; for instance, a query may feature repeated subexpressions prior to reformulation, even if the TBox is empty etc.

MQO vs. reformulation-independent redundancy. Any SQL processor capable of MQO can be profitably used to evaluate the cover-based reformulations we chose, in order to *diminish or eliminate reformulation-independent redundancy*. As mentioned in Section 4.1, there was no algebra-level MQO available in the free and commercial engine used in our study, thus our estimation for a cover-based reformulation cost does not take it into account (although some partial support is provided e.g., in Oracle [16]).

One reason why MQO performed during query optimization is complex is the difficulty of deciding equivalence between two logical expressions; under the set semantics used in our work, this is NP-hard even for `CQ`. Works such as [77, 16] use *syntactic equality* of plans, as a criterium for deciding *semantic equivalence*; this is sufficient in [77] because of the focus restricted on select-group by queries over a single table, and in [16] because *under bag semantics, equivalent conjunctive queries are isomorphic* [34]. Under set semantics, both approaches would lead to missing many equivalences, and thus many sharing opportunities. In [80], a similar syntactic condition is used for efficiency, knowing that it may miss sharing opportunities.

If MQO-enabled systems became available, the cost estimation should also reflect its presence; this would be immediately the case for the system's own cost estimation, and would require a revision of our cost estimation function.

We view the possible usage of an MQO-capable SQL engine *to handle reformulation-independent redundancy in the query* as orthogonal to our work. To avoid unwanted impact on our study, *none of the queries used in our study featured reformulation-independent redundancy.*

MQO vs. reformulation-induced redundancy. Reformulating a CQ against a TBox may introduce many repeated subexpressions. The UCQ has most such redundancies, as illustrated e.g., in Table 3 in the paper.

However, our technique applies *before the common subexpression factorization stage*; in the terms of [77], our work belongs to the *strategical* optimization stage, which injects application knowledge and constraints into the optimization problem. Unlike the setting envisioned in [77], however, we do not use such knowledge to create *one plan*, but *several reformulations*, each of which can be seen mid-way between a plan and a query. Indeed, while a cover-based reformulation is still a query, it does make some ordering decisions, in particular each reformulated fragment query is evaluated, and all but one are materialized, before the cover-based reformulation evaluation is finalized by a join.

Thus, our approach, which starts with the TBox and data statistics, and ends by handing over a chosen reformulation to the RDBMS, *never requires work to detect common (repeated) sub-expressions*. Instead:

1. In the simpler setting of an RDFS TBox (Chapter 3), the so-called SCQ reformulation (which pushes all unions immediately above the scans [96]) has the least possible repeated sub-expressions. For instance, on a query of two atoms a_1, a_2, such that a_1 reformulates into a_1^1 and a_2 reformulates into a_2^1 and a_2^2, the SCQ reformulation is:

   $$(a_1 \vee a_1^1) \wedge (a_2 \vee a_2^1 \vee a_2^2)$$

   while the UCQ (featuring many repeated scans) is:

   $$(a_1 \wedge a_2) \vee (a_1 \wedge a_2^1) \vee (a_1 \wedge a_2^2) \vee (a_1^1 \wedge a_2) \vee (a_1^1 \wedge a_2^1) \vee (a_1^1 \wedge a_2^2)$$

   (For larger queries, the UCQ also features repeated joins.)

2. *In the context of the present work, the* SCQ *may not be a FOL reformulation*. When redundancy conflicts with correctness, we should clearly favor correctness first. The safe cover C_{root} is the closest approximation of the SCQ in our context, as it applies the least amount of atom merging within fragments.

   Fortunately, C_{root} is very efficient to build from the TBox and the query: the complexity is $O(|q|^2)$, since we need to compare all the pairs of atoms to see whether they depend on a same TBox predicate. This contrasts with the very high complexity of the abovementioned algebraic MQO techniques [114, 80].

Interestingly, an Oracle 10g [7] reference mentions MQO rewritings which factorize common join expressions across union terms. This would apply to the UCQ setting, however, as mentioned above, based on the TBox, we can much more efficiently minimize redundancy by chosing the C_{root} cover. Also, according to [7], MQO exploration would have to be drastically cut for queries with hundreds of union terms. Thus, our approach is more practical as it exploits information about *the source of redundancy* (namely the reformulation process using the ontology). In contrast, an optimizer has to *deal with the consequences* (detect redundant subexpressions).

The abovementioned Oracle work [7] also mentions query transformations (rewrites) applied at the level of the syntax, in a bottom-up fashion; to learn if a certain subplan rewriting is profitable, the optimizer's cost estimation is used. This strategy of preprocessing of the query guided by the DBMS cost estimator is similar in spirit to our approach (Figure 4.1), and also comparable to the "strategical optimization" stage [77] which applies first in the query optimization process, and injects data and application semantics into the plan.

Our approach could be profitably integrated as a rewrite specific to ontology-based reformulation, within a strong cost-based optimizer such as the one of Oracle [7]. In particular, this would give us access to more sophisticated query cost estimations, while eliminating the overhead we currently pay to get them through a connection to the server.

Currently, we lack access to strong industrial optimizers such as the ones of DB2, SQL Server or Oracle; Postgres' optimizer is easier to extend, but our experiments have shown it is weaker than DB2's.

Other well-known MQO/CSE algorithms have been described e.g., in [47, 77, 80, 114]. Oracle includes several query rewrites to improve nested query performance, among which *subquery coalesce* reduces some redundancy and thus can be seen as related to CSE [16]. A different class of techniques [69, 115] improve the performance of multiple concurrent reads of a table; this can be seen as a physical-level MQO only applying to one-table plans. Such techniques are implemented in DB2, and they indeed help evaluating our reformulations. However, as stated in Section 4.1, our approach does not require detecting repeated subexpressions.

Chapter 5

Conclusion and perspectives

In this chapter, we summarize the contributions previously presented (Section 5.1) and present perspectives for future work, divided in two main parts: ongoing work in a hybrid store project (Section 5.2), and perspectives connected to the main part of this thesis, namely efficient query answering in an ontology-based data access setting (Section 5.3).

5.1 Summary

This thesis provides solutions for efficient *ontology-based data access* query answering in RDF, the prominent W3C standard for the Semantic Web, and in the DL-Lite$_{\mathcal{R}}$ description logic underpinning W3C's OWL2 QL standard for semantic-rich data management. We summarize the problems below.

Efficient query answering in the presence of RDFS constraints. We consider optimizing *reformulation-based query answering* in the setting of OBDA, where SPARQL conjunctive queries are posed against RDF facts on which constraints expressed by an RDF Schema hold. The literature provides query reformulation algorithms for many fragments of RDF. However, reformulated queries may be complex, thus may not be efficiently processed by a query engine; well established query engines even fail processing them in some cases.

1. We generalize the query reformulation approach, by considering a large *space of alternative (equivalent) reformulations*. We characterize the size of our space of alternatives, and show that it is oftentimes too large to be completely explored.

2. We define a *cost model* for estimating the evaluation performance of our reformulated queries through a relational engine; other functions can be used instead, and we show that an RDBMSs' internal cost model can easily be used, too.

3. We devise a novel *algorithm* which selects one alternative reformulated query

which (i) computes the same result as the reformulated query, q^{ref}, produced by state-of-the-art reformulation algorithms, and (ii) reduces significantly the query evaluation cost (or simply makes it possible when evaluating q^{ref} fails!)

4. We implemented this algorithm and deployed it on top of three well-established RDBMSs. Our experiments show that our technique enables reformulation-based query answering where the state-of-the-art approaches are simply unfeasible, while it may decrease its cost by orders of magnitude in other cases.

5. Finally, we compare our reformulation-based query answering technique against saturation-based query answering, both through an RDBMS and the native RDF platform Virtuoso. These experiments confirm the robustness and performance of our technique, showing in particular that in some cases its performance approaches that of saturation-based query answering.

Efficient `FOL` reducible query answering. We consider ontology languages enjoying `FOL` *reducibility of query answering*: answering a query can be reduced to evaluating a certain first-order logic (`FOL`) formula (obtained from the query and ontology) against only the explicit facts. We extend the language of `FOL` reformulations beyond those considered so far in the literature, and investigate *several (equivalent)* `FOL` *reformulations* of a given query, out of which we pick one likely to lead to the best evaluation performance. This contrasts with existing works from the semantic query answering literature, which use reformulation languages allowing *single* `FOL` reformulation (modulo minimization). Considering a *set of reformulations* and relying on a *cost model* to pick a most efficient one has a very visible impact on the efficiency and feasibility of query answering: indeed, picking the wrong reformulation may cause the RDBMS simply to fail evaluating it (typically due to very lengthy queries), while in other cases it leads to bad performance.

1. For logical formalisms enjoying `FOL` reducibility of query answering, we provide a general *optimization framework* that reduces query answering to searching among a set of alternative equivalent `FOL` reformulations, one with minimal evaluation cost in an RDBMS

2. We apply the above mentioned framework to the DL-Lite$_\mathcal{R}$ Description Logic underpinning the W3C's OWL2 QL ontology language. We characterize interesting spaces of such alternative equivalent `FOL` queries for DL-Lite$_\mathcal{R}$ reformulations, and then optimize query answering in such setting by *picking an alternative equivalent* `FOL` *reformulation with lowest estimated evaluation cost* w.r.t. an RDBMS cost model estimation. We provide two algorithms, an exhaustive and a greedy, for this task.

3. Evaluating any of our `FOL` reformulations through an RDBMS leads to the query answer reflecting both the data and the constraints. We demonstrate *experimentally* the effectiveness and the efficiency of our query answering technique for

DL-Lite$_{\mathcal{R}}$, by deploying our query answering technique on top of Postgres and DB2, using several alternative data layouts.

5.2 Ongoing Work: Towards Scalable Hybrid Stores

Data management goes through interesting times[1], as the number of currently available data management systems (DMSs) is probably higher than ever before. This leads to unique opportunities for data-intensive applications often involving diverse datasets, some very large while others may be of moderate size, some highly structured (e.g., relations) while others may have more complex structure (e.g., graphs) or little structure (e.g., text or log data), as some systems provide excellent performance on certain data processing operations. Yet, it also raises great challenges, as a system efficient on some tasks may perform poorly or not support other tasks, making it impossible to use a single DMS for a given application.

As part of my thesis, I have started to study the possibility to use different DMSs side by side in order to take advantage of their best performance, as advocated under terms such as *hybrid* or *poly-stores*. Observe that even once such a combination of stores is chosen, it may need to be changed over time, as the data or application needs change, as new more efficients system may become available, or on the contrary their usage needs to be discontinued (for instance due to changes in the application owner's IT policy, or in the pricing of a certain commercial system). In such cases, one should not have to modify (rewrite) the applications, but rather have it run and adapt seamlessly to the new context.

The work on ESTOCADA has lead to so far to a CIDR 2015 vision paper [23] and the ICDE demonstration in 2016 [24]. Below, we motivate it through an example (Section 5.2.1), outline the core scientific problems and describe our architecture and state of advancement (Section 5.2.2). Finally, Section 5.2.3 places our work in the area of similar research projects.

5.2.1 Motivating Example and Challenges

We illustrate this through an example. Consider a traditional customer relationship management (CRM) application. While typically CRM needed to deal only with a relational data warehouse, now the application needs to incorporate new data sources in order to build a better knowledge of its customers: (i) information gleaned from social network *graphs* about clients' activity and interests, and (ii) *log file* from multiple e-commerce stores, characterizing the clients' purchase activity in those stores.

[1] Alludes to the so-called Chinese curse "may you live in interesting times" (see e.g., https://en.wikipedia.org/wiki/May_you_live_in_interesting_times).

The in-house RDBMS performs well on the relational data. However, the social graph data fits badly in that system, and the company attempts to store it in a dedicated graph store, until an engineer argues that it should be decomposed and stored into a highly-efficient NoSQL key-value store system she has just experimented with. The storage and processing of log files is delegated to a Hive installation (over Hadoop), until the summer research intern observes that recent work [70] has shown that *some* data from Hive should be lifted at runtime in the relational data warehouse to gain a few orders of magnitude of performance!

Deploying and exploiting the CRM application for best performance is set to be a nightmare now. There is little consensus on what systems to use, if any; three successive engineers have recommended (and moved the social data into and out of) three different stores, one for graphs, one for key-value pairs, and the last an in-memory column database. Part of the log data has been moved in the in-memory column store, too, when the social data was stored there; this made their joint exploitation faster. But the whole log dataset could not fit in the single-node column store installation, and data migration fatigue had settled in before a suggestion was made (and rejected) to move everything to yet another cluster installation of the column store. The team working on the application feels battered and confused. The application is sometimes very slow. Migrating data is painful at every change of system; they are not sure the complete data set survived at each step, and data keeps accumulating. Yet, a new system may be touted as the most efficient for graph (or for log) data next week. The possibility that the next efficient store would drastically improve performance further confuses the situation as the teem sees it. Would it be faster? How to know?

My thesis research has contributed to designing and implementing ESTOCADA, a novel architecture for efficiently handling highly heterogeneous datasets based on a dynamic set of potentially very different data stores. While heterogeneous data integration is an old topic [73, 54, 40, 78], the remark "one-size does not fit all" [95] has been revisited for instance in the last CIDR [75, 59, 41], and the performance advantages brought by multi-stores have been recently noted e.g., in [70]. The set of features which, together, make ESTOCADA novel are:

Natively multi-model ESTOCADA supports a variety of data models, including flat and nested relations, trees and graphs, including important classes of semantic constraints such as primary and foreign keys, inclusion constraints, redundancy of information within and across distinct storage formats, etc. which are needed to enforce application semantics.

Application-invisible ESTOCADA provides to client applications access to each dataset in its native format. This does not preclude other mapping / translation logic above ESTOCADA' client API but we do not discuss them in this paper. Instead, our focus is on efficiently storing the data, even if in a very different format from its original one, as discussed below.

Fragment-based store Each data set is stored as a set of fragments, whose content may overlap. The fragmentation is completely transparent to ESTOCADA' clients, i.e., it is the system's task to answer queries based on the available fragments.

Mixed store Each fragment may be stored in any of the stores underlying a ESTOCADA installation, be it relational, tree- or graph-structured, based on key-value pairs etc., centralized or distributed, disk- or memory-based etc. Thus, potentially any piece of any dataset may reside in any of the available systems; each query may be answered by combining data from any set of systems. Query answering must be aware of the *constraints* introduced implicitly when storing fragments in non-native models. For instance, when tree-structured data are stored in a relational store, the resulting edge relation satisfies the constraint that each node has at most one parent, the descendant and ancestor relations are inverses of each other and are related non-trivially to the edge relation, etc.

View-based rewriting and view selection The invisible glue holding all the pieces together is *view-based rewriting with constraints*. Specifically, each data fragment is internally described as a materialized view over one or several datasets; query answering amounts to view-based query rewriting, and storage tuning relies on view selection. Describing the stored fragments as views over the data allows changing the set of stores with no impact on ESTOCADA' applications [54]; this simplifies the migration nightmare outlined above. Finally, our reliance on views gives sound foundation to efficiency, as it guarantees the complete storage of data, and the correctness of the fragmentation and query answering, among others.

Technical challenges The ESTOCADA scenario involves the coexistence of a large number of materialized views mixing data formats (modeling the native sources) with significant redundancy between them (due to repeated migration and view selection arising organically over the history of the system, as opposed to clean-slate planning). While the problem of rewriting using views is classical, it has typically been addressed and practically implemented only in limited scenarios that do not apply here. These scenarios feature (i) only relatively small numbers of views; (ii) minimal overlap between views as their selection is planned ahead of time; (iii) views expressed over the same data model; (iv) rewriting that exploits only limited integrity constraints (typically only key/foreign key in existing systems). The large number of views and their redundancy notoriously contribute (at least) exponentially to the explosion in the search space for rewritings, even when working within a single data model.

5.2.2 Achitecture and state of advancement

We briefly recall the fundamental definitions of *query equivalence* and *query containment* [33, 8, 9, 60, 34, 74, 35]. We denote a database instance by $\mathcal{D}$ and use $q(\mathcal{D})$ to denote the result of evaluating a query q over database $\mathcal{D}$.

Definition 5.2.1 (Query equivalence)**.** *Two queries q_1, q_2 are* equivalent*, denoted $q_1 \equiv q_2$, iff $q_1(\mathcal{D}) \equiv q_2(\mathcal{D})$ for any database $\mathcal{D}$.*

Definition 5.2.2 (Query containment)**.** *A query q_1 is say to be* contained *in another query q_2, denoted $q_1 \subseteq q_2$, iif we have $q_1(\mathcal{D}) \subseteq q_2(\mathcal{D})$ for any database $\mathcal{D}$.*

Observe that if $q_1 \subseteq q_2$ and $q_2 \subseteq q_1$, then $q_1 \equiv q_2$. Based on the above:

Definition 5.2.3 (Equivalent query rewriting)**.** *Given a query q and a set of views $\mathcal{V}$, an* equivalent rewriting of q using $\mathcal{V}$ *is an expression $e(v_1, v_2, \ldots, v_k)$, $v_i \in \mathcal{V}$, $1 \leq i \leq k$, over the views in $\mathcal{V}$, which is equivalent to q. In other words, for any database $\mathcal{D}$, $e(v_1, v_2, \ldots, v_k)(\mathcal{D}) = q(\mathcal{D})$.*

Definition 5.2.3 covers *full (complete)* rewritings, which rely only on the materialized views. *Partial* rewritings, combining $\mathcal{V}$ views and the database $\mathcal{D}$ itself are also interesting. However, the rewritings that interest us in ESTOCADA are those considering the base data available and modeled by a set of views, i.e., complete rewritings.

Query rewriting using (materialized) views (a.k.a. view-based query rewriting, view-based query reformulation) algorithms sought to find all the possible (equivalent) rewritings of a given query q using an specified set of views $\mathcal{V}$ [54].

We now explain how ESTOCADA aims at automating the solution to scenarios such as the one previously described.

Fragments described as materialized views. To adapt to changes in the datasets, workload, and set of DMSs being used, we chose to internally *represent each data fragment as a materialized view over one or several datasets*; thus, query answering amounts to view-based query rewriting. As is well-known from prior work in data integration, this local-as-view approach allows the application to remain unchanged as the underlying data collections are modified. Further, our reliance on views gives sound foundation to efficiency, as it guarantees the complete storage of data, and the correctness of the fragmentation and query answering.

To further simplify the development of applications, each dataset is accessed through a language specific to its native data model, be it SQL for relational stores, key-based search API for key-value data, etc. However, for efficiency, a fragment F of a dataset D (whose data model is $\mathcal{M}_D$) may be stored in a data model $\mathcal{M}_F$ different from $\mathcal{M}_D$; similarly, a fragment F' may store combined results from different datasets of possibly different data models, leading to more cross-model transformation of the data between the application dataset and the stored fragments.

Relational pivot model with constraints. To enable query rewriting over and across different data models, we translate into an *internal pivot model* the declarative specification of the data stored in each fragment, as well as the incoming query, formulated in the application dataset model; specifically, our pivot model is based on relational conjunctive queries. Further, to correctly account for the characteristics of each application

data model $\mathcal{M}_a$ and storage data model $\mathcal{M}_s$, we describe their specific features in the same pivot model, by means of powerful *constraints*.

For instance, we describe the organization of a document data model (whether this concerns $\mathcal{M}_a$ or $\mathcal{M}_s$) using a small set of relations such as *Node(nID, name)*, *Child(pID, cID)*, *Descendant(aID, dID)*, etc. together with the constraints specifying that every node has just one parent and one tag, that every child is also a descendant. Such modeling had first been introduced in local-as-view data integration XML integration works [78, 40].

More generally, constraints allow a faithful internal modeling of datasets, since they can express functional dependencies and keys (for instance, node or tuple IDs) naturally present in many settings, be it relations, documents or graph stores. Also, importantly for the usage of key-value stores, we rely on an original *encoding of access pattern restrictions* such as "the value of the key must be specified in order to access the values associated to this key" into relations with constraints. This enables building only *feasible* rewritings, i.e., such that the information needed to access a given data source is either provided by the query, or has been obtained from data sources previously accessed while evaluating the rewriting.

Rewriting under constraints: optimized Chase & Backchase. To rewrite queries in the presence of constraints, the method of choice is known as Chase & Backchase (C&B, in short), a classical powerful tool long considered too inefficient to be of practical relevance. ESTOCADA exploits the very significant performance savings brought by the recent provenance-aware C&B algorithm (PACB, in short) [57]. PACB drastically reduces the back-chase effort by keeping track of the results of the various chase steps applied during the algorithms, to avoid repeated and fruitless work; this results in rewriting speedups that can even outperform a commercial relational optimizer by 1-2 orders of magnitude (in terms of combined optimization and execution time).

Rewriting decoding. From the above, it follows that query rewriting takes place, first, at the level of our pivot relational conjunctive model endowed with constraints, and it leads to a rewriting which is a conjunctive query over the relations corresponding to the stored fragments.

Depending on the data model of these fragments, the relational atoms used in the rewritings may either correspond to actual relations, or to key-value collections which can be seen as relations with binding patterns, or to the virtual relations used to encode more complex data models, such as the $Node$, $Child$ and $Descendant$ relations mentioned above (the encoding of nested relations such as supported e.g., in Pig and HBase is very similar). From this relational, conjunctive rewriting, a *rewriting translation step* is performed to:

1. Group the rewriting atoms referring to each distinct fragment involved in the rewriting; for instance, it can be inferred that the three atoms *Document(dID, "file.json"), Root(dID, rID), Child(rID, cID), Node(cID, book)* found in a rewriting refer to a single document, by following the connections among nodes and

knowledge of the JSON data model;

2. Reformulate each such rewriting snippet into a query which can be completely evaluated over a single fragment;

3. If several fragments are stored in the same underlying DMS, identify the largest subquery that can be delegated to that DMS, along the lines of query evaluation in wrapper-mediator systems [97]. Observe that if the DMS has a distributed architecture, e.g., Spark deployed on a cluster, the delegated subquery will be evaluated in parallel fashion, allowing ESTOCADA to leverage its efficiency.

Evaluation of non-delegated operations. A decoded rewriting may be unable to push (delegate) some query operations to the DMS storing a fragment if the DMS does not support them; for instance, most key-value and document stores do not support joins. Similarly, if a query on structured data requests the construction of new nested results (such as JSON or XML documents, or nested tuples), and if the inputs to this operation are not stored in a DMS supporting such result construction natively, it will have to be executed outside of the underlying DMSs. To evaluate such "last-step" operations, ESTOCADA comprises its own *lightweight execution engine*, based on a nested relational model, whose atomic types include constants, node IDs, and document types; it provides in particular implementations of the BindJoin operator needed to access data sources with access restrictions. The engine is a close variant of the one previously developed in the ViP2P project and used e.g., in [79, 64, 63].

Choice of the most efficient rewriting to use. For a given query and set of fragments, there may be several rewritings, each of which may lead to several evaluation plans of different performance. The problem of chosing the best rewriting and best evaluation plan in this setting is quite close to the one previously considered in distributed mediator systems [85], with the extra difficulty that one needs to compare the performance of execution across a variety of stores. At the time of this writing, this part of the project is not complete, and thus will not delve upon it further; we expect it to be addressed in future work.

Architecture. Figure 5.1 outlines the architecture of our prototype based on the above discussion. We assume the typical application uses many data sets $D_1, D_2, \ldots, D_n$, even though our smart storage method may be helpful even for a single data set, distributing it for efficient access across many stores, potentially based on different data models.

The *Storage Descriptor Manager* stores information about the available data fragments D_1/F_1, D_1/F_2, ..., D_1/F_n, D_2/F_1, ... etc., and where they are stored in the underlying DMSs, illustrated by a NoSQL store, a key-value store, a document store, one for nested relations, and finally a relational one. For each data fragment D_i/F_j residing in the store S_k, a *storage descriptor* $sd(S_k, D_i/F_j)$ is produced. The descriptor specifies *what* data (the fragment D_i/F_j) is stored *where* within S_k. The *what* part of the descriptor is specified by a query over the data set D_i, following the *native model* of D_i. The fragment can

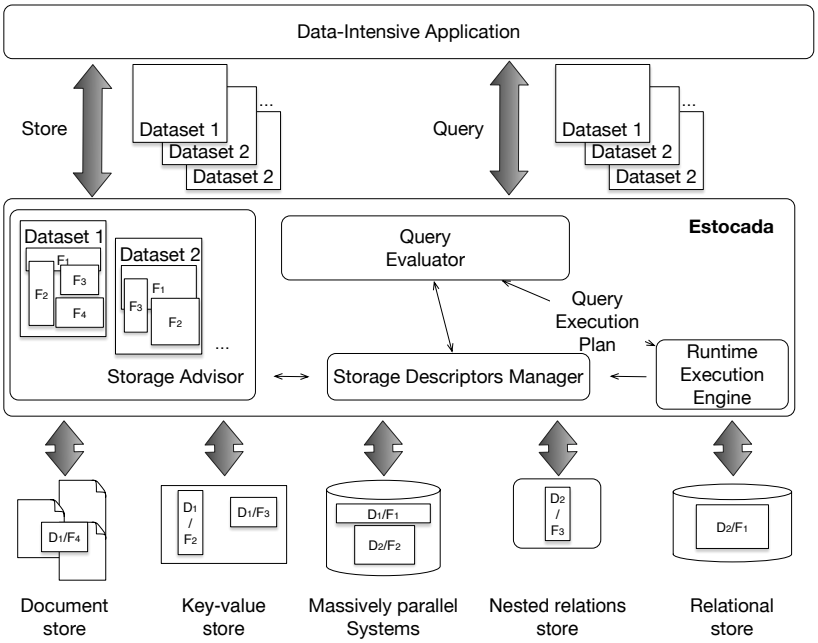

Figure 5.1: ESTOCADA architecture.

thus be seen as a *materialized view* over D_i. The *where* part of the descriptor is structured according to the organization of data within S_k. For instance, if S_k is a relational store, the *where* information consists of the schema and table name, whereas if S_k is a key-value store, it could hold the name of the collection, attribute name, etc. Finally, the descriptor $sd(S_k, D_i/F_j)$ also specifies the data access operation supported by S_k which allows retrieving the D_i/F_j data (such as: a table scan, a look-up based on a collection name, column group name, and column name in a key-value store, etc.), as well as the access credentials required in order to connect to the system and access it.

The *Storage Advisor* recommends dropping redundant fragments that are rarely used or under-performing, and adding new fragments that fit recently heavy-hitting queries. To solve this problem across data models, we once again exploit our pivot model to reduce to the novel setting of relational view selection *under constraints*.

The *Query Evaluator* receives application queries. If a query carries over a single source D_i, the query will likely be in the native language of D_i. If the query carries over multiple sources having different data models, this assumes the existence of a global-as-view integration layer on top of the (application-transparent) local-as-view approach internally followed by ESTOCADA. While we do not focus on this (optional) global-as-view integration layer, in such a case we assume the query is specified by combining algebraic operations (such as filter, join, union, etc.) on top of individual queries carrying over each dataset. It is rather straightforward then to translate such a query in the pivot model, by focusing first on the queries confined to a data source, and then on the combination operators. The evaluator looks up the storage descriptors corresponding to fragments of the queried datasets, calls the PACB engine to obtain rewritings. The *Runtime*

Execution Engine then translates such rewritings into excutable ones as described above and evaluates them.

5.2.3 Related Work on Hybrid Stores

Heterogeneous data integration is an old topic [73, 54, 40, 78] but the remark "one-size does not fit all" [95] has been recently revisited [75, 59]. The performance benefits of using multiple stores together (a Hadoop one and a relational database) have been demonstrated in [70]; they select relational views to be materialized based on cost information, but do not handle multiple data models through a unified approach as we do. Polystores [43, 42] allow querying heterogeneous stores by grouping similar-model platform into "islands" and explicitly sending queries to one store or another; data sets can also be migrated by the users. This contrasts with our LAV approach where the data store variety is hidden to the application layer. The integration of "NoSQL" stores has been considered e.g., in [12] again in a top-down GAV approach without considering materialized views.

Adaptive stores for a single data model have been studied e.g., in [56, 10, 37, 66, 65]; views have been also used in [91, 6] to improve the performance of a large-scale distributed relational store. The novelty of ESTOCADA here is to support multiple data models, by relying on powerful query reformulation techniques under constraints.

Data exchange tools such as Clio [44, 53] allow migrating data between two different schemas. We aim at providing to the applications transparent data access to heterogeneous systems, relying on fundamentally different rewriting techniques.

View-based rewriting and view selection are grounded in the seminal works [54, 73]; the latter focuses on maximally contained rewritings, while we target exact query rewriting, which leads to very different algorithms. Further setting our work apart is the scale and usage of integrity constraints. Our pivot model recalls the ones described in [40, 78] but ESTOCADA generalizes these works by allowing multiple data models both at the application and storage level.

To conclude, we believe hybrid (multi-store) architectures have the potential to bring huge performance improvements, since (redundant) views storing query results can increase the efficiency of query evaluation by many orders of magnitude. ESTOCADA supports this by a local-as-view approach whose immediate benefit is flexibility since it requires no work when the underlying data storage changes; we demonstrate its performance benefits and the interest of simple storage recommendation heuristics. Our work is ongoing toward a cost-based recommendation of optimal fragmentation.

5.3 Perspectives

Semantic Web data management as well as hybrid stores open numerous avenues for future research; we outline some of them below.

DAG plans for query answering in the presence of RDFS constraints. To improve the performance of reformulated query evaluation performance even further, a natural extension of the work presented in Chapter 3 is to consider not only *joins of unions of conjunctive queries* but all plans built using selections, projections, joins, semi-joins, unions and materialization operators (SPJUM), allowing *DAG plans*.

Materialization allows to save extra work in case of worth reuse opportunities for intermediate results, while join and union operators can appear in the computed plans at any level, differently from all the previous proposals in literature.

The sweet spot between Saturation- and Reformulation-based query answering. In Chapter 3, we consider the problem of efficient query answering in the presence of RDF Schema constraints. Two main query answering techniques exist, namely *saturation* and *reformulation*. In the saturation-based query answring approach the constraints are compiled into the database by making all implicit data explicit, while the reformulation-based query answering technique compiles the constraints into a modified query, which, evaluated over the explicit data only, computes all the answer due to explicit and/or implicit data. In this thesis we focus on optimizing reformulation-based query answering in the setting of ontology-based data access. However, a better study of the performance trade-offs between saturation-, reformulation- and mixed query answering approaches such as [99], in order to automatically recommend the best technique to use, and/or how to combine them to achieve better performance in a given application setting, could lead to even more efficient systems for query answering in the presence of constraints.

Efficient query answering for DL-Lite$_\mathcal{R}$ in the presence of mappings. In Chapter 4, we have introduced a novel query optimization framework for ontology-based data access settings enjoying FOL reducibility and applied it to the DL-Lite$_\mathcal{R}$. However, OBDA setting also allows declarative specifications, known as *mappings*, connecting each concept and role in the ontology with a view over the data, materialized in relational databases. For such purpose the W3C introduced R2RML [103], a language expressing customized mappings from relational databases to RDF datasets. Early experiments shown the potential of extending our framework, to support mappings, thus providing efficient query answering for the wider OBDA setting.

Semantic constraints aware query rewriting. Among the RDF data management systems, some support reformulation-based query answering, e.g., Stardog, while others support reformulation-based query answering for a subset of the RDFS rules, e.g., Virtuoso (rdfs:subClassOf and rdfs:subPropertyOf). RDF platforms such as 3store [108], OWLIM [112], Oracle Semantic Graph [111] support saturation-based query answering, based on (a subset of) RDF entailment rules, and others such as Hexastore [107]

or RDF-3X [83], ignore entailed triples and only provide query *evaluation* on top of the RDF graph. Thus, query answering in the presence of semantic constraints requires hybrid store solutions, such as ESTOCADA, to be aware of the semantic capabilities of the underlying systems. In the case of ESTOCADA, this amounts to take into account the ontology, as well as the stores capabilities, while performing the query rewriting. In other words, it requires to encode the ontology into constraints, enabling efficient query answering *using materialized conjunctive query views*, which may partially or completely rewrite the conjunctive queries appearing in the reformulated fragments.

Storage advisor. In ESTOCADA, once the view-based rewriting part of the project is completed and complemented by a cost model to allow chosing the most efficient rewriting, the next step is to devise the automated storage tuning (storage advisor) which will recommend the views to materialize within each underlying data management system, so as to obtain the best performance for a given query workload.

Appendix A

Detailed queries

A.1 Queries used in the efficient query answering in the presence of RDFS constraints experiments

This section lists the SPARQL queries used in the experimental section of Chapter 3. For readability and without loss of information, the URIs starting with "http://www.lehigh.edu" were slightly shortened by eliminating a few /-separated steps.

Q01(?X ?Y) :- ?X "http://www.w3.org/1999/02/22-rdf-syntax-ns#type" "http://www.lehigh.edu/univ-bench.owl#Employee", ?X "http://www.lehigh.edu/univ-bench.owl#worksFor" "http://www.Department0.University0.edu", ?X "http://www.lehigh.edu/univ-bench.owl#degreeFrom" ?Y
Q02(?X ?Y ?U ?V ?W) :- ?X "http://www.w3.org/1999/02/22-rdf-syntax-ns#type" "http://www.lehigh.edu/univ-bench.owl#Employee", ?X "http://www.lehigh.edu/univ-bench.owl#worksFor" "http://www.Department0.University0.edu", ?X "http://www.lehigh.edu/univ-bench.owl#degreeFrom" ?Y, ?X "http://www.lehigh.edu/univ-bench.owl#name" ?U, ?X "http://www.lehigh.edu/univ-bench.owl#emailAddress" ?V, ?X "http://www.lehigh.edu/univ-bench.owl#telephone" ?W
Q03(?X ?Y) :- ?X "http://www.w3.org/1999/02/22-rdf-syntax-ns#type" "http://www.lehigh.edu/univ-bench.owl#Employee", ?X "http://www.lehigh.edu/univ-bench.owl#worksFor" "http://www.Department0.University0.edu", ?X "http://www.lehigh.edu/univ-bench.owl#doctoralDegreeFrom" ?Y
Q04(?X ?Y ?Z) :- ?X "http://www.w3.org/1999/02/22-rdf-syntax-ns#type" ?Y, ?X "http://www.lehigh.edu/univ-bench.owl#doctoralDegreeFrom" ?U, ?X "http://www.lehigh.edu/univ-bench.owl#memberOf" ?Z
Q05(?X ?Y ?Z) :- ?X "http://www.w3.org/1999/02/22-rdf-syntax-ns#type" "http://www.lehigh.edu/univ-bench.owl#Student", ?X "http://www.lehigh.edu/univ-bench.owl#advisor" ?Y, ?Y "http://www.lehigh.edu/univ-bench.owl#teacherOf" ?Z, ?X "http://www.lehigh.edu/univ-bench.owl#takesCourse" ?Z
Q06(?X ?W ?Y ?Z) :- ?X "http://www.w3.org/1999/02/22-rdf-syntax-ns#type" ?W, ?X "http://www.lehigh.edu/univ-bench.owl#advisor" ?Y, ?Y "http://www.lehigh.edu/univ-bench.owl#teacherOf" ?Z, ?X "http://www.lehigh.edu/univ-bench.owl#takesCourse" ?Z

Table A.1: LUBM queries Q1-Q6.

Q07(?X ?Y) :- ?X “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#Faculty”, ?X “http://www.lehigh.edu/univ-bench.owl#degreeFrom” ?Y, ?X “http://www.lehigh.edu/univ-bench.owl#memberOf” ?Y
Q08(?X ?Y) :- ?X “http://www.lehigh.edu/univ-bench.owl#degreeFrom” ?Y, ?X “http://www.lehigh.edu/univ-bench.owl#memberOf” “http://www.Department0.University0.edu”
Q09(?X ?Y) :- ?X “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#Professor”, ?Y “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#Professor”, ?X “http://www.lehigh.edu/univ-bench.owl#degreeFrom” ?U, ?Y “http://www.lehigh.edu/univ-bench.owl#degreeFrom” ?V, ?X “http://www.lehigh.edu/univ-bench.owl#memberOf” ?V, ?Y “http://www.lehigh.edu/univ-bench.owl#memberOf” ?U
Q10(?W ?X ?Y) :- ?X “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#GraduateStudent”, ?Y “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#Faculty”, ?W “http://www.lehigh.edu/univ-bench.owl#publicationAuthor” ?X, ?W “http://www.lehigh.edu/univ-bench.owl#publicationAuthor” ?Y
Q11(?W ?X ?Y) :- ?X “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#GraduateStudent”, ?X “http://www.lehigh.edu/univ-bench.owl#advisor” ?Y, ?W “http://www.lehigh.edu/univ-bench.owl#publicationAuthor” ?X, ?W “http://www.lehigh.edu/univ-bench.owl#publicationAuthor” ?Y
Q12(?W ?X ?Y) :- ?X “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#GraduateStudent”, ?X “http://www.lehigh.edu/univ-bench.owl#advisor” ?Y, ?Y “http://www.lehigh.edu/univ-bench.owl#teacherOf” ?Z, ?X “http://www.lehigh.edu/univ-bench.owl#takesCourse” ?Z, ?W “http://www.lehigh.edu/univ-bench.owl#publicationAuthor” ?X, ?W “http://www.lehigh.edu/univ-bench.owl#publicationAuthor” ?Y
Q13(?Z) :- ?X “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#Student”, ?Y “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#GraduateStudent”, ?X “http://www.lehigh.edu/univ-bench.owl#advisor” ?Z, ?Y “http://www.lehigh.edu/univ-bench.owl#advisor” ?Z
Q14(?Z ?W) :- ?X “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/˜zhp2/univ-bench.owl#Student”, ?Y “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#GraduateStudent”, ?Z “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” ?W, ?X “http://www.lehigh.edu/univ-bench.owl#advisor” ?Z, ?Y “http://www.lehigh.edu/univ-bench.owl#advisor” ?Z
Q15(?X ?U ?Z) :- ?X “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#University”, ?U “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” ?Z, ?Y “http://www.lehigh.edu/univ-bench.owl#publicationAuthor” ?U, ?U “http://www.lehigh.edu/univ-bench.owl#memberOf” ?X
Q16(?Z) :- ?X “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#Student”, ?Y “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#GraduateStudent”, ?Z “http://www.lehigh.edu/univ-bench.owl#teacherOf” ?U, ?Z “http://www.lehigh.edu/univ-bench.owl#teacherOf” ?V, ?X “http://www.lehigh.edu/univ-bench.owl#takesCourse” ?U, ?Y “http://www.lehigh.edu/univ-bench.owl#takesCourse” ?V
Q17(?X) :- ?Y “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#GraduateCourse”, ?X “http://www.lehigh.edu/univ-bench.owl#teacherOf” ?Y, ?X “http://www.lehigh.edu/univ-bench.owl#teacherOf” ?Z
Q18(?X ?W) :- ?X “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” ?W, ?Y “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#GraduateCourse”, ?Z “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#Course”, ?X “http://www.lehigh.edu/univ-bench.owl#takesCourse” ?Y, ?X “http://www.lehigh.edu/univ-bench.owl#takesCourse” ?Z

Table A.2: LUBM queries Q7-Q18.

Q19(?X ?Z ?W) :- ?X “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#Faculty”, ?Z “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” ?W, ?Z “http://www.lehigh.edu/univ-bench.owl#publicationAuthor” ?X
Q20(?X ?Y ?Z) :- ?X “http://www.lehigh.edu/univ-bench.owl#degreeFrom” ?Z, ?Y “http://www.lehigh.edu/univ-bench.owl#publicationAuthor” ?X
Q21(?X ?Y ?Z) :- ?X “http://www.lehigh.edu/univ-bench.owl#doctoralDegreeFrom” ?Z, ?Y “http://www.lehigh.edu/univ-bench.owl#publicationAuthor” ?X
Q22(?X ?Y) :- ?X “http://www.lehigh.edu/univ-bench.owl#doctoralDegreeFrom” ?Z, ?X “http://www.lehigh.edu/univ-bench.owl#teacherOf” ?Y
Q23(?X ?Y) :- ?X “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” ?Y, ?X “http://www.lehigh.edu/univ-bench.owl#degreeFrom” “http://www.University0.edu”
Q24(?X) :- ?X “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#Faculty”, ?X “http://www.lehigh.edu/univ-bench.owl#degreeFrom” “http://www.University0.edu”
Q25(?X ?Y) :- ?X “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#Faculty”, ?X “http://www.lehigh.edu/univ-bench.owl#degreeFrom” “http://www.University532.edu”, ?X “http://www.lehigh.edu/univ-bench.owl#memberOf” ?Y
Q26(?X ?Y) :- ?X “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” ?Y, ?X “http://www.lehigh.edu/univ-bench.owl#degreeFrom” “http://www.University532.edu”, ?X “http://www.lehigh.edu/univ-bench.owl#memberOf” “http://www.Department1.University7.edu”
Q27(?X) :- ?X “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://www.lehigh.edu/univ-bench.owl#Faculty”, ?X “http://www.lehigh.edu/univ-bench.owl#degreeFrom” “http://www.University532.edu”, ?X “http://www.lehigh.edu/univ-bench.owl#memberOf” “http://www.Department1.University7.edu”
Q28(?X ?U ?Y ?V ?Z) :- ?X “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” ?U, ?Y “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” ?V, ?X “http://www.lehigh.edu/univ-bench.owl#mastersDegreeFrom” “http://www.University532.edu”, ?Y “http://www.lehigh.edu/univ-bench.owl#doctoralDegreeFrom” “http://www.University532.edu”, ?X “http://www.lehigh.edu/univ-bench.owl#memberOf” ?Z, ?Y “http://www.lehigh.edu/univ-bench.owl#memberOf” ?Z

Table A.3: LUBM queries Q19-28.

Q01(?X ?Y) :- ?Y “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” “http://sw.deri.org/ aharth/2004/07/dblp/dblp.owl#Document”, ?X “http://sw.deri.org/ aharth/2004/07/dblp/dblp.owl#datatypeField” ?Y,
Q02(?X ?Z) :- ?X “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” ?Z, ?X “http://sw.deri.org/ aharth/2004/07/dblp/dblp.owl#objectField” “http://www.example.org/dblp/”,
Q03(?X ?Y ?Z) :- ?X “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” ?Z, ?X “http://sw.deri.org/ aharth/2004/07/dblp/dblp.owl#datatypeField” ?Y,
Q04(?X ?Y ?Z) :- ?X “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” ?Z, ?X “http://sw.deri.org/ aharth/2004/07/dblp/dblp.owl#datatypeField” ?Y, ?X “http://purl.org/dc/elements/1.1/publisher” “Springer”
Q05(?Y ?U ?V) :- ?X “http://sw.deri.org/ aharth/2004/07/dblp/dblp.owl#objectField” “http://www.example.org/dblp/”, ?X “http://purl.org/dc/elements/1.1/title” ?Y, ?X “http://purl.org/dc/elements/1.1/creator” ?U, ?X “http://purl.org/dc/elements/1.1/date” ?V
Q06(?Y ?U ?Z) :- ?X “http://sw.deri.org/ aharth/2004/07/dblp/dblp.owl#editor” ?Y, ?Y “http://xmlns.com/foaf/0.1/name” ?Z, ?X “http://purl.org/dc/elements/1.1/title” ?U, ?X “http://sw.deri.org/ aharth/2004/07/dblp/dblp.owl#datatypeField” ?V ?X “http://purl.org/dc/elements/1.1/publisher” “Springer”
Q07(?X ?Y ?U ?Z) :- ?X “http://sw.deri.org/ aharth/2004/07/dblp/dblp.owl#editor” ?Y, ?Y “http://xmlns.com/foaf/0.1/name” ?Z, ?X “http://purl.org/dc/elements/1.1/title” ?U, ?X “http://sw.deri.org/ aharth/2004/07/dblp/dblp.owl#datatypeField” ?V ?X “http://purl.org/dc/elements/1.1/publisher” “Springer”
Q08(?X ?Y ?Z) :- ?X “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” ?Z, ?Y “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” ?Z, ?X “http://purl.org/dc/elements/1.1/publisher” “Springer” ?Y “http://purl.org/dc/elements/1.1/publisher” “Morgan Kaufmann”
Q09(?Z ?U ?T) :- ?X “http://sw.deri.org/ aharth/2004/07/dblp/dblp.owl#datatypeField” ?U, ?Y “http://sw.deri.org/ aharth/2004/07/dblp/dblp.owl#datatypeField” ?T, ?X “http://purl.org/dc/elements/1.1/creator” ?Z, ?Y “http://purl.org/dc/elements/1.1/creator” ?Z, ?X “http://purl.org/dc/elements/1.1/publisher” “Springer” ?Y “http://purl.org/dc/elements/1.1/publisher” “Morgan Kaufmann”
Q10(?Z ?V ?U ?W ?T) :- ?X “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” ?V, ?Y “http://www.w3.org/1999/02/22-rdf-syntax-ns#type” ?W, ?X “http://sw.deri.org/ aharth/2004/07/dblp/dblp.owl#datatypeField” ?U, ?Y “http://sw.deri.org/ aharth/2004/07/dblp/dblp.owl#datatypeField” ?T, ?X “http://purl.org/dc/elements/1.1/creator” ?Z, ?Y “http://purl.org/dc/elements/1.1/creator” ?Z, ?X “http://purl.org/dc/elements/1.1/title” ?R, ?Y “http://purl.org/dc/elements/1.1/title” ?S, ?X “http://purl.org/dc/elements/1.1/publisher” “Springer” ?Y “http://purl.org/dc/elements/1.1/publisher” “Morgan Kaufmann”

Table A.4: DBLP queries Q1-Q10.

A.2 Queries used in the Efficient query answering in settings where reformulation is FOL reducible experiments

This section lists the SPARQL queries used in the experimental section of Chapter 4. We have shortened for presentation purposes some of the strings, e.g., AssociateProfessor2@Department1.University0.edu becomes AssocProf2@Dept1.Univ0.edu.

	Query q	$\lvert q^{\text{UCQ}}\rvert$
$q_1(u,i,n,e,t)$:-	ub:Professor(x) $\wedge$ ub:degreeFrom(x,u) $\wedge$ ub:researchInterest$(x,i)\wedge$ ub:name(x,n) $\wedge$ ub:emailAddress(x,e) $\wedge$ ub:telephone(x,t)	145
$q_2(x,e,t)$:-	ub:Professor$(x)\wedge$ ub:degreeFrom$(x$,"http://www.University870.edu")$\wedge$ ub:researchInterest$(x$,"Research21") $\wedge$ ub:name$(x$,"AssociateProfessor2")$\wedge$ ub:emailAddress$(x,e)\wedge$ ub:telephone(x,t)	145
$q_3(x)$:-	ub:Professor$(x)\wedge$ ub:degreeFrom$(x$,"http://www.University870.edu")$\wedge$ ub:researchInterest$(x$,"Research21") $\wedge$ ub:name$(x$,"AssociateProfessor2")$\wedge$ ub:emailAddress$(x$,"AssocProf2@Dept1.Univ0.edu")$\wedge$ ub:telephone$(x$,"xxx-xxx-xxxx")$\wedge$	145
$q_4(x,y)$:-	ub:Professor$(x)\wedge$ ub:teacherOf$(x,y)\wedge$ ub:degreeFrom$(x$,"http://www.University870.edu") $\wedge$ ub:researchInterest$(x$,"Research21")$\wedge$ ub:name$(x$,"AssociateProfessor2")$\wedge$ub:telephone$(x$,"xxx-xxx-xxxx")$\wedge$ ub:emailAddress$(x$,"AssocProf2@Dept1.Univ0.edu")	145
$q_5(x,y,z)$:-	ub:Professor$(x)\wedge$ ub:teacherOf$(x,y)\wedge$ ub:worksFor$(x,z)\wedge$ ub:degreeFrom$(x$,"http://www.University870.edu")$\wedge$ ub:researchInterest$(x$,"Research21") $\wedge$ ub:name$(x$,"AssociateProfessor2")$\wedge$ ub:emailAddress$(x$,"AssocProf2@Dept1.Univ0.edu")$\wedge$ ub:telephone$(x$,"xxx-xxx-xxxx")	290
$q_6(x,n)$:-	ub:Faculty$(x)\wedge$ ub:publicationAuthor$(y,x)\wedge$ ub:researchInterest$(x$,"Research16")$\wedge$ ub:name$(y,n)\wedge$ ub:emailAddress$(x$,"AssocProf0@Dept0.Univ0.edu")	35
$q_7(n)$:-	ub:Professor$(x)\wedge$ ub:teacherOf$(x,c)\wedge$ ub:memberOf$(x$,"http://www.Dep0.Univ0.edu") $\wedge$ ub:name$(x,n)\wedge$ ub:emailAddress$(x$,"FullProf8@Dept0.Univ0.edu")$\wedge$ ub:telephone$(x$,"xxx-xxx-xxxx")	116

Table A.5: LUBM$^{\exists}_{20}$ queries Q1-Q7.

	Query q	$\lvert q^{\text{UCQ}}\rvert$
$q_8(x,n)$:-	ub:Faculty$(x) \wedge$ ub:publicationAuthor$(y,x) \wedge$ ub:researchInterest$(x,$"Research16"$) \wedge$ ub:name(y,n)	35
$q_9(n,e)$:-	ub:Student$(x) \wedge$ ub:takesCourse$(x,c) \wedge$ ub:advisor$(x,a) \wedge$ ub:memberOf$(x,$"http://www.Dept0.University0.edu"$) \wedge$ ub:telephone$(x,$"xxx-xxx-xxxx"$) \wedge$ ub:teacherOf$(p,c) \wedge$ ub:emailAddress$(p,e) \wedge$ ub:researchInterest$(a,$"Research7"$) \wedge$ ub:memberOf$(a,$"http://www.Dept0.University0.edu"$) \wedge$ ub:name(c,n)	368
$q_{10}(n,e)$:-	ub:Professor$(x) \wedge$ ub:takesCourse$(x,c) \wedge$ ub:advisor$(x,a) \wedge$ ub:memberOf$(x,$"http://www.Dept0.University0.edu"$) \wedge$ ub:telephone$(x,$"xxx-xxx-xxxx"$) \wedge$ ub:teacherOf$(p,c) \wedge$ ub:emailAddress$(p,e) \wedge$ ub:researchInterest$(a,$"Research7"$) \wedge$ ub:memberOf$(a,$"http://www.Dept0.University0.edu"$) \wedge$ ub:name(c,n)	464
$q_{11}(x)$:-	ub:Professor$(x) \wedge$ ub:Student(x)	667
$q_{12}(x)$:-	ub:Professor$(x) \wedge$ ub:Department(x)	609
$q_{13}(x)$:-	ub:Publication$(x) \wedge$ ub:Department(x)	357

Table A.6: LUBM$^{\exists}_{20}$ queries Q8-Q13.

Appendix B

Condensé de la thèse en français

B.1 Introduction

B.1.1 Big Data

Le Web 2.0, comme il y est souvent fait référence, représente la deuxième étape du World Wide Web dont les principales caractéristiques sont: (i) l'apparition de pages au contenu dynamique, de services et d'applications offrant aux utilisateurs une expérience plus riche, (ii) la large acceptation et importante croissance de logiciels en services (SaaS) grâce à des protocoles d'intégration légers, des API, etc..., (iii) la participation massive de 46% de la population mondiale, et (iv) un changement dans le paradigme de participation et de consommation des utilisateurs qui, de consommateurs sont devenus producteurs de contenu (Wikipédia est une encyclopédie en ligne sur laquelle n'importe qui peut écrire ou éditer des articles; Blogger est un service de publication de blogs permettant aux utilisateurs de rédiger des posts et de les commenter; Twitter est un réseau social permettant aux utilisateurs de partager des messages de 140 caractères; Youtube est une plateforme de partage vidéo sur laquelle les utilisateurs peuvent commenter et noter les vidéos; WhatsApp Messenger est un service de messagerie instantanée pour échanger des messages texte et audio, des documents, des images, des vidéos, etc...entre deux utilisateurs ou plus; Facebook est un réseau social ayant plus de 1,65 milliards d'utilisateurs actifs chaque mois au 31 mars 2016[1] etc...).

Du point de vue de la gestion des données, le Web 2.0 a été perturbateur en termes de volume de données. Nous créons 2,5 exabytes de données chaque jour! En d'autres termes, 90% des données mondiales ont été créées au cours des 24 derniers mois[2]. En parallèle sont intervenus la baisse du coût du matériel et l'augmentation continue de la capacité de mémoire, l'émergence d'une variété de technologies de calcul et de stockage, telles que GPU, FPGA, etc... [94], les réseaux à haute vitesse (par exemple Infiniband),

[1] `https://newsroom.fb.com/company-info/`

[2] `https://www-01.ibm.com/software/data/bigdata/what-is-big-data.html`

et l'émergence du cloud computing. Ces événements ne sont pas passés inaperçus dans le domaine de l'apprentissage automatique ni dans ceux de la gestions de données et de systèmes distribués, qui avaient fourni les systèmes et les solutions nécessaires (par exemple, Hadoop, Spark, Flink, Kafka, etc...) pour soutenir cette nouvelle génération d'applications Big Data.

B.1.2 Semantic Web

Dans le cadre de la troisième génération Web, le Web 3.0 est également appelé Web sémantique. Contrairement aux systèmes traditionnels de représentation des connaissances généralement centralisés [18], le Web sémantique est conçu pour être une architecture distribuée mondialement. C'est une avancée supplémentaire dans la direction du rêve de Tim Berners-Lee en 2000 [17]: le moment où les ordinateurs deviennent capables d'analyser toutes les données du Web. Le Web sémantique est une extension du World Wide Web dans lequel le contenu a une structure permettant aux ordinateurs de traiter avec fiabilité la sémantique et donc de manipuler les données de manière sensée. Toutefois, afin qu'une telle vision se réalise, un ensemble de modèles de données et de formats pour spécifier les descriptions sémantiques des ressources Web est nécessaire. Dans cette optique, le World Wide Web Consortium (W3C) a présenté le Resource Description Framework (RDF) servant de base au Web sémantique. Le RDF est un modèle de données flexible, permettant d'exprimer des déclarations portant sur des ressources (uniquement identifiées par leur URI) sous la forme d'expressions sujet-prédicat-objet. Pour améliorer la puissance descriptive des ensembles de données RDF, le W3C a proposé le schéma RDF (RDFS) [105] et le Web Ontology Language (OWL) [102], facilitant la représentation de contraintes sémantiques (i.e. contraintes ontologiques) entre les classes et les propriétés utilisées. L'ensemble des faits ainsi que les règles logiques sur l'appartenance d'individus en classes ou en relations formées entre eux, constituent une base de connaissances. Finalement, la popularité actuelle et l'utilisation d'ontologies dans le Web est due à quatre raisons majeures [4]:

- La façon souple et naturelle de structurer les documents de manière multidimensionnelle permet de trouver des informations pertinentes à travers des collections de documents très volumineux.
- La sémantique formelle logique des ontologies fournit des moyens d'inférence menant à un raisonnement. Ainsi, une ontologie peut être interprétée et traitée par des machines.
- Les ontologies permettent de préciser les concepts et d'améliorer la recherche sur le Web. Par exemple, lors de la recherche du mot "aleph", nous pourrions spécialiser le concept dans l'ontologie *livre: aleph*, ce qui aboutirait aux livres écrits par JL Borges, P. Coelho, etc..., et éviter les réponses indésirables où le terme est utilisé avec une autre connotation (comme ceux se rapportant à la lettre

portant le même nom ou à la séquence des nombres utilisés pour représenter la cardinalité des ensembles infinis qui peuvent être bien ordonnés).

- Les ontologies servent de lien local entre des sources d'information hétérogènes. De plus, leur potentiel d'inférence permet d'intégrer automatiquement différentes sources de données.

Par exemple, si le célèbre écrivain argentin J.L. Borges apparaît dans un document, alors il a un URI associé, auquel toutes les autres ressources (livres, articles, Prix, etc...) se réfèrent. La version RDF de la British National Bibliography développée par British Library [3] exprime des informations pertinentes sur les livres, auteurs, éditeurs, etc..., et contiendra les livres écrits par Borges, les éditeurs de ces livres, leur lieu de publication et année. En parallèle, GeoNames [4] fournit des informations sur les lieux de publication, et DBPedia [5] (la contrepartie sémantique de Wikipédia [6]) donnera davantage d'informations sur Borges, comme son opinion politique, sa famille, etc...Tous les ensembles de données interconnectés susmentionnés font partie de Linked Open Data [7].

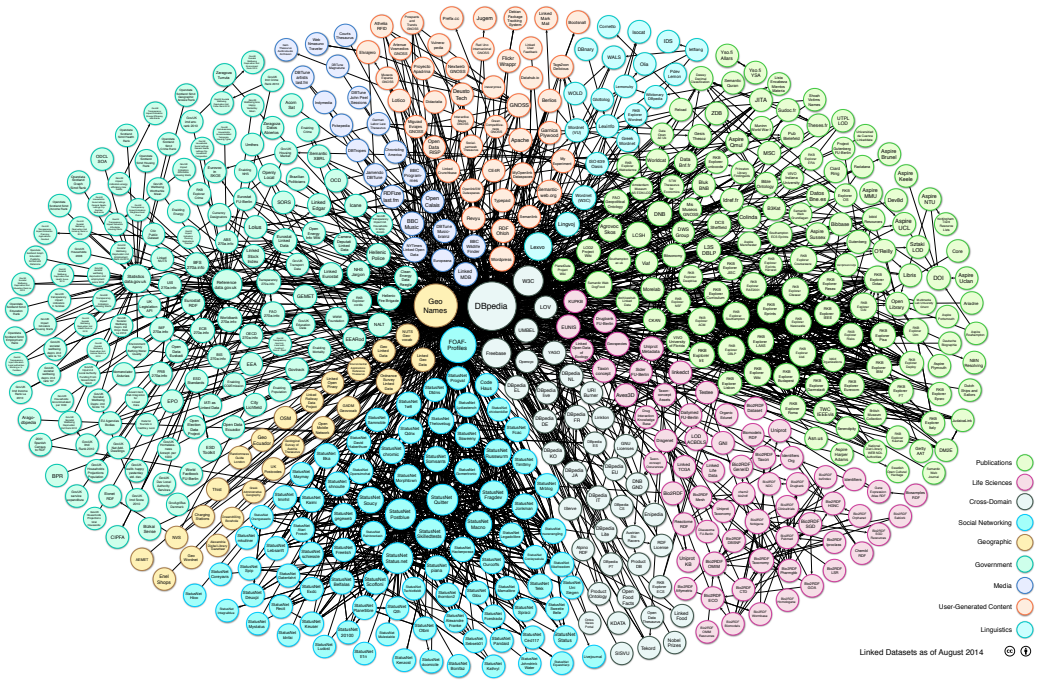

Schéma B.1: Le Linked Open Data cloud en Avril 2014.

Le schéma B.1 [90] illustre certains des ensembles de données les plus connus (RDF) dans le Open Data cloud en avril 2014. Chaque nœud du graphe correspond à un ensem-

[3] http://www.bl.uk/bibliographic/datafree.html
[4] www.geonames.org/
[5] http://wiki.dbpedia.org/
[6] https://en.wikipedia.org/wiki/Main_Page
[7] http://lod-cloud.net/

ble de données en RDF, alors que le diamètre du nœud reflète la taille de l'ensemble des données. En outre, il y a un espace commun entre deux nœuds si les deux ensembles de données ont des URI en commun. Les ensembles de données sont alors interconnectés. Comme indiqué dans [90], la taille du Web sémantique est en pleine croissance, doublant de taille presque chaque année.

Pour exploiter cette richesse de données, le langage de requête SPARQL a été défini [106]. Par la suite, de nouvelles techniques et algorithmes ont été proposés pour le traitement des requêtes SPARQL, fondés sur le partitionnement vertical [1], l'indexation [107], le traitement de jointure efficace [82], la technique de view-selection [49], les systèmes de gestion RDF [108, 110, 113, 83], pour n'en citer que quelques-uns.

B.2 Répondre efficacement aux requêtes en présence de contraintes

Répondre à des requêtes portant sur des données en présence de contraintes déductives, ce qui mène à des données implicites dérivant de données explicites et de contraintes, implique une étape de raisonnement afin de calculer les réponses aux requêtes. Deux techniques de réponse existent : la saturation des données compile les contraintes dans la base de données en rendant explicites toutes les données implicites, tandis que la reformulation de requêtes compile les contraintes dans une requête modifiée, qui, évaluée uniquement sur les données explicites, calcule toutes les réponses que les données soient explicites ou implicites. Jusqu'à présent, répondre aux requêtes fondées sur la reformulation a reçu beaucoup moins d'attention que la saturation. En particulier, les requêtes reformulées peuvent être complexes et leur évaluation peut donc être très difficile.

Dans ce chapitre, nous nous concentrons sur l'optimisation de la réponse aux requêtes fondées sur la reformulation dans le paramétrage des accès aux données fondées sur l'ontologie, lorsqu'il est répondu aux requêtes conjonctives SPARQL via un ensemble de données RDF sur lesquelles les contraintes RDFS ont un impact.

Nous considérons le contexte dans lequel les requêtes conjonctives (`CQ`), une fois reformulées en unions de requêtes conjonctives (`UCQ`) ou semi-conjonctives (`SCQ`), sont traitées pour évaluation d'une requête par un RDBMS, un dépôt de données RDF dédié et un module de traitement des requêtes, ou plus généralement par tout système capable d'évaluer les sélections, projections, jointures et unions. Comme le montrent nos expériences, l'évaluation de requêtes reformulées peut être très difficile, même pour des processeurs RDF relationnels ou natifs, qui peuvent les gérer de manière inefficace ou échouer à les gérer, même sur des ensembles de données de taille modérée.

L'approche sélectionnée est la suivante: étant donné une requête conjonctive SPARQL q et un algorithme de reformulation de requête $\mathcal{A}$ qui transforme un `CQ` en `UCQ`, nous explorons un grand espace de reformulations alternatives de q que nous appelons `JUCQ` (pour des unions jointes de requêtes conjonctives), qui reprennent les reformulations `UCQ`

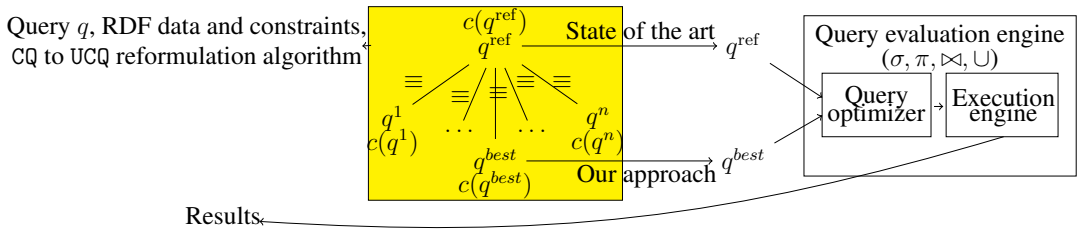

Schéma B.2: Notre approche d'évaluation efficace des requêtes conjonctives SPARQL reformulées.

et SCQ, et dont nous choisissons une reformulation JUCQ avec le coût estimé le plus bas. Chaque reformulation JUCQ est obtenue à partir d'un ensemble soigneusement choisi d'invocations de l'algorithme $\mathcal{A}$ guidé par notre modèle de coût.

Contributions. Les contributions que nous apportons au problème de répondre de manière efficace aux requêtes SPARQL, via la reformulation, peuvent être décrites comme suit (voir Schéma B.2):

1. Nous généralisons l'approche de reformulation des requêtes, en considérant un espace étendu des reformulations alternatives (équivalentes) JUCQ. Cet espace correspond au carré jaune du Schéma B.2. Elle inclut et généralise de façon significative les travaux antérieurs fondés sur la reformulation UCQ ou SCQ. Nous caractérisons la taille de notre espace d'alternatives, et montrons qu'il est souvent trop grand pour être complètement exploré.

2. Nous définissons un modèle de coût pour estimer la performance d'évaluation de nos requêtes reformulées par un moteur relationnel. D'autres fonctions peuvent être utilisées à la place, et nous montrons qu'un modèle de coût interne RDBMS peut être facilement utilisé.

3. Nous concevons un nouvel algorithme qui sélectionne une requête reformulée alternative, appelée q^{best} dans le Schéma B.2, qui (i) calcule le même résultat que la requête reformulée q^{ref} UCQ ou SCQ, et (ii) réduit significativement le coût d'évaluation de la requête (ou simplement le rend possible lorsque l'évaluation q^{ref} échoue!)

4. Nous avons mis en œuvre cet algorithme et l'avons déployé en plus de trois RDBMS qui diffèrent significativement dans leur capacité à gérer les reformulations UCQ et SCQ. Nos expériences confirment que notre algorithme est celui permettant de tirer le maximum de chacun de ces modules, en tirant parti de leurs forces et en évitant leurs faiblesses via l'utilisation de notre modèle de coût, que nous calibrons séparément pour chaque système. Cela rend la reformulation pos-

sible lorsque UCQ et/ou SCQ échouent, et apporte des améliorations de performance de plusieurs ordres de grandeur en relation avec UCQ.

5. Enfin, nous avons mis en perspective notre technique efficace de réponse aux requêtes fondées sur la reformulation en la comparant à la réponse aux requêtes fondées sur la saturation en se fondant sur PostgreSQL et via la plate-forme Virtuoso dédiée à la gestion de données Web sémantique. Ces expériences confirment la robustesse et la performance de notre technique, montrant en particulier que dans certains cas sa performance approche celle de la réponse aux requêtes fondées sur la saturation.

B.3 Répondre efficacement à une Requête réductible FOL

Dans ce chapitre, nous transférons l'idée développée dans le chapitre précédent au paramétrage du modèle de données et des paires de langages de requête bénéficiant de la réductibilité FOL dans le cadre de la réponse aux requête (i.e. le modèle de données et les paires de langages de requête peuvent être réduits à l'évaluation d'une formulation logique de premier ordre, obtenue à partir de la requête et ontologie, contre les faits explicites seulement), englobant de nombreuses bases de connaissances et des paramètres de la base de données, tels que Description Logics, Datalog$^{\pm}$ et des fragments Existential Rules.

Nous proposons un cadre d'optimisation des requêtes pour tout paramètre logique OBDA en profitant de la réductibilité FOL de réponse aux requêtes. Nous étendons le langage des reformulations FOL au-delà des reformulations précédemment envisagées et cherchons plusieurs reformulations FOL d'une requête donnée. Celle susceptible de mener au meilleur résultat sera choisie. Ceci contraste avec les travaux existants à partir de la requête sémantique Littérature (voir la section 4.6), qui utilise des langages de reformulation permettant une seule reformulation FOL (minimisation modulo). Considérer un ensemble de reformulations et s'appuyer sur un modèle de coût pour en choisir un plus efficace a un impact très visible sur l'efficacité et la faisabilité des réponses aux requêtes. En effet, choisir la mauvaise reformulation peut provoquer l'échec d'évaluation du RDBMS (généralement en raison de requêtes très longues) ou de mauvaises performances.

Nous appliquons ce cadre général à la DL-Lite$_{\mathcal{R}}$ Description Logic [32] sous-jacent à la célèbre norme W3C OWL2 QL pour les applications Web sémantiques riches ce qui a démontré des avantages significatifs de performance dans ce contexte. Répondre aux requêtes en DL-Lite$_{\mathcal{R}}$ a fait l'objet d'une attention significative dans la littérature, notamment sur la réductibilité FOL, par exemple, [32, 2, 86, 89, 36, 100].

Contributions. Nous apportons les contributions suivantes au problème de l'optimisation des requêtes FOL (voir Schéma B.3):

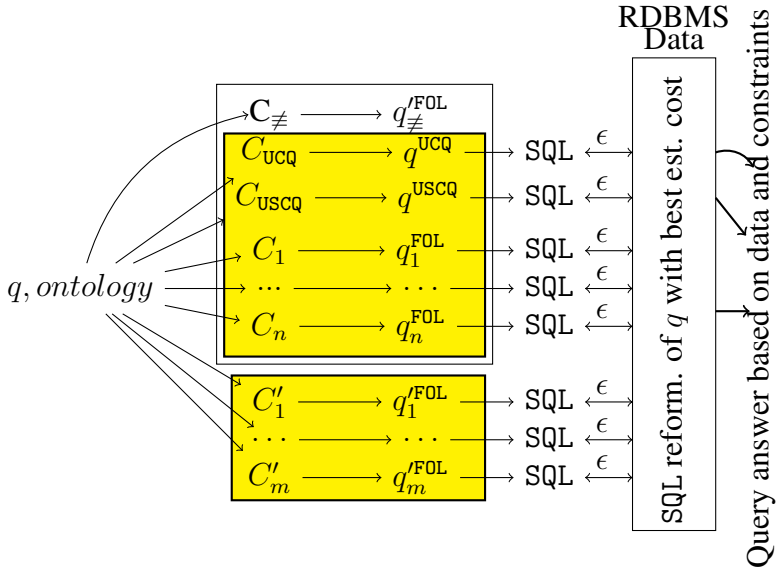

Schéma B.3: Approche de reformulation optimisée des FOL.

1. Pour les formalismes logiques bénéficiant de la réductibilité FOL dans la réponse aux requêtes, nous fournissons un cadre général d'optimisation qui réduit la réponse aux requêtes à la recherche d'autres reformulations équivalentes aux reformulations FOL, l'une d'entre elles avec un coût d'évaluation minimal dans un RDBMS. Dans le schéma B.3, à partir de la requête q et de l'ensemble des contraintes ontologiques $\mathcal{T}$, on déduit d'abord un espace de recouvrements de requêtes, représenté dans le rectangle horizontal blanc, et noté C avec quelques indices. Nous montrons alors comment dériver une requête FOL pouvant être une reformulation FOL de q liée à $\mathcal{T}$.

2.. Nous caractérisons des espaces intéressants pour les requêtes alternatives à la reformulation FOL, soit pour DL-Lite$_\mathcal{R}$. Tout d'abord, nous identifions une condition de sécurité suffisante pour choisir des couvertures qui conduisent à une reformulation FOL de la requête. Cette condition est remplie par les couvertures dans le rectangle jaune du haut dans le Schéma B.3, mais n'est pas satisfaite par $C_{\not\equiv}$ au-dessus. Notre espace de couverture sécuritaire englobe toutes les reformulations FOL, y compris celles déjà étudiées dans la littérature. Ensuite, nous introduisons un ensemble de couvertures généralisées (rectangle jaune en bas dans le SchémaB.3) et une technique généralisée de reformulation donnant toujours une reformulation aux requêtes FOL. Cela est souvent plus efficace que les couvertures simples. Notre approche peut être combinée et permet d'optimiser toute reformulation technique existante pour DL-Lite$_\mathcal{R}$.

3. Nous optimisons alors la réponse aux requêtes dans le cadre de DL-Lite$_\mathcal{R}$ en énumérant des couvertures simples et généralisées, et en choisissant une couverture dérivée de la reformulation FOL au plus bas coût estimé d'évaluation en relation à une estimation du modèle de coût RDBMS ϵ (désigné par les flèches bidirectionnelles ϵ dans le Schéma).

Nous fournissons deux algorithmes, un exhaustif et un gourmand, pour cette tâche.

4. L'évaluation de nos reformulations FOL à travers le RDBMS (flèches épaisses à droite du Schéma B.3) conduit à une réponse reflétant les données et les contraintes. Nous démontrons expérimentalement l'efficacité et l'efficience de notre technique de réponse aux requêtes pour DL-Lite$_\mathcal{R}$ en déployant notre technique de réponse aux requêtes au-dessus de Postgres et DB2, et en utilisant des configurations alternatives de données.

Du point de vue du traitement des requêtes et de l'optimisation, notre approche peut être considérée comme appartenant à l'étape dite d'optimisation stratégique introduite dans [77] (où l'application sémantique est injectée dans la requête). Cela est similaire dans l'esprit à la réécriture du niveau de syntaxe effectué par des optimiseurs tels que Oracle 10g's [7]. Nous partageons avec [77] l'idée d'injecter la sémantique d'abord, et comme [7], nous utilisons l'estimation des coûts pour guider nos réécritures. Un thème commun est de réécrire avant de commander des unions, de sélectionner des opérateurs physiques, etc...

De ce point de vue, notre contribution peut être considérée comme un ensemble d'alternatives (réécritures) avec des garanties d'exactitude et des algorithmes guidant ces réécritures pour une classe spéciale de requêtes obtenues à partir de reformulations FOL de CQ contre des ontologies.

B.4 Conclusion

Cette thèse fournit des solutions pour répondre efficacement aux requêtes portant sur des données fondées sur l'ontologie en RDF, la norme W3C de premier plan pour le Web sémantique, et dans la logique de description DL-Lite$_\mathcal{R}$ qui sous-tend la norme OWL2 QL du W3C pour la gestion des données à la sémantique riche. Résumons les problèmes ci-dessous.

Répondre efficacement aux requêtes en présence de contraintes RDFS. Nous envisageons d'optimiser la réponse aux requêtes fondées sur la formulation dans le cadre de l'OBDA où les requêtes conjonctives SPARQL sont posées contre des faits RDF sur lesquels pèsent des contraintes exprimées par un Schéma RDF.

La littérature fournit des algorithmes de reformulation pour beaucoup de Fragments RDF. Toutefois, les requêtes reformulées peuvent être complexes, mais ne pas être efficacement traitées par un moteur de requête. Même les moteurs de requêtes bien établis échouent parfois dans leur traitement.

1. Nous généralisons l'approche de reformulation de requêtes en considérant un grand espace de reformulations alternatives (équivalentes). Nous caractérisons la taille de notre espace d'alternatives et montrons qu'il est souvent trop grand pour être complètement exploré.

2. Nous définissons un modèle de coût pour estimer la performance d'évaluation de nos requêtes reformulées via un moteur de recherche relationnel. D'autres fonctions peuvent être utilisées à la place et nous montrons qu'un modèle de coût interne RDBMS peut également être facilement utilisé.

3. Nous concevons un nouvel algorithme qui sélectionne une autre requête alternative reformulée qui (i) calcule le même résultat que la requête reformulée, q^{ref}, produit par des algorithmes de reformulation, et (ii) réduit de façon significative le coût d'évaluation de la requête (ou tout simplement le rend possible lorsque l'évaluation q^{ref} échoue!)

4. Nous avons implémenté cet algorithme et l'avons déployé en sus de trois SGBDR déjà bien établis. Nos expériences montrent que notre technique permet de répondre à des requêtes fondées sur la reformulation lorsque des approches classiques sont irréalisables, tout en diminuant le coût dans certains cas.

5. Enfin, nous comparons notre technique de réponse aux requêtes fondées sur la reformulation avec les réponses aux requêtes fondées sur la saturation via un RDBMS et la plateforme RDF Virtuoso. Ces expériences confirment la robustesse et la performance de notre technique, montrant en particulier que dans certains cas sa performance approche celle de la réponse aux requêtes fondées sur la saturation.

Répondre efficacement aux requêtes `FOL` réductibles. Nous considérons les langages d'ontologie jouissant de réductibilité `FOL` dans la réponse à une requête: répondre à une requête peut être réduit à évaluer une certaine formule logique de premier ordre (`FOL`) (obtenue à partir de la requête et de l'ontologie) et des faits explicites.

Nous étendons le langage des reformulations `FOL` au-delà de celles considérées jusqu'à présent dans la littérature, et étudions plusieurs reformulations `FOL` (équivalentes) d'une requête donnée dont nous choisissons celle susceptible de conduire à la meilleure performance. Cela contraste avec les travaux existants de la littérature sur la réponse aux requêtes sémantiques qui utilisent des langages de reformulation permettant une reformulation `FOL` unique (module minimisation). En considérant un ensemble de reformulations et en s'appuyant sur un modèle de coût pour choisir la plus efficace a un impact très visible sur l'efficacité et la faisabilité de la réponse à une requête. En effet, choisir la reformulation erronée peut mener le RDBMS à échouer à l'évaluer (généralement en raison de requêtes très longues), ou conduire à une mauvaise performance.

1. Pour les formalismes logiques bénéficiant de la réductibilité `FOL` dans la réponse aux requêtes, nous fournissons un cadre d'optimisation général qui réduit la réponse aux requêtes à la recherche parmi un ensemble de reformulations `FOL` équivalentes, l'une avec une évaluation minimale dans un RDBMS.

2. Nous appliquons le cadre mentionné ci-dessus à la logique de description DL-Lite$_{\mathcal{R}}$ sous-tendant le langage d'ontologie OWL2 QL du W3C. Nous caractérisons des espaces étant une alternative équivalente aux requêtes FOL pour les reformulations DL-Lite$_{\mathcal{R}}$, puis optimisons la réponse aux requêtes via un paramétrage sélectionnant une alternative équivalente à la reformulation FOL avec le coût d'évaluation estimé dans le plus faible et une estimation de coût RDBMS. Nous fournissons deux algorithmes, un exhaustif et un gourmand pour cette tâche.

3. L'évaluation de nos reformulations FOL à travers le RDBMS conduit à une réponse reflétant les données et les contraintes. Nous démontrons expérimentalement l'efficacité et l'efficience de notre technique de réponse aux requêtes pour DL-Lite$_{\mathcal{R}}$ en déployant notre technique de réponse aux requêtes au-dessus de Postgres et DB2, et en utilisant des configurations alternatives de données.

Bibliography

[1] Daniel J. Abadi, Adam Marcus, Samuel R. Madden, and Kate Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB*, 2007.

[2] Nada Abdallah, François Goasdoué, and Marie-Christine Rousset. DL-LITER in the light of propositional logic for decentralized data management. In *IJCAI*, 2009.

[3] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[4] Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, and Pierre Senellart. *Web Data Management*. Cambridge University Press, 2011.

[5] Philippe Adjiman, François Goasdoué, and Marie-Christine Rousset. SomeRDFS in the semantic web. *JODS*, 8, 2007.

[6] P. Agrawal, A. Silberstein, B. F. Cooper, U. Srivastava, and R. Ramakrishnan. Asynchronous View Maintenance for VLSD Databases. In *SIGMOD*, 2009.

[7] Rafi Ahmed, Allison Lee, Andrew Witkowski, Dinesh Das, Hong Su, Mohamed Zait, and Thierry Cruanes. Cost-based query transformation in Oracle. In *VLDB*, 2006.

[8] Alfred V. Aho, Yehoshua Sagiv, and Jeffrey D. Ullman. Efficient Optimization of a Class of Relational Expressions. *TODS*, 1979.

[9] Alfred V. Aho, Yehoshua Sagiv, and Jeffrey D. Ullman. Equivalences Among Relational Expressions. *SIAM J. Comput.*, 1979.

[10] I. Alagiannis, S. Idreos, and A. Ailamaki. H2O: a hands-free adaptive store. In *SIGMOD*, 2014.

[11] Marcelo Arenas, Claudio Gutierrez, and Jorge Pérez. Foundations of RDF databases. In *Reasoning Web*, 2009.

[12] P. Atzeni, F. Bugiotti, and L. Rossi. Uniform access to NoSQL systems. *Information Systems*, 2014.

[13] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The DL Handbook: Theory, Implementation, and Applications*. Cambridge Univ. Press, 2003.

[14] Jean-François Baget, Michel Leclère, Marie-Laure Mugnier, and Eric Salvat. On rules with existential variables: Walking the decidability line. *Artificial Intelligence*, 175(9-10), 2011.

[15] Barton library data. `http://simile.mit.edu/rdf-test-data/barton`.

[16] Srikanth Bellamkonda, Rafi Ahmed, Andrew Witkowski, Angela Amor, Mohamed Zait, and Chun-Chieh Lin. Enhanced subquery optimizations in Oracle. *PVLDB*, 2(2), August 2009.

[17] Tim Berners-Lee and Mark Fischetti. *Weaving the web - the original design and ultimate destiny of the World Wide Web by its inventor*. HarperBusiness, 2000.

[18] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web: A new form of web content that is meaningful to computers will unleash a revolution of new possibilities. *Scientific American Magazine*, pages 28–37, 2001.

[19] Philip A. Bernstein and Dah-Ming W. Chiu. Using semi-joins to solve relational queries. *J. ACM*, 28(1), 1981.

[20] Barry Bishop, Atanas Kiryakov, Damyan Ognyanoff, Ivan Peikov, Zdravko Tashev, and Ruslan Velkov. OWLIM: A family of scalable semantic repositories. *Semantic Web*, 2(1), 2011.

[21] Mihaela A. Bornea, Julian Dolby, Anastasios Kementsietsidis, Kavitha Srinivas, Patrick Dantressangle, Octavian Udrea, and Bishwaranjan Bhattacharjee. Building an efficient RDF store over a relational database. In *SIGMOD*, 2013.

[22] Jeen Broekstra and Arjohn Kampman. Inferencing and truth maintenance in RDF Schema: Exploring a naive practical approach. In *PSSS Workshop*, 2003.

[23] Francesca Bugiotti, Damian Bursztyn, Alin Deutsch, Ioana Ileana, and Ioana Manolescu. Invisible glue: Scalable self-tunning multi-stores. In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*, 2015.

[24] Francesca Bugiotti, Damian Bursztyn, Alin Deutsch, Ioana Manolescu, and Stamatis Zampetakis. Flexible hybrid stores: Constraint-based rewriting to the rescue. In *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*, pages 1394–1397, 2016.

[25] Damian Bursztyn, François Goasdoué, and Ioana Manolescu. Optimizing reformulation-based query answering in RDF. In *EDBT*, 2015.

[26] Damian Bursztyn, François Goasdoué, and Ioana Manolescu. Reformulation-based query answering in RDF: alternatives and performance. *PVLDB*, 8(12):1888–1891, 2015.

[27] Damian Bursztyn, François Goasdoué, and Ioana Manolescu. Optimizing FOL reducible query answering: Understanding performance challenges. In *Proceedings of the ISWC 2016 Posters & Demonstrations Track co-located with 15th International Semantic Web Conference (ISWC 2016), Kobe, Japan, October 19, 2016.*, 2016.

[28] Damian Bursztyn, François Goasdoué, and Ioana Manolescu. Teaching an RDBMS about ontological constraints. *PVLDB*, 9(12):1161–1172, 2016.

[29] Damian Bursztyn, François Goasdoué, Ioana Manolescu, and Alexandra Roatis. Reasoning on web data: Algorithms and performance. In *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, pages 1541–1544, 2015.

[30] Andrea Calì, Georg Gottlob, and Thomas Lukasiewicz. Datalog$^{\pm}$: a unified approach to ontologies and integrity constraints. In *ICDT*, 2009.

[31] Andrea Calì, Georg Gottlob, and Thomas Lukasiewicz. A general datalog-based framework for tractable query answering over ontologies. In *PODS*, 2009.

[32] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. Tractable reasoning and efficient query answering in description logics: The DL-Lite family. *JAR*, 39(3), 2007.

[33] Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC*, 1977.

[34] Surajit Chaudhuri and Moshe Y. Vardi. Optimization of Real Conjunctive Queries. In *PODS*. ACM, 1993.

[35] Chandra Chekuri and Anand Rajaraman. Conjunctive query containment revisited. *Theor. Comput. Sci.*, 2000.

[36] Alexandros Chortaras, Despoina Trivela, and Giorgos B. Stamou. Optimized query rewriting for OWL 2 QL. In *CADE*, 2011.

[37] Debabrata Dash, Neoklis Polyzotis, and Anastasia Ailamaki. Cophy: A scalable, portable, and interactive index advisor for large workloads. *PVLDB*, 4(6), 2011.

[38] DBLP. `http://kdl.cs.umass.edu/data/dblp/dblp-info.html`.

[39] Dbpedia. `http://wiki.dbpedia.org/About`.

[40] A. Deutsch and V. Tannen. MARS: A System for Publishing XML from Mixed and Redundant Storage. In *VLDB*, 2003.

[41] J. Dittrich. Say No! No! and No! In *CIDR*, 2013.

[42] J. Duggan, A. J. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden, D. Maier, T. Mattson, and S. B. Zdonik. The BigDAWG polystore system. *SIGMOD Record*, 2015.

[43] A. Elmore, J. Duggan, M. Stonebraker, and al. A Demonstration of the BigDAWG Polystore System. *PVLDB*, 2015.

[44] R. Fagin, P. Kolaitis, R. Miller, and L. Popa. Data exchange: Semantics and query answering. In *ICDT*, 2003.

[45] Georges Gardarin, Fei Sha, and Zhao-Hui Tang. Calibrating the query optimizer cost model of IRO-DB, an object-oriented federated database system. In *VLDB*, 1996.

[46] Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, Antonella Poggi, Riccardo Rosati, Marco Ruzzi, and Domenico Fabio Savo. MASTRO: A reasoner for effective ontology-based data access. In *ORE*, 2012.

[47] Georgios Giannikis, Darko Makreshanski, Gustavo Alonso, and Donald Kossmann. Shared workload optimization. *PVLDB*, 7(6), 2014.

[48] François Goasdoué, Ioana Manolescu, and Alexandra Roatiş. Efficient query answering against dynamic RDF databases. In *EDBT*, 2013.

[49] François Goasdoué, Konstantinos Karanasos, Julien Leblay, and Ioana Manolescu. View selection in semantic web databases. *PVLDB*, 5(2), 2011.

[50] Georg Gottlob, Giorgio Orsi, and Andreas Pieris. Ontological queries: Rewriting and optimization. In *ICDE*, 2011. Keynote.

[51] Georg Gottlob, Giorgio Orsi, and Andreas Pieris. Query rewriting and optimization for ontological databases. *ACM TODS*, 39(3):25, 2014.

[52] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. *J. Web Sem.*, 3(2-3):158–182, 2005.

[53] Laura M. Haas, Mauricio A. Hernández, Howard Ho, Lucian Popa, and Mary Roth. Clio grows up: from research prototype to industrial tool. In *SIGMOD*, 2005.

[54] Alon Y. Halevy. Answering queries using views: A survey. *VLDBJ*, 10(4):270–294, 2001.

[55] T. Hearne and C. Wagner. Minimal covers of finite sets. *Discrete Mathematics*, 5:247–251, 1973.

[56] S. Idreos, M.L. Kersten, and S. Manegold. Database Cracking. In *CIDR*, 2007.

[57] Ioana Ileana, Bogdan Cautis, Alin Deutsch, and Yannis Katsis. Complete yet practical search for minimal query reformulations under constraints. In *SIGMOD*, 2014.

[58] Tomasz Imielinski and Witold Lipski. Incomplete information in relational databases. *JACM*, 31(4), 1984.

[59] A. Jindal, J.-A. Quiané-Ruiz, and J. Dittrich. WWHow! Freeing Data Storage from Cages. In *CIDR*, 2013.

[60] David S. Johnson and Anthony C. Klug. Optimizing Conjunctive Queries that Contain Untyped Variables. *SIAM J. Comput.*, 1983.

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

[62] Zoi Kaoudi, Iris Miliaraki, and Manolis Koubarakis. RDFS reasoning and query answering on DHTs. In *ISWC*, 2008.

[63] Konstantinos Karanasos, Asterios Katsifodimos, and Ioana Manolescu. Delta: Scalable data dissemination under capacity constraints. *PVLDB*, 7(4):217–228, 2013.

[64] Konstantinos Karanasos, Asterios Katsifodimos, Ioana Manolescu, and Spyros Zoupanos. Vip2p: Efficient XML management in DHT networks. In *Web Engineering - 12th International Conference, ICWE 2012, Berlin, Germany, July 23-27, 2012. Proceedings*, pages 386–394, 2012.

[65] M. Karpathiotakis, I. Alagiannis, T. Heinis, M. Branco, and A. Ailamaki. Just-in-time data virtualization: Lightweight data management with ViDa. In *CIDR*, 2015.

[66] Asterios Katsifodimos, Ioana Manolescu, and Vasilis Vassalos. Materialized view selection for XQuery workloads. In *SIGMOD*, 2012.

[67] Mélanie König, Michel Leclère, Marie-Laure Mugnier, and Michaël Thomazo. A sound and complete backward chaining algorithm for existential rules. In *RR*, 2012.

[68] Roman Kontchakov, Carsten Lutz, David Toman, Frank Wolter, and Michael Zakharyaschev. The combined approach to ontology-based data access. In *IJCAI*, pages 2656–2661, 2011.

[69] C. A. Lang, B. Bhattacharjee, T. Malkemus, S. Padmanabhan, and K. Wong. Increasing buffer-locality for multiple relational table scans through grouping and throttling. In *ICDE*, April 2007.

[70] J LeFevre, J. Sankaranarayanan, H. Hacigümüs, and al. MISO: souping up big data query processing with a multistore system. In *SIGMOD*, 2014.

[71] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *PVLDB*, 9(3):204–215, 2015.

[72] Maurizio Lenzerini. Ontology-based data management. In *CIKM*, 2011.

[73] A. Levy, A. Rajaraman, and J. Ordille. Querying Heterogeneous Information Sources Using Source Descriptions. In *VLDB*, 1996.

[74] Leonid Libkin and Limsoon Wong. Query Languages for Bags and Aggregate Functions. *Journal of Computer and System Sciences*, 1997.

[75] H. Lim, Y. Han, and S. Babu. How to Fit when No One Size Fits. In *CIDR*, 2013.

[76] Carsten Lutz, Inanç Seylan, David Toman, and Frank Wolter. The combined approach to OBDA: taming role hierarchies using filters. In *ISWC*, 2013.

[77] Stefan Manegold, Arjan Pellenkoft, and Martin L. Kersten. A multi-query optimizer for Monet. In *BNCOD*, 2000.

[78] Ioana Manolescu, Daniela Florescu, and Donald Kossmann. Answering XML queries on heterogeneous data sources. In *VLDB*, 2001.

[79] Ioana Manolescu, Konstantinos Karanasos, Vasilis Vassalos, and Spyros Zoupanos. Efficient xquery rewriting using multiple views. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 972–983, 2011.

[80] Thomas Neumann and Guido Moerkotte. Generating optimal DAG-structured query evaluation plans. *Computer Science - R&D*, 24(3), 2009.

[81] Thomas Neumann and Guido Moerkotte. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *ICDE*, 2011.

[82] Thomas Neumann and Gerhard Weikum. Scalable join processing on very large RDF graphs. In *SIGMOD*, 2009.

[83] Thomas Neumann and Gerhard Weikum. The RDF-3X engine for scalable management of RDF data. *VLDBJ*, 19(1):91–113, 2010.

[84] Giorgio Orsi and Andreas Pieris. Optimizing query answering under ontological constraints. *PVLDB*, 4(11), 2011.

[85] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems, Third Edition*. Springer, 2011.

[86] Héctor Pérez-Urbina, Ian Horrocks, and Boris Motik. Efficient query answering for OWL 2. In *ISWC*, 2009.

[87] François Picalausa, Yongming Luo, George H.L. Fletcher, Jan Hidders, and Stijn Vansummeren. A structural approach to indexing triples. In *ESWC*, 2012.

[88] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, Inc., NY, USA, 3 edition, 2003.

[89] Riccardo Rosati and Alessandro Almatelli. Improving query answering over DL-Lite ontologies. In *KR*, 2010.

[90] Max Schmachtenberg, Christian Bizer, and Heiko Paulheim. Adoption of the linked data best practices in different topical domains. In Peter Mika, Tania Tudorache, Abraham Bernstein, Chris Welty, Craig A. Knoblock, Denny Vrandecic, Paul T. Groth, Natasha F. Noy, Krzysztof Janowicz, and Carole A. Goble, editors, *The Semantic Web - ISWC 2014 - 13th International Semantic Web Conference, Riva del Garda, Italy, October 19-23, 2014. Proceedings, Part I*, volume 8796 of *Lecture Notes in Computer Science*, pages 245–260. Springer, 2014.

[91] J. Shute, R. Vingralek, B. Samwel, and al. F1: A Distributed SQL Database That Scales. In *PVLDB*, 2013.

[92] Konrad Stocker, Reinhard Braumandl, Alfons Kemper, and Donald Kossmann. Integrating semi-join-reducers into state-of-the-art query processors. In *ICDE*, 2001.

[93] Markus Stocker, Andy Seaborne, Abraham Bernstein, Christoph Kiefer, and Dave Reynolds. SPARQL basic graph pattern optimization using selectivity estimation. In *WWW*, 2008.

[94] Ion Stoica. Trends and challenges in big data processing. *PVLDB*, 9(13):1619, 2016.

[95] M. Stonebraker and U. Cetintemel. "One Size Fits All": An Idea Whose Time Has Come and Gone. In *ICDE*, 2005.

[96] Michaël Thomazo. Compact rewriting for existential rules. *IJCAI*, 2013.

[97] Anthony Tomasic, Louiqa Raschid, and Patrick Valduriez. Scaling heterogeneous databases and the design of disco. In *Proceedings of the 16th International Conference on Distributed Computing Systems, Hong Kong, May 27-30, 1996*, pages 449–457, 1996.

[98] Jacopo Urbani, Alessandro Margara, Ceriel J. H. Jacobs, Frank van Harmelen, and Henri E. Bal. Dynamite: Parallel materialization of dynamic RDF data. In *ISWC*, 2013.

[99] Jacopo Urbani, Frank van Harmelen, Stefan Schlobach, and Henri Bal. QueryPIE: Backward reasoning for OWL Horst over very large knowledge bases. In *ISWC*, 2011.

[100] Tassos Venetis, Giorgos Stoilos, and Giorgos B. Stamou. Incremental query rewriting for OWL 2 QL. In *Description Logics*, 2012.

[101] Roberto De Virgilio, Giorgio Orsi, Letizia Tanca, and Riccardo Torlone. NYAYA: A system supporting the uniform management of large sets of semantic data. In *ICDE*, 2012.

[102] OWL Web Ontology Language Overview. `http://www.w3.org/TR/owl-features`.

[103] R2RML: RDB to RDF mapping language. `http://www.w3.org/TR/r2rml/`.

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

[105] RDF Schema. `http://www.w3.org/TR/rdf-schema`.

[106] SPARQL protocol and RDF query language. `http://www.w3.org/TR/rdf-sparql-query`.

[107] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore: Sextuple indexing for Semantic Web data management. *PVLDB*, 2008.

[108] 3store. `http://www.aktors.org/technologies/3store`.

[109] AllegroGraph RDFStore Web 3.0 Database. `http://franz.com/agraph/allegrograph`.

[110] Apache Jena. `http://jena.apache.org`.

[111] Oracle Semantic Graphs. `http://www.oracle.com/technetwork/database-options/spatialandgraph/overview/rdfsemantic-graph-1902016.html`.

[112] Owlim. `http://owlim.ontotext.com`.

[113] Sesame. `http://www.openrdf.org`.

[114] Jingren Zhou, Per-Ake Larson, Johann-Christoph Freytag, and Wolfgang Lehner. Efficient exploitation of similar subexpressions for query processing. In *SIGMOD*, 2007.

[115] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter Boncz. Cooperative scans: Dynamic bandwidth sharing in a DBMS. In *VLDB*, 2007.

université PARIS-SACLAY
ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : Répondre efficacement aux requêtes Big Data en présence de contraintes

Mots clés : Web sémantique, Optimisation des requêtes, Répondre à des requêtes en présence de contraintes, Reformulation des requêtes, Polystores

Résumé : Les contraintes sont les artéfacts fondamentaux permettant de donner un sens aux données. Elles garantissent que les données sont conformes aux besoins des applications. L'objet de cette thèse est d'étudier deux problématiques liées à la gestion efficace des données en présence de contraintes.

Nous abordons le problème de répondre efficacement à des requêtes portant sur des données, en présence de contraintes déductives. Cela mène à des données implicites dérivant de données explicites et de contraintes. Les données implicites requièrent une étape de raisonnement afin de calculer les réponses aux requêtes. Le raisonnement par reformulation des requêtes compile les contraintes dans une requête modifiée qui, évaluée à partir des données explicites uniquement, génère toutes les réponses fondées sur les données explicites et implicites. Comme les requêtes reformulées peuvent être complexes, leur évaluation est souvent difficile et coûteuse.

Nous étudions l'optimisation de la technique de réponse aux requêtes par reformulation dans le cadre de l'accès aux données à travers une ontologie, où des requêtes conjonctives SPARQL sont posées sur un ensemble de faits RDF sur lesquels des contraintes RDF Schema (RDFS) sont exprimées. La thèse apporte les contributions suivantes. (i) Nous généralisons les langages de reformulation de requêtes précédemment étudieés, afin d'obtenir un espace de reformulations d'une requête posée plutôt qu'une unique reformulation. (ii) Nous présentons des algorithmes effectifs et efficaces, fondés sur un modèle de coût, permettant de sélectionner une requête reformulée ayant le plus faible coût d'évaluation. (iii) Nous montrons expérimentalement que notre technique améliore significativement la performance de la technique de réponse aux requêtes par reformulation.

Au-delà de RDFS, nous nous intéressons aux langages d'ontologie pour lesquels répondre à une requête peut se réduire à l'évaluation d'une certaine formule de la Logique du Premier Ordre (obtenue à partir de la requête et de l'ontologie), sur les faits explicites uniquement. (iv) Nous généralisons la technique de reformulation optimisée pour RDF, mentionnée ci-dessus, aux formalismes pour répondre à une requête LPO-réductible. (v) Nous appliquons cette technique à la Logique de Description DL-LiteR sous-jacente au langage OWL2 QL du W3C, et montrons expérimentalement ses avantages dans ce contexte.

Nous présentons également, brièvement, un travail en cours sur le problème consistant à fournir des chemins d'accès efficaces aux données dans les systèmes Big Data. Nous proposons d'utiliser un ensemble de systèmes de stockages hétérogènes afin de fournir une meilleure performance que n'importe lequel d'entre eux, utilisé individuellement. Les données stockées dans chaque système peuvent être décrites comme des vues matérialisées sur les données applicatives. Répondre à une requête revient alors à réécrire la requête à l'aide des vues disponibles, puis à décoder la réécriture produite comme un ensemble de requêtes à exécuter sur les systèmes stockant les vues, ainsi qu'une requête les combinant de façon appropriée.

université PARIS-SACLAY
ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Title : Efficient Big Data query answering in the presence of constraints

Keywords : Semantic Web, Query optimization, Query answering under constraints, Query reformulation, Hybrid stores

Abstract : Constraints are the essential artefact for giving meaning to data, ensuring that it fits real-life application needs, and that its meaning is correctly conveyed to the users. This thesis investigates two fundamental problems related to the efficient management of data in the presence of constraints.

We address the problem of efficiently answering queries over data in the presence of deductive constraints, which lead to implicit data that is entailed (derived) from the explicit data and the constraints. Implicit data requires a reasoning step in order to compute complete query answers, and two main query answering techniques exist. Data saturation compiles the constraints into the database by making all implicit data explicit, while query reformulation compiles the constraints into a modified query, which, evaluated over the explicit data only, computes all the answer due to explicit and/or implicit data. So far, reformulation-based query answering has received significantly less attention than saturation. In particular, reformulated queries may be complex, thus their evaluation may be very challenging.

We study optimizing reformulation-based query answering in the setting of ontology-based data access, where SPARQL conjunctive queries are answered against a set of RDF facts on which constraints hold. When RDF Schema is used to express the constraints, the thesis makes the following contributions. (i) We generalize prior query reformulation languages, leading to a space of reformulated queries we call JUCQs (joins of unions of conjunctive queries), instead of a single fixed reformulation. (ii) We present effective and efficient cost-based algorithms for selecting from this space, a reformulated query with the lowest estimated cost. (iii) We demonstrate through experiments that our technique drastically improves the performance of reformulation-based query answering while always avoiding "worst-case" performance.

Moving beyond RDFS, we consider the large and useful set of ontology languages enjoying FOL reducibility of query answering: answering a query can be reduced to evaluating a certain first-order logic (FOL) formula (obtained from the query and ontology) against only the explicit facts. (iv) We generalize the above-mentioned JUCQ-based optimized reformulation technique to improve performance in any FOL-reducible setting, and (v) we instantiate this framework to the DL-LiteR Description Logic underpinning the W3C's OWL2 QL ontology language, demonstrating significant performance advantages in this setting also.

We also report on current work regarding the problem of providing efficient data access paths in Big Data stores. We consider a setting where a set of different, heterogeneous storage systems can be used side by side to provide better performance than any of them used individually. In such a setting, the data stored in each system can be described as views over the application data. Answering a query thus amounts to rewrite the query using the available views, and then to decode the rewriting into a set of queries to be executed on the systems holding the views, and a query combining them appropriately.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France